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

Roe 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.

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 [3, 14, 16, 14, 8]. These systems simplify the query specification process as compared to the ad-hoc approach and/or using low-level languages.

One of the issues that remain unaddressed is providing advanced query capabilities over data distributed across multiple geographically distributed repositories. The classical join operator and its variants [9, 12] are challenging to support in this setting. 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). 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.

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*. Figure 1 shows two simple execution plans for executing a join between two one-dimensional arrays A and B. A is stored entirely on *Machine₁* and is of length 100,

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

whereas B is distributed between *Machine₂* and *Machine₃*, each having an array of length 10. The *join selectivity* is 1%. In Plan 1, the variable A is sent to both nodes *Machine₂* and *Machine₃*. A partial resultset of length 10 is produced on each machine, and then the resultset from *Machine₃* is sent to *Machine₂* for combining with the local resultset. Plan 2 combines the distributed array B before performing the join on *Machine₁*.

Multiple challenges have been implicitly introduced here. Translating a query to a plan has been thoroughly researched before [6, 5], 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. Bringing communication latencies into account might make Plan 2 favorable over the highly parallelized Plan 1.

2 DISTRIBUTED JOINS

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 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 is AVG, average). If C is not mentioned, 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. It may be necessary to allow joining only by values (without limiting the dimensions), an operator we mark as $\bowtie_v$ and is similar to *Cartesian Multiplication*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SSDBM '17, Chicago, IL, USA

© 2017 ACM. 978-1-4503-5282-6/17/06...\$15.00

DOI: 10.1145/3085504.3085529

Figure 2: A walkthrough the join process

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

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. Plans contain an hierarchical ordering of operators, each of which has at most two *children* nodes.

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 can be collected to a central node, and joined there. Another extreme can be to execute the join in the most distributed setting possible, and collect the results afterwards.

3 PLAN SELECTION ALGORITHMS

Query Plans

Formally, an execution plan for a given query is a tree representation of the 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 three operators are introduced: *Sync*, *SendData*, and *Union*. *Sync* delays the beginning of its parent operation until all of the children nodes have completed execution – synchronization is often 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 distributes 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 from its children.

Our goal is to create an efficient Cost Based Optimizer (CBO) to find the optimal distribution of a query. Note that in a CBO, the optimizer produces a cost value that aligns with execution time, and thus helps choosing 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*. 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.

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. 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 main observation is that many of these options are a repeat of each other. 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. 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 (assuming homogeneity of both data sizes and computing power).

Rule 2: Isomorphism Removal. Rule 1 prunes options which are obviously more expensive than other plans, yet, many of the plans are still *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. Since isomorphic plans evaluate to the same cost, the optimal plan is not pruned.

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. In a high level, the algorithm begins by enumerating all the options for 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 not processing its own data) sends its data to the first

Algorithm 1 Build Plans Without Repetitions

```

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

```

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

Table 1: Costs of a node by operator

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 using the pruning rules are added.

4 COST MODEL

The presented cost model 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.

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

Table 2: Expected Results by operator

Costs are evaluated recursively, starting with the root, summing all costs together. Each node has up to two children nodes, denoted by $n \rightarrow \text{left}$ and $n \rightarrow \text{right}$, where the right child node is populated only for operators that accept two inputs, like joins. We also use $n \rightarrow \text{children}$ 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 defined to be 0. The selectivity, $n \rightarrow \text{selectivity}$, is evaluated beforehand, within the RBO, by using techniques established in literature [7, 10]. *Penalty* represents the synchronization and communication overheads. For simplicity we use a fixed set of penalties in our experiments, whose value depends upon the network configuration. This penalty value normally averages the networking delays (mainly latency) across sites.

Filtering: Each filter condition 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. The cost of joining 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.

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 (Section 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 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 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

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: $((A \bowtie B) \bowtie C) \bowtie D \bowtie E$.

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, 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 on 10 nodes the complete join is conducted over approximately 8GB, which is the actual array size).

The selectivities of the join queries are between 0.1% to 5%, values which are commonly observed in Data Warehousing queries [13]. 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, 3.3Ghz, processors, with 2GB of memory. All machines run Linux kernel version 2.6. Unless mentioned otherwise, we set the penalty of the optimizer to be 400, equivalent to network latency of a WAN – this latency is based on data shown in related work [17, 4, 11].

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 Figure 3 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 each graph the title includes

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

Table 3: 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

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 to 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. In practice, a dataset is likely to be split across a much smaller number of distributed repositories than 32. The maximum run time of our optimizer 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., query Q2. In Table 3 we show how long it takes to span plans for the distribution of a non-distributed plan (of a 3 relations join). 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. Rules 1 and 2 given in Section 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 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. 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.

Figure 3: 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.

Figure 4: 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 4

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 4: The distribution of relations for each query – the specific distribution of each array for the queries presented in Figure 4

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 [15, 2] and Stratosphere [1]. 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.

In Figure 4 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

	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 5: Queries Execution Time for the Settings in Table 4

query – see Table 4). 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.

In Table 5 we show the actual execution time for some chosen settings from Table 4. As can be seen, in all cases

Act. \ Exp.	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 6: Performance Slowdown Percentage of a Plan Optimized for a specific, Expected(Exp), penalty but Executed with Different Actual(Act) network setting - Q1 with a setting of 3,5,4

Act. \ Exp.	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 7: Performance Slowdown Percentage of a Plan Optimized for a specific, Expected(Exp), penalty but Executed with Different Actual(Act) network setting - Q3-24 in Table 4

the cheapest plan execute 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 Figure 4).

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 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 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 6 and 7 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 setting was the one intended for 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.

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 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 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 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.

REFERENCES

- [1] A. Alexandrov, R. Bergmann, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 2014.
- [2] S. Blanas, J. M. Patel, et al. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986. ACM, 2010.
- [3] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, 2010.
- [4] R. L. Carter and M. E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *INFOCOM*. IEEE, 1997.
- [5] S. Ceri and G. Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE TSE*, 1985.
- [6] S. Chaudhuri. An overview of query optimization in relational systems. In *SIGACT-SIGMOD-SIGART*. ACM, 1998.
- [7] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. *SIGMOD Rec.*, 23(2):150–160, May 1994.
- [8] R. Ebenstein and G. Agrawal. Dsdquery dsi-querying scientific data repositories with structured operators. In *Big Data*. IEEE, 2015.
- [9] R. Ebenstein, N. Kamat, and A. Nandi. Fluxquery: An execution framework for highly interactive query workloads. In *SIGMOD*. ACM, 2016.
- [10] G. Graefe and K. Ward. Dynamic query evaluation plans. In *ACM SIGMOD Record*, volume 18, pages 358–366. ACM, 1989.
- [11] C. Huang and T. Abdelzaher. Towards content distribution networks with latency guarantees. In *IWQOS*. IEEE, 2004.
- [12] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 1992.
- [13] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*. ACM, 2005.
- [14] Y. Su, Y. Wang, et al. SDQuery DSI: integrating data management support with a wide area data transfer protocol. In *SC*. ACM, 2013.
- [15] A. Thusoo, J. S. Sarma, et al. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2009.
- [16] A. R. van Ballegooij. RAM: a multidimensional array DBMS. In *EDBT 2004 Workshops*, 2005.
- [17] S. Veeramani, M. N. Masood, et al. A pacs alternative for transmitting dicom images in a high latency environment. In *IECBES*. IEEE, 2014.