DistriPlan - An Optimized Join Execution Framework for Geo-Distributed Scientific Data

Roee Ebenstein
The Ohio State University
Columbus, Ohio
ebenstein.2@osu.edu

Gagan Agrawal
The Ohio State University
Columbus, Ohio
agrawal@cse.osu.edu

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 an 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.

CCS CONCEPTS

•Information systems → **Parallel and distributed DBMSs; Information retrieval query processing; Retrieval efficiency;** **•Applied computing** → Earth and atmospheric sciences;

ACM Reference format:
Roee Ebenstein and Gagan Agrawal. 2017. DistriPlan - An Optimized Join Execution Framework for Geo-Distributed Scientific Data. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

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 [7, 39, 41]. These systems simplify the specification of processing as compared to the adhoc approach and/or using low-level languages. Some of these systems require that the data be loaded into a database [7, 41], whereas others can provide query processing capabilities working directly with low-level data layouts, such as a flat-file, or data in formats like NetCDF and HDF5 [15, 39].

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 [33] 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. Similarly, data related to other disciplines spreads across multiple repositories as well– e.g. genetic variation data is found in 1000 (Human) Genomes Project and 1000 Plant Genomes Repository.

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 locally 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 have been thoroughly researched before in the domain of relational data [11], however, these systems work on data at a central location or a cluster. For tighly coupled settings, distributed query plans use Rule Based Optimizers (RBO) [2, 27, 34, 46]. 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

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

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

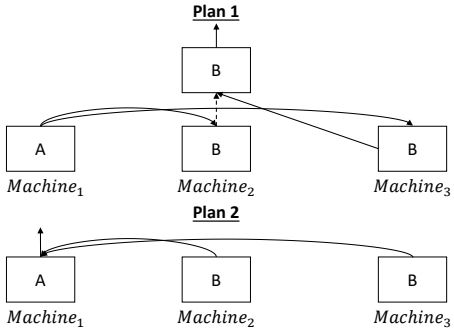

Figure 1: Alternative Ways of Evaluating a Query Across Multiple Repositories

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 [2, 8, 15, 39, 44, 45].

Data processing has increasingly moved towards using implementations of the Map-Reduce concept [14]. Most systems in this space are limited to processing within a single cluster. Note that joins can be supported over such systems using a high-level language, like in Hive [40]. In such cases, join optimization [1, 43] is based on a set of rules or heuristics that will not be sufficient to optimize for geo-distributed data join queries. MapReduce systems have also been extended to geographically distributed data [24], but we are not aware of any work optimizing the join operation in such settings, which requires careful attention.

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. Fig. 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 Cartersian multiplication of both joined relations [22], 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 reuslt-set 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 have been thoroughly researched before [9, 11], 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 in this paper that Cost-Based Optimizers (CBO) 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. 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 datasets. 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 is a *regular join* across two relations.

2.1 Formal Definition

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 in 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*). The given 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 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

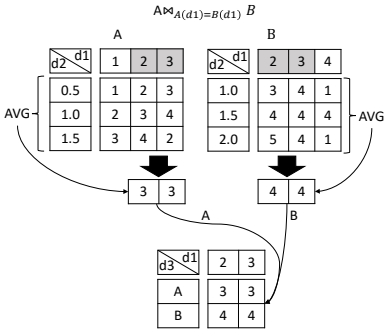

Figure 2: A walkthrough the join process

may be necessary to allow joining only by values (without limiting the dimensions), an operator we mark as $\tilde{\bowtie}$ and is similar to *cartesian multiplication*.

In Figure 2 we show a walkthrough the execution of the join $A \bowtie_{A.d1=B.d1} B$. The common values along the joinned 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. An additional dimension has been added to record the data source, referred to in the figure as $d3$.

```
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
```

Figure 3: SQL for Query Example

In Figure 3 we show a SQL (Structured Querying Language [31], a declarative language to represent queries [9]) for a subset of the query stated in the beginning of this section. 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 seperately. We can look at relation `A` as if it represents a specific day temperature, while `B` represents the temperature of the day before. Therefore, the output is, for each distinct latitute 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. The plan contains an hierarchical ordering of operators, each of which has at most two *children* nodes. When the tree is followed from the bottom to its top the intended query results are returned. In Figure 4, we show two possible trees for the query: $A \bowtie B \bowtie C$ (we omitted the $A \bowtie C \bowtie B$ option). Each plan shows a different ordering of operations that provides the intended results.

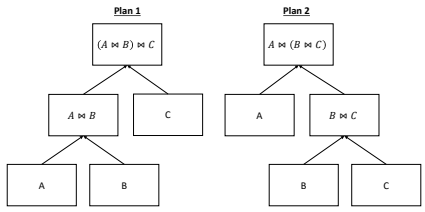

Figure 4: Non-distributed plan samples

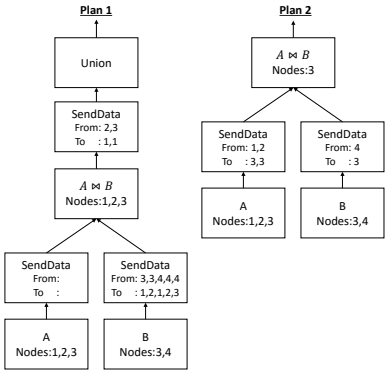

Figure 5: Examples of Distributed Plans

Optimizing (or choosing) query plans, and especially join optimization, has been extensively studied in the database community. The distinct part of our work is optimizing 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.

A *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 Figure 5 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 Figure 5, 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*. 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. Yet another option can be as follows: we force each machine to have at most one dataset of each relation it holds by unionization of the multiple datasets each machine has before processing, and then rest of the processing can be done similarly to the way it is executed for the previous method – the sample in Figure 5, 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 the plan to a distributed plan, additional operators are introduced (An example plan has already been introduced in Figure 5). We introduce three new node types: *Sync*, *SendData*, and *Union*. *Sync* delays the beginning of its parent operation until all of the children nodes have completed execution. 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 distribution of data 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.

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 Cost Based Optimizer (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.

For implementing a CBO we need to first span or enumerate different execution plans and subsequently evaluate costs of these plans. 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 movement. An advantage for this two-step method is that producing all non-distributed plans has been well researched previosly [11]. Our focus is on the second step, and the details are presented next.

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 of the same node, or even do both. Given this, we must be able to prune the search space.

Our first observation is that many of these options are essentially a repeat of each other. For example, assuming homogeneity of nodes, and given data distributed on 3 nodes, processing on 2 nodes should cost the same irrespective of which of the two nodes process it. 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. Thus, there are $n - i$ nodes left, which send their data to all options of the chosen i nodes.

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. Therefore, the optimal plan cannot be pruned by this rule.

Rule 2: Isomorphism Removal. Rule 1 prunes options which are 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. 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 (unless the nodes are not homogenuous; in which case the highest rank node will always be cheaper to process on than any other node, maintaining the correctness of the claim).

For example, assume array A is distributed on nodes ranked 1, 2, and 3. In Figure 6 we demonstrate all possible distribution options following the given rule. Notice that any other

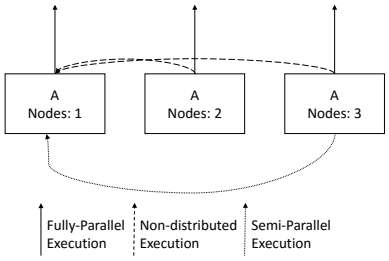

Figure 6: Distribution Options for An array (Using Rule 2)

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. For example, 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 itself. This would either imply using a weaker node to process the 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. 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 (*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 not processing its own data) send it 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 Rule2 – the number of nodes sending data to the lower ranked nodes are 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 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 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:          ∀_j baseOptions.from i = i                          ▷ Initialize senders
6:          ∀_{j≤i} baseOptions.to j = j                        ▷ Send to itself
7:          ∀_{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 × n − ii do
11:                 hasToGet ← min k, i − 1
12:                 ∀_{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
```

4 COST MODEL

Cost models for facilitating query optimization in a non-distributed environment are well researched [4, 11, 16, 23]. Enabling a Cost Based Optimizer (CBO) for distributed settings involves a number of additional challenges, particularly, network latency and bandwidth.

To motivate the need for a nuanced model, consider the following example. Suppose there are two Value Joins ($\bowtie$) 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: sending data to 4 nodes from 2 (50 tuples), synchronization across 4 nodes, sending data to 4 nodes from 2 (50 tuples), 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:

Algorithm 2 Build arrays to spread data holding rule 2

```
1: function BUILDARRAYSOFOPTOINS(TotalNodes ReceivingNodes NodesToAssign)
2:                                          ▷ How many nodes will be spread
3:     for i ← 1..NodesToAssign do
4:         actualRcvNodes ← mini, ReceivingNodes
5:                                          ▷ How many nodes receive data
6:         for j ← 1..actualRcvNodes do
7:             ar ← mostDistributedOption
8:             arRet ∪ ar
9:             lastPopulated ← LargestNonZeroIndex(ar)
10:            ar ← duplicate(ar)
11:            while lastPopulated ! = 0 do
12:                while ar[lastPopulated] ! = 0 do
13:                    t ← lastPopulated − 1
14:                    ar[t+1] ← ar[t+1] − 1
15:                    ar[t] ← ar[t] + 1
16:                    while ar[t] > ar[t-1] do
17:                        t ← t-1
18:                        if t == 0 then
19:                            break
20:                        end if
21:                        ar[t+1] ← ar[t+1] − 1
22:                        ar[t] ← ar[t] + 1
23:                    end while
24:                    if t == 0 then
25:                        break
26:                    end if
27:                    arRet ∪ ar
28:                end while
29:                lastPopulated ← lastPopulated − 1
30:            end while
31:        end for
32:    end for
33:    return arRet
34: end function
```

Operator	Cost
Projection	$Cn = \frac{En}{normalizer}$
Filter	$Cn = \frac{En \to left}{normalizer} + \underset{i \in dims}{} i \to length$
Distribute	$Cn = \frac{Penalty^{numOfNodes-1} \times En}{PacketSize \times normalizer} + \frac{En}{normalizer}$
Join over dimensions	$Cn = \frac{En \to left + En \to right}{normalizer}$
Join over values	$Cn = \frac{En \to left \times En \to right}{normalizer}$
Union	$Cn = \underset{k \in \{n \to left \cup n \to right\}}{} \frac{Ek}{normalizer} \times k \to numOfNodes$
Sync	$Cn = Penalty^{numOfNodes-1}$
Source	$Cn = \frac{En}{normalizer}$

Table 1: Costs of a node by operator

Operator	Expected Results
Projection	$En = En \to left$
Filter	$En = En \to left \times n \to selectivity$
Distribute	$En = En \to left$
Join over dimensions	$En = En \to left + En \to right \times n \to selectivity$
Join over values	$En = En \to left \times En \to right \times n \to selectivity$
Union	$En = \frac{\overset{k \in \{n \to left \cup n \to right\}}{Ek \times k \to numOfNodes}}{n \to numOfNodes}$
Sync	$En = En \to left$
Source	$En = n \to sourceCells$

Table 2: Expected Results by operator

sending data to 2 nodes from 4 (25 tuples), synchronization across 2 nodes, sending data from 2 nodes to 1 (25 tuples), synchronization across 2 nodes, and processing 55,000 tuples on each node. If a relatively slow wide-area-network is connecting these nodes, 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 *Cn* as the cost of the node *n* and *En* as the size of the expected resultset after the node operator is executed. Costs are evaluated recursively, starting with the root. Each node has up to two children nodes, denoted by $n \to left$ and $n \to right$, where the right child node is populated only for operators that accepts two inputs, like joins. For each operation we list its cost in Table 1 and its expected resultset size in Table 2. The cost of an empty node, *CNULL*, is obviously defined to be 0. The selectivity, $n \to selectivity$, is evaluated beforehand, within the RBO, by using techniques established in literature [12, 21]. We define *dims* to be the list of array dimensions, and *length* to be the size of a dimension. *Penalty* represents the synchronization and communication overheads. We use a fixed penalty in our experiments, whose value depends upon the network configuration we use.

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.

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, $\bowtie$. 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. 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 join process of dimensional joins, since in most cases dimensions are already sorted. Exhaustive discussion of this issue can be found in the existing literature [18, 38].

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 (Subsection 3.2 and Subsection 3.3) each of these plans separately. The cost of each plan is evaluated by the cost model presented above and the cheapest plan is selected for execution.

Since all plans possible 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 plan, runs somewhat faster due to slight overheads the cost model does not account for.

5 EVALUATION

This section evaluates both our plan building methods and the performance of executing the plans. 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, 3.3Ghz, processors, with 2GB of memory. All machines run Linux kernel version 2.6. The reported results are over 3 different consecutive runs, with no warm-up runs. Standard deviation is not reported since results were largely consistent.

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. Because of the challenges of performing repeatable experiments in a wide-area settings, communication latencies were introduced programmatically.

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

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

Queries: All queries executed by the engine in the evaluation are value joins, $\bowtie$, which differ by the amount of joined arrays, selectivities, and array sizes. For the evaluation we use at most 5 relations (A, B, C, D, E). Query 1 (Q1) joins 3 tables by using 2 joins: $A \bowtie B \bowtie C$, Query 2 (Q2) joins 4 tables by using 3 joins: $A \bowtie B \bowtie C \bowtie D$, and last Query 3 (Q3) joins 5 tables by using 4 joins: $A \bowtie B \bowtie C \bowtie D \bowtie E$. We believe these three simple queries represent a wide range of real world queries – by using small selectivities we can simulate subsettings as well as joinability.

Data Sizes: The experiments were designed in a manner that each node operates on data sizes of between 8 MB to 800 MB. These sizes were chosen based on real datasets available on the ESG portal. The data we used was generated in a pattern designed to enforce the given selectivities (because indexes are not currently used, the data origin is insignificant). Unless mentioned otherwise, we set the penalty of the optimizer to be 400ms, based on data shown in related work [8, 26, 42].

In Table 3 we present, for each query, the join selectivities and the average size of array processed. The selectivities are between 0.1% to 5%, values which are commonly observed in Data Warehousing queries [37]. 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.

5.1 Pruning of Query Plans

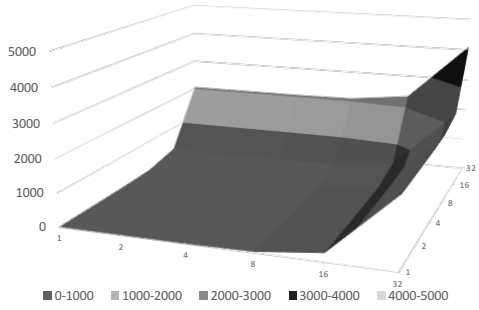

Figure 7: Number of pruned Candidate Join Plans As the Number of Hosts Increases - Join Between Two Variables

We initially consider a simple join operation between two arrays, where each array is split across a given number of hosts. In Figure 7 we show how the numbers of trees spanned for a join between two variables increases by using the two

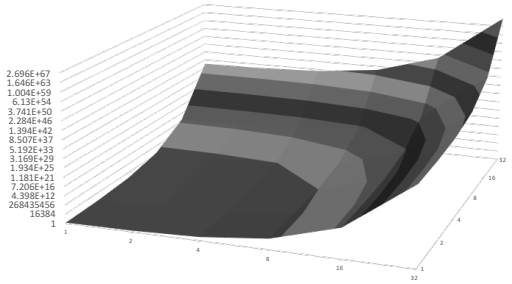

Figure 8: Number of Total Candidate Join Plans As the Number of Hosts Increases - Join Between Two Variables

No. of Nodes Relation Partitioned			spanning	options
A	B	C	time (S)	spanned
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

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

pruning rules we have introduced. For comparasion, in Figure 8 we present the number of spanned options without utilizing these rules – notice the scales are substantially different. For example, when both the arrays being joined are spread among 32 machines, there are 4,100 options the spanning algorithm produces (compared to $\sim 1.4 \times 10^{71}$ without pruning). In practice, a dataset is likely to be split across a much smaller number of distributed repositories than 32. The maximum run time of the algorithm was 0.46 Seconds, while the average was 0.08 Seconds, showing that query plans can be enumerated quickly with our method.

Next we consider a join over three arrays, i.e., the query Q2. In Table 4 we show how long it takes to span plans for the distribution of the non-distributed plan shown in Figure 4, "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 see, all execution times are under 2 seconds. Rules 1 and 2 given in Subsection 3.3

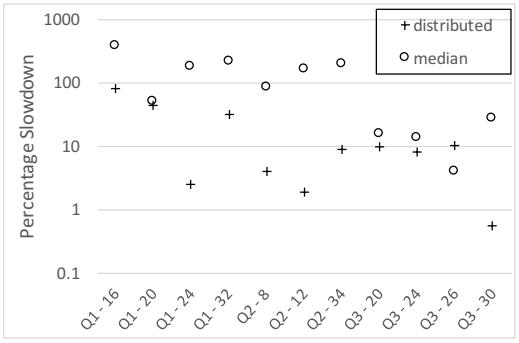

Figure 9: Execution Time Slowdown (in %) For the Most Distributed and Median Plans Compared to The Cheapest Plan. Each query is listed by the query ID, and a detailed distribution by array appears in Table 5

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 regular plan, for the setting when the 3 datasets involved are partitioned across 8, 8, and 16 nodes, respectively, with the final execution time only being 0.37 Seconds. The same plan ran 20 seconds faster compared to the plan with most parallelism. Similar gains were seen in most experiments. Overall, we conclude the conditions provided in Section 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 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 [6, 40]). 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 400 ms because of the reasons mentioned before.

In Figure 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). 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

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 5: The distribution of relations for each query – the specific distribution of each array for the queries presented in Figure 9

	Q1 16	Q1 20	Q1 32	Q2 34	Q3 20	Q3 24	Q3 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

Table 6: Queries Execution Time for the Settings in Table 5

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 array in the cheapest plan, i.e, at least one of the processing nodes processes data 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 execute the fastest. In addition, nearly in all cases the most parallel plan is faster than the median plan. In some of the cases, the plans are mostly similar and the performance difference is small (such is the case for Q3-30 in Figure 9).

An interesting pattern uncovered 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 $\sim$10%. This behavior is caused by parallelism that the more complex plans enable – for example, a 3-way join forces sequentiality, while for a 4-way join processing of some of the joins can be performed in parallel for certain plans.

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 then 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 built plans for a specific value of the network penalty, and execute each plan using different latency values than the one that matches the penalty. We chose the following values for the penalty: 0 (no latency), 40 (Cluster), 400 (WAN), and 4000 (extreme). We built plans optimized for each penalty, and executed each plan multiple times using different penalties.

Exp. \ 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 7: Performance Slowdown Percentage of a Plan Optimized for a specific, Expected(Exp), penalty but Executed with Different Actual(Act) penalty values - Q1 with a setting of 3,5,4

Exp. \ 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%

Table 8: Performance Slowdown Percentage of a Plan Optimized for a specific, Expected(Exp), penalty but Executed with Different Actual(Act) penalty values - Q3-24 in Table 5

In Tables 7 and 8 we show the percentage of 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 penalty was 40 – the execution time of the cheapest plan optimized for a penalty of 0 is 7 Seconds, the optimized plan for a penalty of 40 executes in 4 seconds. 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 2,386 seconds while the plan optimized for a 0 penalty ran in this setting for 2,807 seconds - 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 Section 1.1. We now discuss other relevant efforts here.

Scientific arrays are often stored and distributed using portals, such as ESG [5]. These portals use multiple methods in their backend to store and retrieve data. The most common data transportation technologies used is FTP [35], and in fact, querying operators have been integrated with one implementation of FTP, the GridFTP [15, 39]. 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 [3, 7, 10, 13, 28–30, 32, 36, 41, 47]. 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 [40], which provides a high-level query interface to a MapReduce implementation [14], optimizes distributed querying over data within a cluster. This work primarily focuses on data within a single cluster, and the heuristics used assume 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 [25].

Optimization of distributed data (outside a cluster) was considered in the Volcano project [17, 19, 20]. 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 [44] is a recent proposal from Microsoft for developing analytics on geograpically distributed datasets, but their target is not the join operator, nor scientific data – which makes this work an ideal continuation of that work.

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 setting. 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 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 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] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[2] P. M. Apers, A. Hevner, et al. Optimization algorithms for distributed queries. *Software Engineering, IEEE Transactions*, (1):57–68, 1983.

[3] P. Baumann, A. Dehmel, et al. The Multidimensional Database System RasDaMan. In *SIGMOD*, pages 575–577, 1998.

[4] A. Belussi and C. Faloutsos. Self-spacial join selectivity estimation using fractal concepts. *ACM Transactions on Information Systems (TOIS)*, 16(2):161–201, 1998.

[5] D. Bernholdt, S. Bharathi, et al. The earth system grid: Supporting the next generation of climate modeling research. *Proceedings of the IEEE*, 93(3):485–495, 2005.

[6] S. Blanas, J. M. Patel, et al. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986. ACM, 2010.

[7] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[8] R. L. Carter and M. E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, volume 3, pages 1014–1021. IEEE, 1997.

[9] S. Ceri and G. Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Transactions on software engineering*, 11(4):324, 1985.

[10] J. a. P. Cerveira Cordeiro, G. Câmara, et al. Yet Another Map Algebra. *Geoinformatica*, 13(2):183–202, June 2009.

[11] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[12] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. *SIGMOD Rec.*, 23(2):150–160, May 1994.

[13] R. Cornacchia, S. Héman, et al. Flexible and efficient IR using array databases. *VLDB J.*, 17(1):151–168, 2008.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] R. Ebenstein and G. Agrawal. Dsdquery dsi - querying scientific data repositories with structured operators. 2015.

[16] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *ACM SIGMOD Record*, volume 30, pages 461–472. ACM, 2001.

[17] G. Graefe. Parallelizing the volcano database query processor. In *Compcon IEEE Computer Society International Conference.*, pages 490–493. IEEE, 1990.

[18] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[19] G. Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering*, 6(1):120–135, 1994.

[20] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Data Engineering*, pages 209–218. IEEE, 1993.

[21] G. Graefe and K. Ward. Dynamic query evaluation plans. In *ACM SIGMOD Record*, volume 18, pages 358–366. ACM, 1989.

[22] P. J. Haas, J. F. Naughton, et al. Fixed-precision estimation of join selectivity. In *ACM SIGACT-SIGMOD-SIGART*, pages 190–201. ACM, 1993.

[23] P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In *ACM SIGACT-SIGMOD-SIGART*, pages 14–24. ACM, 1994.

[24] B. Heintz, A. Chandra, and J. Weissman. *Cloud Computing for Data-Intensive Applications*, chapter Cross-Phase Optimization in MapReduce, pages 277–302. Springer New York, New York, NY, 2014.

[25] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[26] C. Huang and T. Abdelzaher. Towards content distribution networks with latency guarantees. In *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 181–192. IEEE, 2004.

[27] M. Isard, M. Budiu, et al. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS*, volume 41, pages 59–72.

ACM, 2007.
[28] A. Lerner and D. Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.
[29] A. P. Marathe and K. Salem. A Language for Manipulating Arrays. In *VLDB*, pages 46–55, 1997.
[30] A. P. Marathe and K. Salem. Query processing techniques for arrays. *VLDB J.*, 11(1):68–91, 2002.
[31] J. Melton. Iso/ansi: Database language sql. *ISO/IEC SQL Revision. New York: American National Standards Institute*, 1992.
[32] J. Mennis and C. D. Tomlin. Cubic map algebra functions for spatio-temporal analysis. *CaGIS*, 32:17–32, 2005.
[33] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.
[34] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, pages 563–574. ACM, 2003.
[35] J. Postel and J. Reynolds. File transfer protocol. 1985.
[36] D. Pullar. MapScript: A Map Algebra Programming Language Incorporating Neighborhood Analysis. *Geoinformatica*, 5(2):145–163, June 2001.
[37] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *conference on Very large data bases*, pages 1228–1239. VLDB Endowment, 2005.
[38] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *SIGMOD Rec.*, 18(2):110–121, June 1989.
[39] Y. Su, Y. Wang, G. Agrawal, and R. Kettimuthu. SDQuery DSI: integrating data management support with a wide area data transfer protocol. In *SC*, page 47. ACM, 2013.
[40] A. Thusoo, J. S. Sarma, et al. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.
[41] A. R. van Ballegooij. RAM: a multidimensional array DBMS. In *EDBT 2004 Workshops*, pages 154–165, 2005.
[42] S. Veeramani, M. N. Masood, and A. S. Sidhu. A pacs alternative for transmitting dicom images in a high latency environment. In *Biomedical Engineering and Sciences (IECBES), 2014 IEEE Conference on*, pages 975–978. IEEE, 2014.
[43] S. D. Viglas, J. F. Naughton, and J. Burger. 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-Volume 29*, pages 285–296. VLDB Endowment, 2003.
[44] A. Vulimiri, C. Curino, et al. Wanalytics: Analytics for a geo-distributed data-intensive world. In *CIDR*, 2015.
[45] F. Wang. Query optimization for a distributed geographic information system. *Photogrammetric engineering and remote sensing*, 65:1427–1438, 1999.
[46] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SIGOPS*, pages 247–260. ACM, 2009.
[47] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: Bridging the Gap Between Science and Relational DBMS. In *IDEAS*, pages 124–133, Sept. 2011.