Distributed and Parallel Databases (2020) 38:127–152
https://doi.org/10.1007/s10619-019-07264-z

DistriPlan: an optimized join execution framework for geo-distributed scientific data

Roee Ebenstein[1] · Gagan Agrawal[1]

Published online: 23 March 2019

Abstract
Scientific data is frequently stored across geographically distributed data repositories. Although there have been recent efforts to query scientific datasets using structured query operators, they have not yet supported joins across distributed data repositories. This paper describes a framework that supports join-like operations over multi-dimensional array datasets that are spread across multiple sites. More specifically, we first formally define join operations over array datasets and establish how they arise in the context of scientific data analysis. We then describe a methodology for optimizing such operations—components of our approach include enumeration algorithms for candidate plans, methods for pruning plans before they are enumerated, and a detailed cost model for selecting the best (cheapest) plan. We evaluate our approach using candidate queries, and show that the optimization effort is practical and profitable—query performance was improved significantly using our approach.

Keywords Scientific array database · Database optimizer · Distributed join · Database join

1 Introduction

The need for supporting scientific array data processing using declarative languages or structured operators has been raised in the past, and many systems addressing this need have been built [9,46,48]. These systems simplify the query specification process as compared to the ad-hoc approach and/or using low-level languages. Some of these systems require that the data be loaded into a database [9,48], whereas others can

✉ Roee Ebenstein
ebenstein.2@osu.edu

Gagan Agrawal
agrawal@cse.ohio-state.edu

1 Department of Computer Science and Engineering, The Ohio State University, 2015 Neil Avenue, Columbus, OH 43210-1277, USA

provide query processing capabilities working directly with low-level data layouts, such as a flat-file, or data in formats like NetCDF and HDF5 [19,46].

One of the issues that remain unaddressed is providing advanced query capabilities over data distributed across multiple geographically distributed repositories. The need for such functionality is increasingly arising as scientific data is growing in size and complexity. In supporting structured query operators over distributed data, most common query operators (selections, projections, and aggregations) are relatively straightforward to support. However, the classical join operator and its variants [37] are challenging to support when the data is at geographically disparate locations. In this paper, we focus on the challenge of executing and optimizing the join operator over geographically distributed array data.

As a motivation, consider the current status of dissemination of climate data. In the United States, much of the climate data is disseminated by Earth System Grid Forum (ESGF).[1] However, world-wide, climate data is also made available by agencies of other countries, such as those from Japan, Australia, and others. A climate scientist interested in comparing data across datasets collected from different satellites or agencies will need to run multiple queries across these repositories and to bind the data manually.

Similarly, array data related to other disciplines spread across multiple repositories as well—e.g. genetic variation data is found in 1000 (Human) Genomes Project and 1000 Plant Genomes Repository. Scientists are likely to have the need to produce join queries over different variables collected in these arrays. For example, linking behavioral data to genes or correlating particles behavior.

1.1 Existing approaches

Today, queries over distributed data are executed in one of several ways. First, often scientists simply copy all relevant data from repositories to their local environment and process it using existing tools—a solution very unlikely to be feasible as data sizes increase. Writing a workflow engine can be another option, but details of data movement, partitioning of the work, and its placement are handled by the developer. Moreover, individual operations in a workflow are still written in a low-level language.

Other approaches use ideas from the database (DB) domain to optimize overall query performance. Query optimization has been thoroughly researched before in the domain of relational data [14], however, these systems work on data at a central location or a cluster (with all nodes connected by using a Local Area Network—LAN). For tightly coupled settings, distributed query plans use rule based optimizers (RBOs) [3,31,39,54]. RBOs build execution plans from parsed queries based on pre-defined set of rules or heuristics. Heuristics are used to allow taking optimization decisions for each node of the parsed query tree separately, e.g. a *local decision*. An example for such a heuristic is: *“execute each query in the most distributed manner possible until data unionization or aggregation is necessary”*. Some of the DB query optimization approaches have been extended to geographically distributed data [3,11,19,46,51,52]. These extensions often use a heuristic to transfer all data to a central site and process

[1] See https://www.earthsystemgrid.org/about/overview.htm.

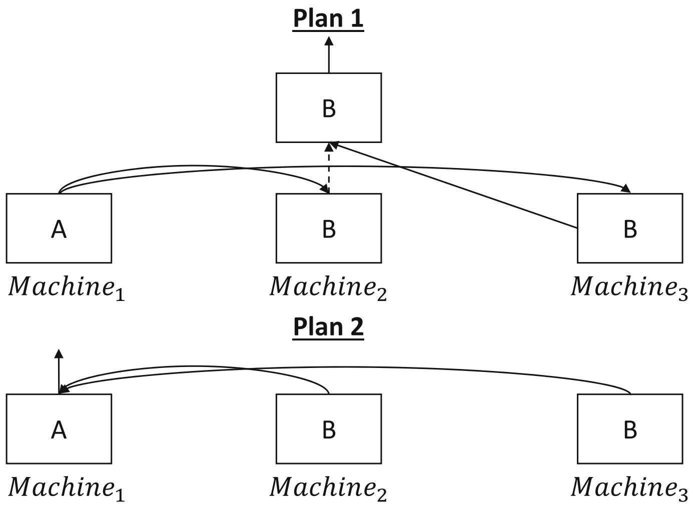

Fig. 1 Alternative ways of evaluating a query across multiple repositories

the query there. None of these extensions consider data shuffle as part of the query optimization process, assuming the most distributed heuristic cannot be improved.

Data processing has increasingly moved towards using implementations of the MapReduce concept [18]. Most systems in this space are limited to processing within a single cluster, where network latency is predictable and low. Note that joins can be supported over such systems using a high-level language, like in Hive [47]. In such cases, join optimization [1,50] is based on a set of rules or heuristics that will not be sufficient to fully optimize geo-distributed data join queries, as will be shown later. MapReduce systems have also been extended to geographically distributed data [28], but we are not aware of any work optimizing the join operation in such settings, which requires careful attention. Stratosphere [2] offers a cost based optimizer (CBO) for join queries. The CBO optimizes the operators order, and not the distribution of work, which is done with the simple heuristic mentioned in the previous paragraph—the more the parallelism, the faster a query should execute.

1.2 Challenges

To illustrate the challenges in executing join(-like) queries across multiple repositories, we take a specific example. Given a declarative query the system needs to decide what to do for providing the intended results—a process referred to as *building an execution plan*. An *execution plan* is a tree representation of ordered steps to perform for retrieving the expected results. Figure 1 shows two simple execution plans for executing a join between two one-dimensional arrays `A` and `B`. `A` is stored entirely on $Machine_1$ and is of length 100, whereas `B` is distributed between $Machine_2$ and $Machine_3$, each having an array of length 10. The *join selectivity*, which means the percent of the results holding the *join criteria* out of all possible results generated by a Cartesian multiplication of both joined relations [26], is 1%. In Plan 1, the variable

A is sent to both nodes $Machine_2$ and $Machine_3$. A partial resultset of length 10 is produced on each machine, and then the resultset from $Machine_3$ is sent to $Machine_2$ for combining with the local resultset. The final resultset is sent to the user. Plan 2 combines the distributed array B before performing the join on $Machine_1$.

Multiple challenges have been implicitly introduced here. Translating a query to a plan has been thoroughly researched before [12,14], yet building execution plans that consider different processing ordering on different nodes while the data is distributed among multiple nodes and sites have not. In our example, anticipating which of the two presented plans would execute faster is not trivial. A possible reason for choosing Plan 1 will be that more calculations are performed in parallel. However, with communication latencies taken into account, Plan 2 may be preferable. In addition, when two or more joins need to be performed, producing an execution plan becomes hard since the amount of distribution and evaluation options increases exponentially.

1.3 Contributions

This paper presents a methodology for executing and optimizing joins over geographically distributed array data. We will show that CBO's provide better optimization opportunities for our target setting, where simple heuristics are not sufficient. We develop algorithms for building distributed query execution plans while pruning not efficient and *isomorphic* plans. We introduce a cost model for distributed queries. The cost model considers the physical distribution of the data and the networking properties. We have extensively evaluated our query plan generation and execution modules and demonstrated their effectiveness.

2 Distributed joins

Join operations help compare data across multiple relations, and have been extremely common in the relational database world. In scientific array data analysis, joins are also essential for analyzing data and confirm hypotheses. As an example, consider a simple hypothesis such as "*when wind speed increases, the temperature drops*". Verifying this hypothesis using climate simulation outputs involves multiple joins across different datasets (because scientists commonly program these operations, many times they do not directly identify nor recognize a join was used). For detecting the change (*increases/drops*), each relation has to be compared with a subset of itself, i.e., we perform what is referred to as a *self join*. The pattern detection, i.e. relating the wind speed to the temperature value, is a *regular join* across both relations.

2.1 Formal definition

The definition provided here is very similar to the one found in relational database literature and only differs to match the domain it is imposed on. Here, we address dimensions and values instead of columnar values. In addition, aggregations are an inherent part of the scientific data array querying (including joins). In real scientific

Fig. 2 Query for walkthrough example

```
SELECT avgA.d1, avgA.value, avgB.value
FROM (SELECT A.d1 d1, AVG(A.value) value
      FROM A
      GROUP BY A.d1) as avgA,
     (SELECT B.d1 d1, AVG(B.value) value
      FROM B
      GROUP BY B.d1) as avgB
WHERE avgA.d1 = avgB.d1
```

settings, queries rarely do not require aggregations. Dimension reduction is conducted by using aggregations—data collected includes a higher dimensional setting than the one used in queries, resulting in the common use of aggregations. For example, when querying the average temperature we reduce multiple dimensions such as time and depth of sampling. Therefore, our engine uses aggregations by default, and our implementation minimizes its cost by integrating aggregations into the join operator.

Formally, the operator $\bowtie$ signifies a join—$A \bowtie_C^G B$ joins the relations A and B based on the set of conditions C and using the aggregation function G. C is a concatenation of conditions of the form $A' = B'$, using $\wedge$ (and) or $\vee$ (or), where A' and B' can be either a set of dimensions or the relation names themselves (the latter being referred to as *joining by value*). G (in the superscript) allows controlling the aggregation function used for the join (if no aggregation function is mentioned, the default function to be used if necessary is `AVG` (average)).

If C is not mentioned, i.e., the operation is $A \bowtie B$, common dimensions of both relations are joined based on their names, while the rest of the dimensions are aggregated by using an aggregation function. On the other hand, when the join explicitly states certain dimensions, an aggregation over the non-mentioned dimensions is expected. One could use the `rename` operator, $\rho_{from,to}$, which renames a dataset, variable, or a dimension for forcing name match when necessary. It may be necessary to allow joining only by values (without limiting the dimensions), an operator we mark as $\bar{\bowtie}$ and is similar to *Cartesian Multiplication*.

In Fig. 2 we translate the relational algebra query $A \bowtie_{A(d1)=B(d1)} B$ to SQL (Structured Querying Language [35]), a declarative language to represent queries [12]). We use SQL since it is a widely used and allows us to succinctly and clearly define what data processing is needed, ignoring the technical implementation issues. In Fig. 3 we show a walkthrough of this query execution. The common values along the joined dimension, $d1$, are 2 and 3. Since the other dimension in this array, $d2$, does not appear in the join criteria, aggregation has been used for it (as emphasized in the declarative query). An additional dimension has been added to record the data source, referred to in the figure as $d3$, this dimension in the relational model is translated to two columns.

In Fig. 4 we show a SQL for a subset of the query stated at the beginning of this section, demonstrating self joins. The SQL shown is equivalent to the request: *"Return the temperature difference for each day from its previous day"*. In the query, we first rename both relations from `TEMP` to `A` and `B`—this is done since this query is a *self-join* and therefore we need to be able to address both relations separately. We can look at relation `A` as if it represents specific day temperatures, while `B` represents the previous

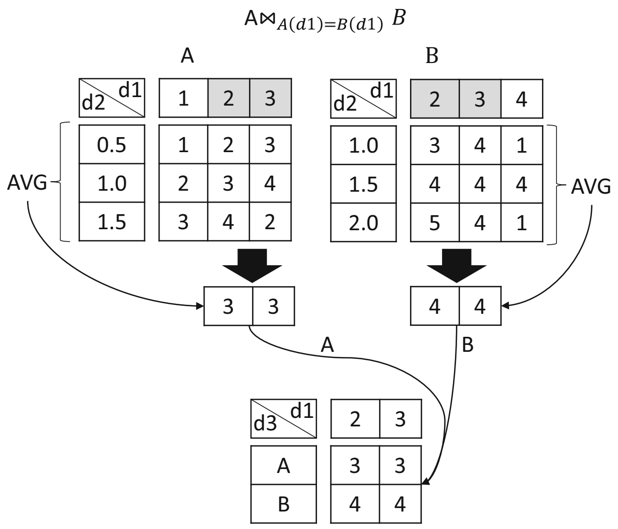

Fig. 3 A walkthrough the join process

Fig. 4 SQL for query example

```
SELECT A.temp - B.temp
FROM TEMP A, TEMP B
WHERE A.sample_date = B.sample_date - 1
AND A.longitude = B.longitude
AND A.latitude = B.latitude
```

day temperatures. Therefore, the output is, for each distinct latitude and longitude, the value of today's temperature minus yesterday's temperature, as requested.

2.2 Execution plans

An *execution plan*, or simply a plan, is a tree representation that contains processing instructions to an execution engine for providing the correct query results. While the SQL handles the "What", the plan targets the "How". Plans contain an hierarchical ordering of operators, each of which has at most two *children* nodes. When the plan tree is followed from the bottom to its top the intended query results are produced. In Fig. 5, we show two possible trees for the query: $A \bowtie B \bowtie C$ (we omitted the $(A \bowtie C) \bowtie B$ option for brevity). Each plan shows a different ordering of operations that produce the intended results.

Optimizing (or choosing) query plans, and especially data join optimization, has been extensively studied in the database community. The distinct part of our work is optimizing data joins when the data and execution are geo-distributed—for example, a situation where cluster one contains the temperatures in Canada and the U.S., while cluster two contains the temperatures elsewhere.

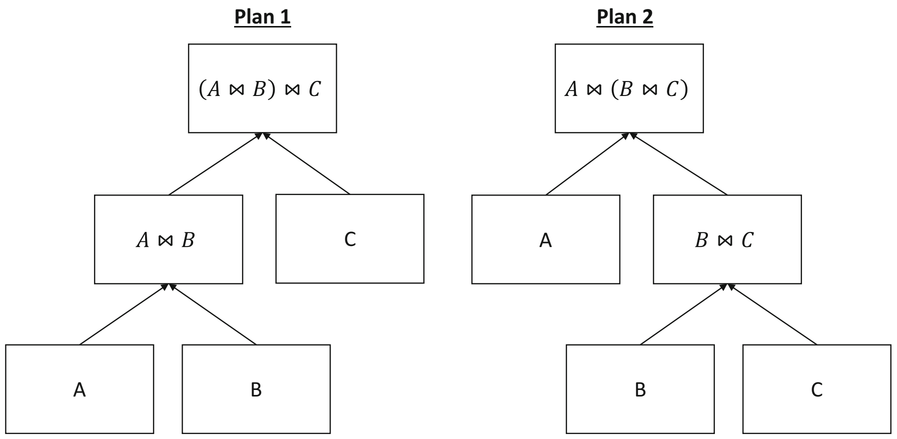

Fig. 5 Non-distributed plan samples

Our *distributed plan* has additional information within each node, as well as additional nodes, which provide the necessary information for parallel and distributed execution of queries. In Fig. 6 we demonstrate two different distributed execution plans for the simple query $A \bowtie B$ where A is an array distributed over three nodes, 1, 2, and 3, while B is an array distributed over 2 nodes, i.e, nodes 3 and 4. Plan 1 utilizes the most parallelism possible in this case—3 nodes process data in parallel, and afterwards all data are sent to node 1 where it is accumulated and *unionized* (*unionize* means combining multiple datasets to one). Plan 2 demonstrates the other extreme, in which all the data is copied to one node, which then processes it. Since only one node processed the data, there is no need to *unionize* data. These are just the two extreme plans, among many possible options.

More broadly, distributed execution plans need to represent parallel execution correctly, including representation of data communication among the nodes, unionization, and synchronization. There can be many options for such distribution. For example, as one extreme, all the datasets content is collected to a central node, and joined there (as presented in Fig. 6, Plan 2). After the join was processed, the results are tunneled either to the client, if this is the final plan step, or pipelined to the next step defined by the query execution plan. Another extreme can be as follows: since a join is performed between two relations, we choose one relation that we refer to as the *internal relation* (either relation can be the internal one, we choose the internal one based on the cost model presented later). The internal relation is kept stationary, while the other (*external relation*) is sent across the network to *all* the nodes that contain the internal relation. Subsequently, each receiving node executes the join/s and the results are sent forward according to the execution plan. We note some machines may contain many datasets (a common practice in real environments), and therefore it is optional for one node to communicate with another multiple times in this approach. Avoiding this behavior introduces a third option: we force each machine to transport at most one dataset of each relation it holds by unionization of the multiple datasets each machine has before processing. The rest of the processing can be done similarly to the way it

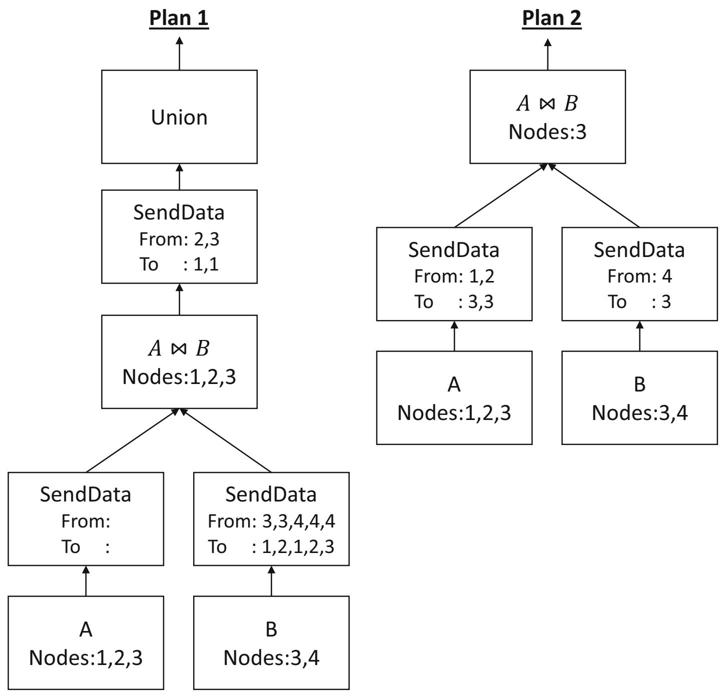

Fig. 6 Examples of distributed plans

is executed for the previous method—the sample in Fig. 6, Plan 1, demonstrates such a plan. The advantage of this approach is that it decreases the number of times data movement occurs and still allows parallel execution of joins.

3 Plan selection algorithms

3.1 Query plans

Formally, an execution plan for a given query is a tree representation of a query, where each node has at most two children, left and right. Each tree node represents either a *source* (relation, array, or dimension) or an *operator*. Each node outputs a data stream. Each operator node receives up to 2 incoming data streams. In the case a node represents an operator, the operator applies to the node's inputs.

In extending execution plans to distributed ones, additional operators are introduced (An example plan has already been introduced in Fig. 6). We introduce three new node types: *Sync*, *SendData*, and *Union*. *Sync* is synchronizing the execution of its dependent nodes, by waiting for all related sync nodes to start execution. All machines executing a sync node will start its execution at a different times, but will complete at the same time resulting in its dependent nodes beginning execution at the same time. In many cases synchronization is implicit and partial. A *SendData* node implies that machines that execute the children nodes need to send the produced results to a set of nodes. Thus, this operator achieves data distribution from a set of machines that produced a dataset to a (possibly distinct) set of machines that will later execute operations on that data. *Union* nodes are used to accumulate distributed data that was received from its children.

Each node has a *tag* that holds information needed for the operator execution. The tag includes what type of node it is, what subsetting conditions it executes (if applicable), statistical information, and operator specific data. For example, a SendData node's tag contains which data is sent, where from, and to which node.

3.2 Plan distribution algorithm

Our goal is to create an efficient CBO to find the optimal distribution of a query. A CBO is an alternative to a rule (or heuristic) based optimizer (RBO). RBO's are unlikely to build optimal plans in this case due to the complexity of our queries and diverse set of environments where they may be executed. Note that in a CBO, the optimizer produces a cost value that aligns with execution time, and thus helps to choose the lowest cost plan. CBO's do not aim to predict the actual execution time.

For implementing a CBO we need to first span or enumerate different execution plans and subsequently to evaluate these plans costs. We enumerate the plans using a two-step process: (1) choosing between different ordering of operators, leading to a set of *non-distributed plans*—these are built using a simple RBO since we span all options here. (2) enumerating all possible *distributed plans* corresponding to each non-distributed plan, each of which involves different choices for where the data is processed and required data movements. An advantage of this two-step method is that producing all non-distributed plans has been well researched previously [14]. Therefore, we focus here on the second step.

3.3 Pruning search space

The key challenge we face is that the number of distribution options for a given (non-distributed) plan can be extremely large. When all options of data sending are considered, a blowup of options occurs. For example, if n nodes are involved, n^n options of data movement exist. However, not all options have different costs or are even sensible candidates. As a simple example, one server can send its data to a neighboring node, accept data from the other node, or even do both (both nodes swap their data in this last setting). Given this, we must be able to prune the search space efficiently.

Our first observation is that many of these options are essentially a repeat of each other. For example, assuming homogeneity of nodes for simplicity, and given data distributed on 3 nodes, processing on 2 nodes should cost the same irrespective of which of the two nodes are used. Although it is not often guaranteed that nodes computing power and data sizes are homogeneous, within scientific data environments those assumptions often hold. We form the following two rules:

Rule 1: A node can receive data only if it does not send any data This rule prevents two nodes from swapping data with each other. The following is an analysis of the reduction in spanning options. Out of n nodes that have the data, we pick i nodes that will receive data. There are $n - i$ nodes left, which send their data to all options of the chosen i nodes. The summation of all of these reduce the number of options needed to be spanned.

Intuitively, a plan in which one node processes another node's data, while the other node processes the first node's data, is more expensive than a plan in which the nodes do not swap the data. Consider the cost of data processing of two datasets when node 1 and node 2 both process their own data, assuming data sizes are the same, or when node 1 processes node's 2 data and vice versa. The processing cost is the same, yet the data transportation, which does not exist in the first setting, exists on the second. The full scenario is more complex, yet this consideration helps clarifying why swapping data is more expensive than maintaining data stationary. Based on the above discussion, the optimal plan cannot be pruned by Rule 1 as shown in Theorem 1.

Theorem 1 *Rule 1 does not prune the optimal plan.*

Proof We will prove this claim by negation. Assume all machines have the same computing power (as justified before) and that the optimal plan contains a node receiving data while sending its own. Two options arise: (1) the number of machines processing the query is the same after the data transportation, or (2) the number of machines was decreased. If the number of machines is the same, it means data transportation was added, but the same computational power is used; a similar plan without the data swap must be cheaper. If the number of machines processing the query was decreased, we both—reduced the available computational power and added a data transportation overhead, again, a similar plan without the data transportation must be cheaper. In both cases, we found a plan with cheaper cost that did not include a node sending its data while receiving another node's data, therefore the original plan was not optimal, and the plan with a node that sends its own data while receiving data of another node should have been pruned. □

Rule 2: Isomorphism removal Rule 1 prunes options which are obviously more expensive than other plans, yet, many of the plans are still *isomorphic*. Clearly, there is no particular advantage of choosing one plan over other if they are isomorphic. We avoid the generation of isomorphic plans using the following approach. First, we assume that nodes are ordered and ranked by a unique identification number. With that, we require that a higher ranked node receives data from at least the same amount of nodes its lower ranked neighbor does. The nodes are ranked for the algorithm to provide consistent results and to enable addressing non-homogeneous environments in the future; the ranking can be dynamic, and can be used to address additional challenges (for example, non-homogeneous data sizes by assigning larger chunks of data to stronger nodes). For example, if the first node, assumed to have the highest rank, receives data from 4 nodes (including itself), the second node can receive data from at most 4 nodes, and so on. The optimal plan is again not pruned since all isomorphic plans evaluate to the same cost—the cost of isomorphic plans is the same since machines have the same processing power, are connected to the same network, the size of processed data is the same, and they perform the same calculation.

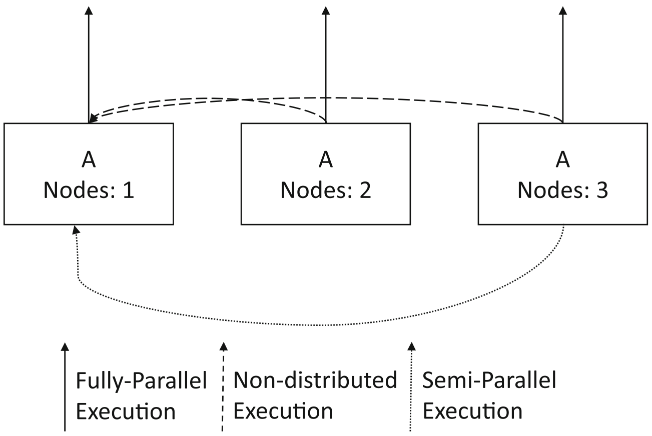

Fig. 7 Distribution options for an array (using rule 2) distributed over 3 nodes

An analysis of Rule 2 shows it prevents isomorphism. Assume we have n nodes and d datasets, while $n \leq d$. Generically, each of the n nodes process at least 1 dataset and all the nodes altogether process the d datasets. Represent this information in an array where the index of the array represents the node id and the content is the number of datasets processed on this node, for example [1, 3, 2, 5, 4] ($n = 5, d = 15$). Consider the setting after sorting the array ([5, 4, 3, 2, 1] for the example above)—the sorted setting holds Rule 2. Therefore, for any distribution setting chosen, there is a setting isomorphic to it, that holds Rule 2. The approach that issues such plans is referred to as the *reverse waterfall*, and we discuss it in more details later.

For example, assume array A is distributed over nodes ranked 1, 2, and 3. In Fig. 7 we demonstrate all possible distribution options following the given rule. Notice that any other option, given the nodes ranking, would either not make sense in a non-homogeneous setting, or would repeat an already existing option when nodes are homogeneous. E.g., consider the semi-parallel option of sending the data from node 2 to node 1 instead of from node 3, while node 3 keeps its data. This would either imply using a weaker node to process the same amount of data is preferable or will be equivalent to the semi-distributed option presented.

Algorithm 1 enumerates all distributed plans for a given non-distributed plan node. The input to this algorithm is a list of ranked nodes (ordered in an array), on which the data is distributed. The goal of the algorithm is to return all distribution options for populating each distribution node's tag. The algorithm iteratively builds all the options of sending data, by enumerating all options for processing nodes. In a high level, the algorithm begins by enumerating all the options for the number of nodes processing data (from 1 to the number of nodes). For each of these numbers, the algorithm iteratively builds all the options for distributing the data going from the most distributed approach to the least. More specifically, i represents the number of nodes processing data, i.e., by Rule 1 maintaining their data locally. The algorithm spans all options for i between 1 to the total number of nodes. In lines 5–7, we build a *base option* for the current i, which is the option where each "free node" (a node that is

Algorithm 1 Build plans without repetitions

```
1: function DataSendWithoutRepetitions(node list)
2:     options ← ∅
3:     n ← number of nodes (receiving data)
4:     for i ← 1..n do                                        ▷ i - number of nodes not sending data
5:         $\forall_j baseOptions.from[i] = i$                         ▷ Initialize senders
6:         $\forall_{j \leq i} baseOptions.to[j] = j$                   ▷ Send to itself
7:         $\forall_{j > i} baseOptions.to[j] = 1$                     ▷ All nodes send to first
8:         for j ← 1..min(i,n-i) do
9:             currOption ← duplicate(baseOption)
10:            for k ← 1..$(i-1) \times ((n-i)/i)$ do
11:                hasToGet ← $min(k, i-1)$
12:                $\forall_{t=0..hasToGet-1} currOption.to[i+t] = t+1$
13:                for l ← 1..(k-hasToGet) do
14:                    ar ← buildArraysOfOptoins(n,j,l)
15:                    options ∪ = span options based on ar
16:                end for
17:            end for
18:        end for
19:    end for
20:    return options
21: end function
```

not processing its own data) sends its data to the first node. In line 8 index j represents the number of nodes that receive data from other nodes (j of the base options is 1). In line 10 we define k to be the number of nodes not sending data to the first node. In line 15 all options spanned by Algorithm 2 are added (based on the sending/receiving pattern demonstrated by the resulted array)—we send data to higher ranked nodes from the lower ranked ones, assuring an optimal plan will not be pruned.

This algorithm provides all distribution options for a specific SendData node. Each option is used to fill a SendData node's tag.

In Algorithm 2 we use a *reverse waterfall* method to create all sending options. This algorithm outputs an array of numbers, each represents how many free nodes need to send data to the matching node. In general, we first determine the most distributed setting by initializing an array to contain the largest number of nodes each receiving node may receive data from (line 7). Then, in lines 11–30 we *bubble up* nodes by following Rule 2—the number of nodes sending data to the lower ranked nodes is systematically decreased while their higher ranked neighbors are increased.

For example, if there are 4 nodes, and 2 nodes are set to receive data, the *reverse waterfall* will create the following arrays: [1, 1, 0, 0], [2, 0, 0, 0] marking that for the first array both nodes receive data from one node each, while for the second array, one of the nodes sending data to the second node have been *bubbled up*, resulting in the first node receiving data from both *free* nodes. The reason we call this method the *reverse waterfall* is since one can look on the numbers as if when a number to the right decreases, the number to its left increases. Notice each array represents a unique option, since we always send data from the weakest node to the strongest one, and an array such as [1,1,0,0] would mean the lowest ranked node, 4, sends its data to the strongest one, node 1, while node 3 sends its data to node 2.

Algorithm 2 Build arrays to spread data holding rule 2

```
function BuildArraysOfOptoins(TotalNodes ReceivingNodes NodesToAssign)
                                                        ▷ How many nodes will be spread
    for i ← 1..NodesToAssign do
        actualRcvNodes ← $min(i, ReceivingNodes)$
                                                        ▷ How many nodes receive data
        for j ← 1..actualRcvNodes do
            ar ← mostDistributedOption
            arRet ∪ = ar
            lastPopulated ← LargestNonZeroIndex(ar)
            ar ← duplicate(ar)
            while lastPopulated ! = 0 do
                while ar[lastPopulated] ! = 0 do
                    t ← lastPopulated − 1
                    ar[t+1] ← ar[t+1] − 1
                    ar[t] ← ar[t] + 1
                    while ar[t] > ar[t-1] do
                        t ← t-1
                        if t == 0 then
                            break
                        end if
                        ar[t+1] ← ar[t+1] − 1
                        ar[t] ← ar[t] + 1
                    end while
                    if t == 0 then
                        break
                    end if
                    arRet ∪ = ar
                end while
                lastPopulated ← lastPopulated − 1
            end while
        end for
    end for
    return arRet
end function
```

4 Cost model

Cost models for facilitating query optimization in a non-distributed environment are well researched [6,14,20,27]. Enabling a CBO for distributed settings involves a number of additional challenges, particularly, capturing network latency and bandwidth. As mentioned before, the cost is calculated for the query, including all its terms and operators, with the goal that when comparing multiple plans, the plan with the lowest cost will also execute the fastest.

To motivate the need for a nuanced model, consider the following example. Suppose there are two Value Joins ($\bar{\bowtie}$) with a selectivity of 10%, and using three relations (A, B, and C), each containing 100 tuples. We expect the first join, between A and B, to produce 1,000 tuples, and the second one to produce 10,000 tuples. Assume the data is distributed in the following way: array A is distributed on nodes 1 and 2, array B is distributed on nodes 3, 4, 5, and 6, and array C is distributed over nodes 1 and 2. The plan that involves most parallelism would require copying array A to the nodes that contain array B, and doing the same for array C. This plan's execution involves:

Table 1 Costs of a node by operator

Operator	Cost (C(n))
Projection	$\frac{E(n)}{normalizer}$
Filter	$\frac{E(n \rightarrow left)}{normalizer} + \sum_{i \in dims} i \rightarrow length$
Distribute	$\frac{Penalty^{numOfNodes-1} \times E(n)}{PacketSize \times normalizer} + \frac{E(n)}{normalizer}$
Join over dimensions	$\frac{E(n \rightarrow left) + E(n \rightarrow right)}{normalizer}$
Join over values	$\frac{E(n \rightarrow left) \times E(n \rightarrow right)}{normalizer}$
Union	$\sum_{k \in \{n \rightarrow children\}} \frac{E(k)}{normalizer} \times k \rightarrow numOfNodes$
Sync	$Penalty^{numOfNodes-1}$
Source	$\frac{E(n)}{normalizer}$

sending data to 4 nodes from 2 (50 tuples), [an implicit] synchronization across 4 nodes, sending data to 4 nodes from 2 (50 tuples), [an implicit] synchronization across 4 nodes, and processing of 27,500 tuples on each processing node. If the same join is executed on nodes 1 and 2, its execution would involve: sending data to 2 nodes from 4 (25 tuples), [an implicit] synchronization across 2 nodes, sending data from 2 nodes to 1 (25 tuples), [an implicit] synchronization across 2 nodes, and processing 55,000 tuples on each node. If a relatively slow wide-area-network is connecting these nodes, the synchronizations and data movement operations can be expensive. Thus, it is quite possible that despite more computations on any given node, the second plan would execute faster.

The goal of our model is to assess these options and choose the best plan. Note that the cost model itself can be very dependent upon the communication and processing modalities used. We discuss the model presented at a high-level.

4.1 Costs

We define $C(n)$ as the cost of the node n and $E(n)$ as the size of the expected resultset after the node operator is executed. Costs are evaluated recursively, starting with the root, summing all costs together. Each node has up to two children nodes, denoted by $n \rightarrow left$ and $n \rightarrow right$, where the right child node is populated only for operators that accept two inputs, like joins. We also use $n \rightarrow children$ to mark both children together, left and right. For each operation, we list its cost in Table 1 and its expected resultset size in Table 2. The cost of an empty node, $C(NULL)$, is obviously defined to be 0. The selectivity, $n \rightarrow selectivity$, is evaluated beforehand, within the RBO, by using techniques established in the literature [16,25]. $Penalty$ represents the synchronization and communication overheads. We use the generic term $normalizer$

Table 2 Expected results by operator

Operator	Expected results (E(n))
Projection	$E(n \rightarrow left)$
Filter	$E(n \rightarrow left) \times n \rightarrow selectivity$
Distribute	$E(n \rightarrow left)$
Join over dimensions	$(E(n \rightarrow children) \times n \rightarrow selectivity$
Join over values	$E(n \rightarrow children) \times n \rightarrow selectivity$
Union	$\frac{\sum_{k \in \{n \rightarrow children\}} E(k) \times k \rightarrow numOfNodes}{n \rightarrow numOfNodes}$
Sync	$E(n \rightarrow left)$
Source	$n \rightarrow sourceCells$

as a tool to normalize the values returned to a reasonable and usable scale, to assure no overflow occur and to decrease side effects of large numbers math. For simplicity, we use a fixed set of penalties in our experiments, whose value depends upon the network configuration we use. This penalty value normally averages the networking delays (mainly latency) across the clusters. In a more heterogeneous environment, where the penalties average does not suffice, multiple penalty values can be used.

Filtering Multiple filtering operators are supported in our framework: $=$, $!=$, $<$, $>$, $<=$, and $>=$. Each filter has different volume or fraction of expected results, which can typically be estimated based on data statistics. In addition, there are multiple ways to scan the data and optimize the query, especially when index structures are available. Dimensional array values are unique, and in most cases are also sorted—both properties can be used for estimating selectivities and number of data scans.

Distributing In a distributed plan, we assess the cost of data movement among nodes. The cost of distribution has two components—the volume of data sent and the number of packets needed to be sent. Both are calculated for evaluation.

Joining Joins can only be run between 2 relations or dimensions at a time. Therefore, when multiple join criteria are mentioned in the join clause, they are nested one after another, a common scenario for array data since these often involve multiple dimensions. Consider joining by value or a non-contextual join, $\bar{\bowtie}$. The cost would simply be the multiplication of the joined array size, while the expected resultset size is the same value multiplied by the join selectivity—assuming a Cartesian-Like join is used. When the join is of a different type, the value should be adjusted to an equation controlling how many values are touched, evaluating the cost more accurately for that scenario. If the join is over dimensions, a variable reconstruction might be needed. In this case, the dimensions are joined first, and afterwards, the resulted dimensions are used to subset the variable. This process involves communicating the indices of the values that matched between the nodes executing the join to the nodes holding the variable data for efficiency. We do not focus on this step within our cost calculation here. For brevity, we assume merge sort is used for the dimensional joins since very rarely dimensions are not sorted. The total join time is the summation of both—the set of 1-D dimensional joins and the actual data join. An exhaustive discussion of this issue can be found in the existing literature [22,45].

4.2 Summarizing: choosing a plan

For a given query, a rule based optimizer builds all options for a non-distributed plan. Since the number of plans built is small, we distribute each of these plans separately by using the two pruning rules given before (Sects. 3.2 and 3.3). The cost of each plan is evaluated by using the cost model presented above and the cheapest plan is selected for execution.

Since all possible plans are evaluated, and only isomorphic or repeating plans are pruned, the cheapest plan is found. In the case multiple plans have the same cost, we use the simple heuristic: choose the least distributed plan among these plans. Contrary to expectation, we found the least distributed plan, among equal cost plans, runs somewhat faster due to slight overheads the cost model does not account for; mainly meta-data communication and data broadcast, which assumed to be parallelized but some portions of it are executed sequentially due to hardware limitations and the frameworks we use.

5 Evaluation

This section evaluates both our plan building methods and the performance of plan execution.

System We built two systems for the experimentation—a query optimizer and an execution engine. The query optimizer was written in C++, for efficiency, while the query execution engine, which executes plans produced by the optimizer, was written in Java, which allows its integration in many available frameworks. Because of the challenges of performing repeatable experiments in wide-area settings, communication latencies were introduced programmatically. The query optimizer accepts as input a query, and based on the algorithms provided in this paper creates an XML file with the cheapest plan. The query executor reads the execution plan from the XML file, follows the plan for executing the query (communication is implemented by using MPI [15]), and sends the query results to the user in a form of a NetCDF [44] file. The join algorithm itself and the order of execution are dynamic and provided by the optimizer. Because of scientific array data properties, dimensional joins are implemented by using sort-merge join, while skipping the sort since the data is already sorted. The array data join is implemented as described Sect. 2.

Queries All queries executed by the engine in the evaluation are value joins, $\bar{\bowtie}$, which differ by the amount of joined arrays, selectivities, and array sizes. We focus on joins since analytical querying requires full data scans, and often does not decrease the array sizes—all other operators are cheaper. For the evaluation, we use at most five relations (A, B, C, D, E). Query 1 (Q1) joins 3 tables by using 2 joins: $(A\bar{\bowtie}B)\bar{\bowtie}C$, Query 2 (Q2) joins 4 tables by using 3 joins: $((A\bar{\bowtie}B)\bar{\bowtie}C)\bar{\bowtie}D$, and last Query 3 (Q3) joins five tables by using 4 joins: $(((A\bar{\bowtie}B)\bar{\bowtie}C)\bar{\bowtie}D)\bar{\bowtie}E$. These three simple queries

Table 3 Join selectivities and processed dataset average sizes by query—N, number of tables involved in the join (N − 1 joins); Jn, the selectivity of the n^{th} join; AVG DS, average dataset size on each node

Q	N	J1 (%)	J2 (%)	J3 (%)	J4 (%)	AVG DS (MB)
1	3	5.0	0.5			381
2	4	1.0	1.0	0.1		762
3	5	0.1	0.5	0.1	1.0%	40

represent a wide range of real world queries—by using a small selectivity we can simulate subsetting conditions as well as joinability. Since the optimizer is mostly challenged by the number of nodes on which a relation is distributed on (and not the number of relations) we report results of joins of up to five relations. Using more relations for a query does not affect the feasibility of the approaches presented in this work, specifically—the amount of trees needed to be spanned is dependent on the distribution, and not the number of relations.

Data sizes The experiments were designed in a manner that each node stores an array with data sizes of between 8 MB to 800 MB. These sizes were chosen based on real datasets available on the ESGF portal. Also, the reported size is the size of an individual array read from a file (files are larger as they typically host multiple arrays). The array size mentioned is the local portion of each array, when we experiment over n nodes the total array size is the local one multiplied by n (for example, when experimenting over 800M arrays which are distributed over 10 nodes, the complete join is conducted over approximately 8GB, which is the actual array size). The data we used was generated in a pattern designed to enforce the given selectivities and holds the same patterns of real climate data.

In Table 3 we present, for each query, the join selectivities and the average size of processed array. The selectivities are between 0.1% to 5%, values which are commonly observed in Data Warehousing queries [43]. Since data is distributed over multiple nodes, the total data sizes vary for each query and for each array, therefore, we present the average dataset size.

Experimental setting All experiments in which we execute the queries ran on a cluster where each node has an 8 core, 2.53, GHz Intel(R) Xeon(R) processor with 12 GB of memory. All the experiments where we focus on building plans have been executed on a 4 cores Intel(R) Core(TM) i5, 3.3 Ghz, processors, with 2 GB of memory. All machines run Linux kernel version 2.6. The reported results are over three different consecutive runs, with no warm-up runs. Standard deviation is not reported since results were largely consistent. Unless mentioned otherwise, we set the optimizer penalty to be 400, equivalent to network latency of a WAN—this latency is based on data shown in related work [11,30,49].

Since no other environment provides the ability to join distributed scientific array data, we could not have compared our engine to any other. For example, SciDB [10] does not allow running generic joins as described here and does not support geographically distributed data. No implementation of (and/or higher level layer on top of) MapReduce can directly support joins over geographically distributed data.

Throughout the experiments, we assume no *multi-tenant* effect exists. The domain of multi-tenancy in context of data analytics is being researched separately [4,38], and considering it is beyond the scope of this work.

5.1 Pruning of query plans

We initially consider a simple join operation between two arrays, where each array is split across a given number of hosts. In Fig. 8 we show how the number of trees spanned for a join between two variables increases, showing the two pruning rules we have introduced drastically decrease the number of plans we need to evaluate. In the figure, there are multiple graphs. In each graph, the title includes the number of nodes the first joined relation is distributed over, the X-axis represents the number of nodes the second joined relations is distributed over (between 1 and 32), and the Y-axis holds a logarithmic scale for the number of spanned trees built for each of the settings. The continuous line represents the number of trees built by DistriPlan, while the other represents the number of trees need to be built when no pruning technique is used. Notice the difference in the initial values between the two graphs, and the scale changes, as the number of nodes increase between the graphs. For example, when both the arrays being joined are spread among 32 machines, our algorithms generate only 4,100 plans, out of the $\sim 1.4 \times 10^{71}$ optional ones. In practice, a dataset is likely to be split across a much smaller number of distributed repositories than 32. The maximum runtime of our optimizer was 0.46 s, while the average was 0.08 s, showing that query plans can be enumerated quickly with our method.

Next, we consider a join over three arrays, i.e., query Q2. In Table 4 we show how long it takes to span plans for the distribution of the non-distributed plan shown in Fig. 5, “Plan 1”. We consider a set of representative distribution options of the three datasets across different amount of nodes (each between 1 and 32 nodes). Each row considers a specific partitioning of the three datasets and shows the number of plans traversed and the time taken. As one sees, all execution times are under 2 s. Rules 1 and 2 given in Sect. 3.3 limit the increase in number of distribution options (for comparison, without rule 1 and 2, the first row would have had to span 2.63×10^{35} trees—clearly not a feasible option). In all cases we experimented with, the query performance improvement was substantial compared to the other, not optimized, plans. For example, we saved about 20 minutes of execution time compared to the median plan for the setting where the 3 involved datasets are partitioned across 8, 8, and 16 nodes respectively, with the final execution time only being 0.37 s. The same plan ran 20 s faster compared to the plan with most parallelism. Similar gains were seen in most experiments. Overall, we conclude the conditions provided in Sect. 3 are sufficient for handling cases where data to be queried is spread across a modest number of repositories.

5.2 Query execution performance improvement

In this experiment, we measure how effective our query plan selection method (and the underlying cost model) is. We execute three different plans for each query, the

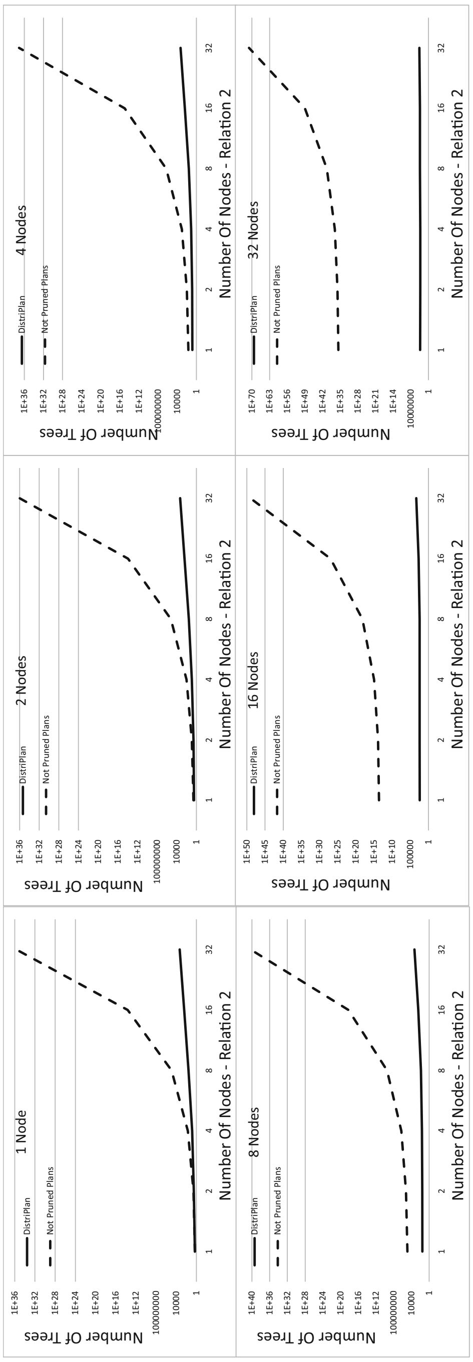

Fig. 8 Number of pruned candidate join plans as the number of hosts increase for a join between two variables. Each graph has a fixed number of nodes for the first relation, and increasing number of nodes for the second one on the X-axis

Table 4 Time required to span plans and number of spanned plans for a three-way join distributed among multiple nodes, the first columns present for each relation on how many nodes it is distributed

No. of nodes			Spanning time (s)	Options spanned
Relation partitioned				
A	B	C		
1	1	32	0.19	4,104
1	4	16	0.02	1,816
1	16	16	1.76	72,646
2	4	32	0.91	16,427
4	4	32	1.27	24,644
4	8	8	0.01	1,521
4	16	8	0.30	15,034
8	8	8	0.03	2,614
8	8	16	0.37	17,398
16	16	4	0.41	15,762
16	16	8	0.84	29,640

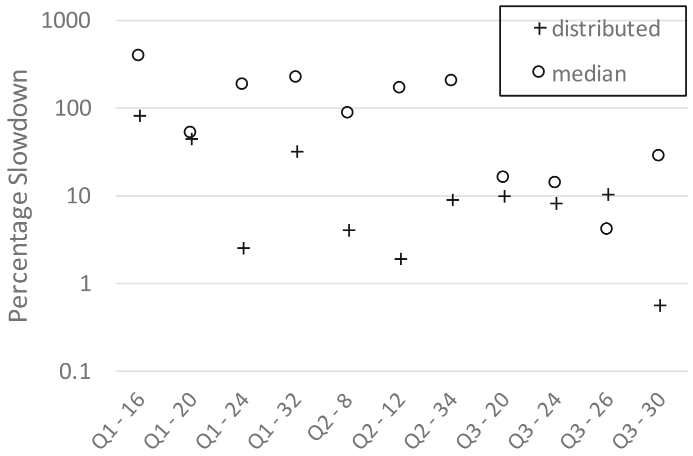

Fig. 9 Execution time slowdown (in %) for the most distributed and median plans compared to the cheapest plan. Each query is listed by the query ID, and a detailed distribution by array appears in Table 5

cheapest and median cost plans generated by our CBO model, and the plan with most parallelism. The latter is a commonly used heuristic in current systems such as Hive [8,47] and Stratosphere [2]. For each of the queries presented above, we first build a plan using our CBO for different distributed settings. The latency used in all experiments is of WAN, equivalent of 400 ms.

In Fig. 9 we report the increase in execution time of the median and most parallel plan versions, compared to the cheapest plan our optimizer selected. Along the X-axis, we list the query and the number of nodes that held the data for the query (each array is distributed differently for each query—this information can be seen in Table 5).

Table 5 The distribution of relations for each query—the specific distribution of each array for the queries presented in Fig. 9

Q	No	A	B	C	D	E
1	16	4	4	8		
1	20	4	8	8		
1	24	8	8	8		
1	32	8	16	8		
2	8	2	2	2	2	
2	12	2	2	4	4	
2	34	12	8	6	8	
3	20	4	4	4	4	4
3	24	7	5	3	4	5
3	26	4	6	6	4	6
3	30	7	4	4	4	4

Table 6 Queries execution time for the settings in Table 5

	Q1	Q1	Q1	Q2	Q3	Q3	Q3
	16	20	32	34	20	24	26
Cheapest	6S	20S	57S	1093S	31S	36S	146S
Distributed	11S	29S	75S	1191S	34S	39S	161S
Median	29S	30S	183S	3304S	36S	41S	152S

Along the Y-axis, we list the slowdown of the median plan and the most parallel plan compared to the cheapest plan. For example, Q1-16 cheapest plan ran 83% faster than the most parallel plan. We note that in certain specific cases (depending on the selectivities of the joins, among other factors) the cheapest plan is also the one with most parallelism. These cases are not shown. When arrays are distributed unevenly, the optimal plans are rarely the most distributed ones. In fact, the number of processing nodes is often smaller than the number of nodes that originally hold the inner joined array in the cheapest plan, i.e, at least one of the processing nodes processes data of the same array that is copied from another node.

In Table 6 we show the actual execution time for some chosen settings from Table 5. As can be seen, in all cases the cheapest plan executes the fastest. Furthermore, nearly in all cases the most parallel plan is faster than the median plan. In addition, plans that are mostly similar have small performance difference (such is the case for Q3–30 in Fig. 9).

A pattern uncovered here is a decrease in the improvement for some of the more complex queries (queries executing more joins and/or using a larger number of nodes). For example, in the case of Q1–16 (4, 4, 8), the slowdown for the most distributed query is 83%, while for Q3–20 (4, 4, 4, 4, 4) it is ∼10%. This behavior is rooted in parallelism that the more complex plans enable—for example, a three-way join forces sequentiality, while for a 4-way join, processing of some of the joins can be performed in parallel for certain plans.

Table 7 Performance slowdown percentage of a plan optimized for a specific, optimizer setting (Set), penalty but executed with different actual (Act) network setting—Q1 with a setting of 3, 5, 4

Set	Act			
	0 None	40 Cluster	400 WAN	4000 Extreme
0	0.00%	75.00%	11.20%	12.42%
40	0.00%	0.00%	10.37%	0.00%
400	75.00%	75.00%	0.00%	0.18%
4000	150.00%	150.00%	1.66%	0.00%

Table 8 Performance slowdown percentage of a plan optimized for a specific, optimizer setting (Set), penalty but executed with different actual (Act) network setting—Q3–24 in Table 5

Set	Act			
	0 None	40 Cluster	400 WAN	4000 Extreme
0	0.00%	18.18%	31.36%	17.64%
40	11.11%	0.00%	49.54%	36.96%
400	6122.22%	1436.36%	0.00%	19.90%
4000	159.26%	0.00%	26.81%	0.00%

We conclude the CBO approach for performance improvement is profitable. In all cases observed using our cost model, the fastest query to execute is the one the CBO evaluated to be the cheapest. In addition, the fastest query execution time was always faster than the median cost query and the most distributed plan (when both were different).

5.3 Impact of network latency

We executed the query optimizer and the resulting query plans to emulate four different cases. Here, we set the optimizer to build plans for a specific value of the network penalty, and execute each plan using different latency values than the one that matches the actual setting. We chose the following values for the penalty: 0 (no latency), 40 (Cluster), 400 (WAN), and 4000 (extreme), thus covering a wide variety of possible networks. We built plans optimized for each penalty and executed each plan multiple times using different actual settings.

In Tables 7 and 8 we show the percentage of the slowdown in execution time of the query optimized for a specific value compared to the query optimized for that value. For example, the value of the first row, second column, in the first table signifies that the optimized plan for a penalty value of 0 executed 75% slower than the plan optimized for 40 when the actual setting was the one intended for 40—the execution time of the cheapest plan optimized for a penalty of 0 is 7 s, the optimized plan for a penalty of 40 executes in 4 s. Similarly, in the second table for the first row last column, the execution time of a plan optimized for a penalty of 4000, which was also used for the actual one, is 2386 s while the plan optimized for a 0 penalty ran in this setting for 2807 s—a slowdown of 17.64%.

Overall, we can conclude that the penalty has to be selected carefully to reflect the actual setting—wrong values might harm performance as significantly as the right values improve it. It also shows that the best plan can vary significantly depending upon the latency, which implies that detailed cost modeling is critical.

6 Related work

Our work is related to the areas of (scientific) array data querying and distributed querying. We had earlier discussed some of the efforts in this area in Sect. 1.1. We now discuss other relevant efforts here.

Scientific arrays are often stored and distributed using portals, such as ESGF [7]. These portals use multiple methods in their backend to store and retrieve data. The most common data transportation technologies used is FTP [41], and in fact, querying operators have been integrated with one implementation of FTP, the GridFTP [19,46]. However, these systems can neither support distributed repositories, nor the Join operators. A more structured approach to querying scientific data involves array databases, and there is a large body of work in this area [5,9,13,17,32–34,36,42,48,55]. These systems require that the data be ingested by a central system, before it could be queried. Thus, they cannot support queries across multiple repositories. They also cannot directly operate on low-level scientific data. Finally, the query optimizer in Hive [47], which provides a high-level query interface to a MapReduce implementation [18], optimizes distributed querying over data within a cluster. This work primarily focuses on relational data within a single cluster, and the heuristics used assumes a very low latency—which is obviously not true in the case of geo-distributed arrays. The same is true for other research efforts in this area [29].

Optimization of distributed data (outside a cluster) was considered in the Volcano project [21,23,24]. However, this work did not include a cost-based optimizer that considered different options for distributing the processing and data movement, and uses implicit heuristics as well. WANalytics [51] is a recent proposal from Microsoft for developing analytics on geographically distributed datasets, but their target is not the join operator, nor scientific data—which makes this work an ideal continuation of that work. Stratosphere [2] offers a CBO for optimizing joins (among other operations). However, the CBO is used only for join-order, and choices among distributed computing options are made using a simple heuristic.

Another approach for joining distributed data may be to not evenly transport, or shuffle, the data. Examples could be instead of holding the machines that store Array 2 idle while the machines that hold Array 1 are processing all the data, to send both arrays to all optional processing nodes, and to utilize the machines that held the outer joined array as well (increasing the number of used machines) [40,53]. While these approaches minimize the communication traffic (assuming less communication leads to faster execution time), the same assumption of low latency and local cluster are used. Our approaches complement these methods as well and can be adjusted to fit these approaches. In addition, these approaches do not currently handle the situation of arrays stored in a distributed patters. It is unclear what performance hit adjusting these algorithms for that setting entails.

7 Conclusions and future work

In this paper, we have presented and evaluated a framework for optimized execution of array-based joins in geo-distributed settings. We developed a query optimizer, which prunes plans as it generates them. For our target queries, the number of plans is kept at a manageable level, and subsequently, a cost model we have developed can be used for selecting the cheapest plan. We have shown our pruning approach makes the plans spanning problem practical to solve. We evaluated our system and shown the cost model cheapest plan executes faster than more expensive plans. We have shown through experimentation that the penalty parameter introduced in the cost model is a critical one, and should be adjusted to fit the physical system setting carefully.

Our work can be extended in multiple directions. One of the ways to improve query plan generation will be to use learning algorithms, which can also learn multiple weights and penalties to fit each environment better. Similarly, creating an engine which finds the cheapest distributed execution plan directly from a query (without first enumerating all join options using an RBO) is an interesting challenge. Cases where data is not distributed evenly, and each node has different data distribution (skewed data), are an interesting research venue as well.

References

1. Afrati, F.N., Ullman, J.D.: Optimizing joins in a map-reduce environment. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 99–110. ACM (2010)
2. Alexandrov, A., Bergmann, R., et al.: The stratosphere platform for big data analytics. VLDB J. **23**, 939–964 (2014)
3. Apers, P.M., Hevner, A., et al.: Optimization algorithms for distributed queries. IEEE Trans. Softw. Eng. **1**, 57–68 (1983)
4. Aulbach, S., Grust, T., Jacobs, D., Kemper, A., Rittinger, J.: Multi-tenant databases for software as a service: schema-mapping techniques. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1195–1206. ACM (2008)
5. Baumann, P., Dehmel, A., et al.: The multidimensional database system RasDaMan. In: SIGMOD, pp. 575–577 (1998)
6. Belussi, A., Faloutsos, C.: Self-spacial join selectivity estimation using fractal concepts. ACM Trans. Inf. Syst. (TOIS) **16**(2), 161–201 (1998)
7. Bernholdt, D., Bharathi, S., et al.: The earth system grid: supporting the next generation of climate modeling research. Proc. IEEE **93**(3), 485–495 (2005)
8. Blanas, S., Patel, J.M., et al.: A comparison of join algorithms for log processing in MapReduce. In: SIGMOD, pp. 975–986. ACM (2010)
9. Brown, P.G.: Overview of SciDB: large scale array storage, processing and analysis. In: SIGMOD (2010)
10. Brown, P.G.: Overview of SciDB: large scale array storage, processing and analysis. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM (2010)
11. Carter, R.L., Crovella, M.E.: Server selection using dynamic path characterization in wide-area networks. In: Proceedings IEEE INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution, vol. 3, pp. 1014–1021. IEEE (1997)
12. Ceri, S., Gottlob, G.: Translating SQL into relational algebra: optimization, semantics, and equivalence of SQL queries. IEEE Trans. Softw. Eng. **11**(4), 324 (1985)
13. Cerveira Cordeiro, Ja.P., Câmara, G., et al.: Yet another map algebra. Geoinformatica **13**(2), 183–202 (2009)

14. Chaudhuri, S.: An overview of query optimization in relational systems. In: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 34–43. ACM (1998)
15. Clarke, L., Glendinning, I., Hempel, R.: The MPI message passing interface standard. In: Programming environments for massively parallel distributed systems, pp. 213–218. Springer (1994)
16. Cole, R.L., Graefe, G.: Optimization of dynamic query evaluation plans. SIGMOD Rec. **23**(2), 150–160 (1994). https://doi.org/10.1145/191843.191872
17. Cornacchia, R., Héman, S., et al.: Flexible and efficient IR using array databases. VLDB J. **17**(1), 151–168 (2008)
18. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
19. Ebenstein, R., Agrawal, G.: DSDquery DSI-querying scientific data repositories with structured operators. In: Big Data. IEEE (2015)
20. Getoor, L., Taskar, B., Koller, D.: Selectivity estimation using probabilistic models. In: ACM SIGMOD Record, vol. 30, pp. 461–472. ACM (2001)
21. Graefe, G.: Parallelizing the volcano database query processor. In: Compcon IEEE Computer Society International Conference, pp. 490–493. IEEE (1990)
22. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–169 (1993). https://doi.org/10.1145/152610.152611
23. Graefe, G.: Volcano—an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**, 120–135 (1994)
24. Graefe, G., McKenna, W.J.: The volcano optimizer generator: extensibility and efficient search. In: Data Engineering, pp. 209–218. IEEE (1993)
25. Graefe, G., Ward, K.: Dynamic query evaluation plans. In: ACM SIGMOD Record, vol. 18, pp. 358–366. ACM (1989)
26. Haas, P.J., Naughton, J.F., et al.: Fixed-precision estimation of join selectivity. In: ACM SIGACT-SIGMOD-SIGART, pp. 190–201. ACM (1993)
27. Haas, P.J., Naughton, J.F., Swami, A.N.: On the relative cost of sampling for join selectivity estimation. In: ACM SIGACT-SIGMOD-SIGART, pp. 14–24. ACM (1994)
28. Heintz, B., Chandra, A., Weissman, J.: Cloud computing for data-intensive applications. Cross-phase optimization in MapReduce, pp. 277–302. Springer, New York (2014)
29. Herodotou, H., Babu, S.: Profiling, What-If analysis, and cost-based optimization of MapReduce programs. Proc. VLDB Endow. **4**(11), 1111–1122 (2011)
30. Huang, C., Abdelzaher, T.: Towards content distribution networks with latency guarantees. In: Twelfth IEEE International Workshop on Quality of Service, 2004. IWQOS 2004, pp. 181–192. IEEE (2004)
31. Isard, M., Budiu, M., et al.: Dryad: distributed data-parallel programs from sequential building blocks. In: ACM SIGOPS, vol. 41, pp. 59–72. ACM (2007)
32. Lerner, A., Shasha, D.: AQuery: query language for ordered data, optimization techniques, and experiments. In: VLDB, pp. 345–356 (2003)
33. Marathe, A.P., Salem, K.: A language for manipulating arrays. In: VLDB, pp. 46–55 (1997)
34. Marathe, A.P., Salem, K.: Query processing techniques for arrays. VLDB J. **11**(1), 68–91 (2002)
35. Melton, J.: ISO/ANSI: database language SQL. ISO/IEC SQL revision. American National Standards Institute, New York (1992)
36. Mennis, J., Tomlin, C.D.: Cubic map algebra functions for spatio-temporal analysis. CaGIS **32**, 17–32 (2005)
37. Mishra, P., Eich, M.H.: Join processing in relational databases. ACM Comput. Surv. (CSUR) **24**, 63–113 (1992)
38. Narasayya, V., Das, S., Syamala, M., Chandramouli, B., Chaudhuri, S.: SQLVM: performance isolation in multi-tenant relational database-as-a-service (2013)
39. Olston, C., Jiang, J., Widom, J.: Adaptive filters for continuous queries over distributed data streams. In: SIGMOD, pp. 563–574. ACM (2003)
40. Polychroniou, O., Sen, R., Ross, K.A.: Track join: distributed joins with minimal network traffic. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 1483–1494. ACM (2014)
41. Postel, J., Reynolds, J.: File transfer protocol. RFC (1985)
42. Pullar, D.: MapScript: a map algebra programming language incorporating neighborhood analysis. Geoinformatica **5**(2), 145–163 (2001)

43. Reddy, N., Haritsa, J.R.: Analyzing plan diagrams of database query optimizers. In: Conference on Very Large Data Bases, pp. 1228–1239. VLDB Endowment (2005)
44. Rew, R., Davis, G.: NetCDF: an interface for scientific data access. IEEE Comput. Graph. Appl. **10**(4), 76–82 (1990)
45. Schneider, D.A., DeWitt, D.J.: A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. SIGMOD Rec. **18**(2), 110–121 (1989). https://doi.org/10.1145/66926.66937
46. Su, Y., Wang, Y., Agrawal, G., Kettimuthu, R.: SDQuery DSI: integrating data management support with a wide area data transfer protocol. In: SC, p. 47. ACM (2013)
47. Thusoo, A., Sarma, J.S., et al.: Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. **2**(2), 1626–1629 (2009)
48. van Ballegooij, A.R.: RAM: a multidimensional array DBMS. In: EDBT 2004 workshops (2005)
49. Veeramani, S., Masood, M.N., Sidhu, A.S.: A PACS alternative for transmitting DICOM images in a high latency environment. In: 2014 IEEE Conference on Biomedical Engineering and Sciences (IECBES), pp. 975–978. IEEE (2014)
50. Viglas, S.D., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of the 29th International Conference on Very Large Data Bases, vol. 29, pp. 285–296. VLDB Endowment (2003)
51. Vulimiri, A., Curino, C., et al.: Wanalytics: analytics for a geo-distributed data-intensive world. In: CIDR (2015)
52. Wang, F.: Query optimization for a distributed geographic information system. Photogramm. Eng. Remote Sens. **65**, 1427–1438 (1999)
53. Warneke, D., Kao, O.: Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. IEEE Trans. Parallel Distrib. Syst. **22**(6), 985–997 (2011)
54. Yu, Y., Gunda, P.K., Isard, M.: Distributed aggregation for data-parallel computing: interfaces and implementations. In: SIGOPS, pp. 247–260. ACM (2009)
55. Zhang, Y., Kersten, M., Ivanova, M., Nes, N.: SciQL: bridging the gap between science and relational DBMS. In: IDEAS, pp. 124–133 (2011)