Cost Estimation Across Heterogeneous SQL-Based Big Data Infrastructures in Teradata IntelliSphere®

Kassem Awada
Teradata Labs, CA, USA
kassem.awada@teradata.com

Mohamed Y. Eltabakh*
Teradata Labs, CA, USA
mohamed.eltabakh@teradata.com

Conrad Tang
Teradata Labs, CA, USA
conrad.tang@teradata.com

Mohammed Al-Kateb
Teradata Labs, CA, USA
mohammed.al-kateb@teradata.com

Sanjay Nair
Teradata Labs, CA, USA
sanjay.nair@teradata.com

Grace Au
Teradata Labs, CA, USA
grace.au@teradata.com

ABSTRACT

In big data ecosystems, it is becoming inevitable to query data that span multiple heterogeneous data sources (remote systems) to build meaningful querying and analytical workflows. Existing work that aims at unifying heterogeneous systems into a single architecture lacks the fundamental aspect of efficient cost estimation of SQL-based operators over remote systems. The problem is fundamental because all modern optimizers are cost-based, and without accurate cost estimation for each query operator, the generated plans can be way off the optimal plan. Nevertheless, the problem is mostly overlooked by existing systems because the focus is either on homogeneous distributed RDBMSs in which cost estimation is already extensively studied, or on fully heterogeneous engines in which SQL querying and SQL query optimization are not applicable (or at least are not the core problem). In this paper, we propose a comprehensive remote-system cost estimation module for SQL operators, which is a core module within the Teradata IntelliSphere architecture. The proposed module encompasses three costing approaches, namely *logical-operator*, *sub-operator*, and *hybrid* approaches, which are suitable for black box, open box, and a mix of black and open box systems, respectively. The cost estimation module leverages analytical and deep learning models with novel techniques for efficient extrapolation when needed. The techniques presented in this paper are modular and can be adopted by other systems. Extensive experimental evaluation shows the practicality and efficiency of the proposed system.

1 INTRODUCTION

There has been an increasing necessity, especially in big data applications, for managing and querying data that span multiple heterogeneous data sources (remote systems) [12, 13, 31]. The number of the remote system types is increasing dramatically, each system has unique inherent characteristics and processing capabilities, some systems are *openbox* with well-known internal details while others are *blackbox* with very little knowledge about their internals—and many levels in between, and each system offers different levels of sophistication w.r.t. query planning and optimization. Although such interconnectivity and interoperability create unprecedented opportunities for advanced analytics and data sciences, the unification of such diverse systems in a single architecture and the orchestration of the overall processing among them represent a classical challenging problem of many facets.

Several architectures have been proposed to address different aspects of the problem including *federated systems* [8, 11, 24], *polystore systems* [13], and *data integration and warehousing systems* [10, 12, 28, 31]. A big bulk of federated systems' research has focused on distributed relational database systems where distributed transaction processing, concurrency control, recovery control, and replica management have been extensively studied [9, 14, 20, 24]. Other research focuses on heterogeneous federated systems, where schema mapping, query translation, conflict management, and mediation design are the core addressed issues [10, 12, 28, 31]. More recent polystore systems, e.g., the BigDAWG system [13], target transparent unification and access across multiple backend systems of different data models, e.g., array, graph, streaming, and relational models. Although query optimization is a core component of BigDAWG, building an advanced cost estimation module is not the current focus as reported in [13]. Finally, the data integration and warehousing systems focus on offline data integration issues in contrast to online ad-hoc querying and query optimization.

"Teradata IntelliSphere" [4] is a project that shares a common theme with the aforementioned systems, i.e., accessing data across multiple heterogeneous data sources. In the *IntelliSphere* architecture (See Figure 1), Teradata is the master engine and the communication point with the end-users. The other underlying sources (called *remote systems*) are heterogeneous, but they are all assumed to have SQL-like interface (even if the internal execution is not SQL). This covers a wide spectrum of systems such as Hive [25, 26], SparkSQL [7], Presto [27], Impala [22], and other RDBMSs [1, 2, 23]. Therefore, *IntelliSphere*'s query language is SQL, and Teradata is responsible for building a SQL query plan and deciding where each SQL operator, e.g., join or aggregation, will execute on one of the *IntelliSphere*'s systems (either Teradata or a remote system).

In this paper, we focus on one fundamental aspect of *IntelliSphere*, which is the *cost estimation of a given SQL operator over remote systems*. The *"cost"* in our context is basically the elapsed execution time of a SQL operator on the remote system. The problem is fundamental because all modern optimizers (including Teradata's optimizer) are cost-based, and without accurate cost estimation for each query operator, the generated plans can be way off the optimal plan. Evidently, in the popular pay-as-you-go cloud model, bad execution plans can have unacceptable time and monetary overheads. Despite the importance of the problem, it is briefly touched by existing systems because as highlighted above, and will be elaborated on further in Section 6, each of the existing systems focuses on other aspects of the big problem.

Accurately estimating a remote operator cost is a challenging problem because: 1) Some remote systems are *openbox* where experts can inject a lot of details about them into *IntelliSphere* while

[*] Dr. Eltabakh is a faculty in the Computer Science Department at Worcester Polytechnic Institute, MA, USA (meltabakh@wpi.edu).

Series ISSN: 2367-2005

10.5441/002/edbt.2020.64

other systems are *blackbox* with very little knowledge on how they execute. 2) Two remote systems, e.g., Hive and Impala, may offer entirely different set of algorithms to physically implement a given operator, e.g., joining two tables, and thus whatever learned from one system does not necessarily apply to another system. 3) Within a single remote system, it is not trivial for *IntelliSphere* to predict which physical algorithm, possibly from several candidates, will be used for a given operation. And 4) Putting the simplistic assumption that all remote systems are blackboxes and the only way to learn their behavior is by submitting many queries as in [13] is not a practical scalable solution. This is because, as we will show in the paper, such approach of learning is very expensive and should be used as a last resort instead of the default and only solution.

In this paper, we propose a comprehensive remote-system cost estimation module for SQL operators that addresses the challenges highlighted above. To be specific, the costing metric that we try to measure in this project is the *elapsed execution time within the remote system*. This time encapsulates and reflects other detailed costs, e.g., query compilation, scheduling, I/O and CPU costs within the remote system. We assume that the network costs, e.g., establishing a connection and data transfer back and forth, are learned through some other mechanisms, which are outside the scope of this paper. Ultimately, the Teradata optimizer will combine multiple costs together to come up with a final cost for the SQL operator, and based on that it decides where to execute the operator. The techniques presented in this paper focus only on estimating the elapsed execution time, which is a major factor in the cost equation.

We propose three costing approaches, namely *logical-operator*, *sub-operator*, and *hybrid* approaches, which are suitable for *blackbox*, *openbox*, and a mix of black and open box systems, respectively. The cost estimation module leverages analytical models as well as deep learning models within the different approaches. We show that although the deep learning models are good in capturing non-linear cost estimation, they fall short in providing accurate estimations for un-seen (un-trained) ranges. To overcome this limitation, we propose *online remedy* and *offline tuning* phases to enhance the estimation quality.

The key contributions of the paper are summarized as follows:

• Proposing a comprehensive remote-system cost estimation module for SQL operators, that encompasses three approaches, namely *logical-operator* (*logical-op*), *sub-operator* (*sub-op*), and *hybrid* approaches. Each of the *logical-op* and *sub-op* approaches has pros, cons, and applicability cases. The *hybrid* approach combines their advantages.

• Leveraging both analytical cost models and deep learning models within the different costing approaches. The deep learning models are empowered with online remedy and offline tuning phases to ensure high quality estimations even for un-trained ranges.

• The proposed cost estimation module is modular, and due to its applicability to openbox and blackbox systems, it can be easily adopted by and integrated within other systems such as polystore systems.

• Evaluating the proposed cost estimation module empirically in the context of Teradata and Hive as a proof of concept. Extensions to other systems such as SparkSQL, Presto, and Impala follow the same methodology. The results show the effectiveness of the proposed module compared to the state-of-art approaches.

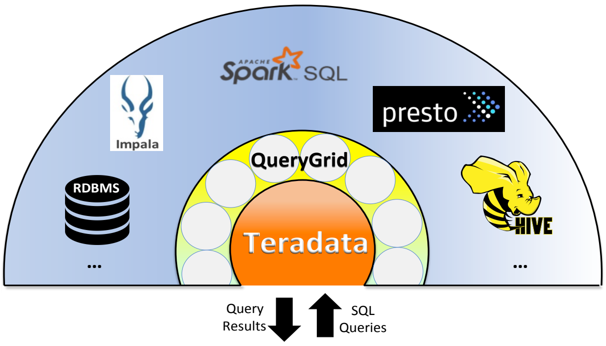


Figure 1: Teradata IntelliSphere Architecture.

The rest of the paper is organized as follows. In Section 2, we present the architecture of the *IntelliSphere* system and introduce the problem definition. In Sections 3, 4, and 5, we describe the details of the three costing approaches: *logical-op*, *sub-op*, and *hybrid*, respectively. In Section 6, we overview the related work, and in Section 7, we present the experimental evaluation of the system. Finally, Section 8 contains the conclusion remarks.

2 TERADATA INTELLISPHERE

In this section, we overview a simplified architecture of the *Teradata IntelliSphere* system [4] and the basic workflow components related to this paper[1]. *Teradata IntelliSphere* is designed to be a cost-effective and scalable analytical ecosystem that offers numerous software solutions to ingest, access, and manage big data across multiple heterogeneous data sources. For the purpose of this paper, we focus on the following basic components of the architecture (See Figure 1):

Teradata: The master engine in the entire architecture is the Teradata Database [2]. It also represents the communication point with the end-users. It receives a user's query in the form of a SQL query, generates a cost-based efficient query plan where each SQL operator is scheduled for execution on one of the *IntelliSphere*'s systems, combines the results, and passes the final answer back to the user.

Remote Systems: The underlying heterogeneous data sources are referred to as *remote systems*. They are all assumed to have SQL-like interface where they can receive a SQL operation such as a *join*, *aggregation*, *filter*, and *projection*, perform the computations of that operation and return the results back to Teradata. It is possible that the internal execution of a remote system is different from the relational DBMS model, e.g, Hive's internal execution is map-reduce. And it is also possible that a remote system may not support some of the SQL operations, e.g., a remote system may not have the capability to perform a join operation.

Remote System Profile: Each remote system registers in the *IntelliSphere* architecture through a profile. This profile describes the remote system setup, e.g., a cluster configuration, and the capabilities of the remote system, e.g., what operations it can or cannot support. The profile is constructed during the registration step, and can be modified afterwards as needed. We will use the profile extensively to store all metadata information related to the cost estimation module as will be described in the following sections.

QueryGrid: It is the communication layer that facilitates the transfer of data across the involved systems [3]. Several QueryGrid

[1] *Teradata IntelliSphere* is a more comprehensive architecture with features and functionalities beyond what is presented in this paper. We only highlight the aspects related to our paper.

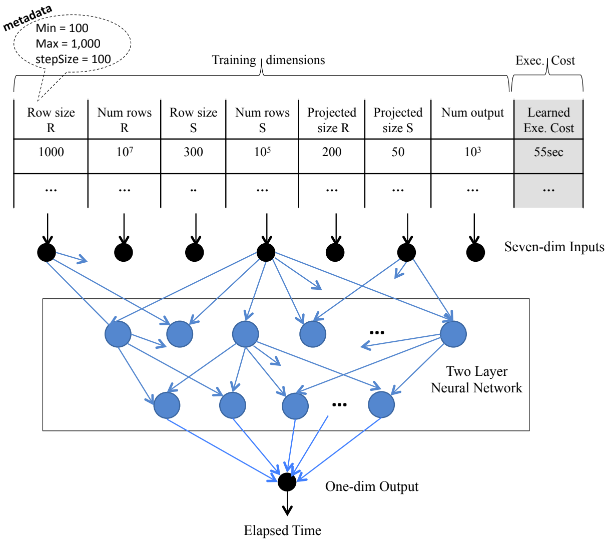
connectors are built to enable queries to access tables stored in remote systems. The differentiating factor between Teradata's QueryGrid technology and other connectors is that it works in conjunction with the query processing engines to optimize the overall execution. For example, simple predicates—in a well-defined language—can be passed to QueryGrid for execution on-the-fly while the data is being transferred from one system to another. This capability can save unnecessary scanning of a local data, writing back to the file system after evaluating the predicate, and then passing the results to the QueryGrid for transfer.

Data Storage, Statistics, and Transfer: A given dataset consists of a set of tables $\{T_1, T_2, ..., T_k\}$, where each table is stored on one of the *IntelliSphere*'s systems (Teradata or a remote system). Any remote table is registered inside Teradata as a *foreign table*—and thus Teradata knows its schema and location. As a result, a single SQL query can seamlessly reference multiple foreign tables across several remote systems. We assume that Teradata can collect basic statistics on remote tables, e.g., the number of rows, average row size, the number of distinct values in each column, etc. Such information is either already available on the remote system or Teradata can estimate them by submitting some queries over the data. Regarding the transfer of data, the data cannot be transferred directly between two remote systems, instead it can be only transferred between a remote system and Teradata.

Query Plans: As in standard RDBMSs, Teradata generates many equivalent SQL query plans during the optimization phase, and part of that is deciding on where each operator will execute—which clearly implies different costs depending on the host system. To limit the search space, *IntelliSphere* considers scheduling an operator only on a remote system that owns the input data (or part of it) or the Teradata system. For example, assume joining two relations R and S, where R is stored in Hive and S is stored in Presto. Then, there are three possibilities for placing the join operator, either on Hive (and S will be passed to Teradata and then to Hive), on Presto (and R will be passed to Teradata and then to Presto), or on Teradata (and both R and S will be passed to Teradata). The results computed on a remote system do not have to be immediately transferred to Teradata, instead they may remain on that remote system for further computations, and then at some point in the query, the results will be transferred to Teradata.

Problem at Hand and Design Assumptions: Given the setup described above, *IntelliSphere* leverages the full-fledged capabilities of Teradata's mature optimizer in generating efficient cost-based query plans. The only missing piece is estimating an operator's cost were it to be executed on a remote system. As highlighted in Section 1, this cost involves several factors, we only focus on estimating the wall-clock elapsed execution time within the remote system. Therefore, while estimating the execution cost, we assume that the needed data is already on the remote system—and thus the network communication and data transfer costs are out of the picture [2].

Supervised ecosystem: The learning and model building step for a given remote system is performed only once when the remote system is added to the *IntelliSphere* ecosystem. Therefore, the learned models are for specific cluster configuration, access methods, physical data layout, etc. The *IntelliSphere* ecosystem is supervised in the sense that changes to a remote system, e.g., adding or removing nodes, creating or dropping indexes, re-partitioning the data, etc., are known to Teradata. Such changes, would require re-doing the learning phase from scratch.



Row size R	Num rows R	Row size S	Num rows S	Projected size R	Projected size S	Num output	Learned Exe. Cost
1000	10^7	300	10^5	200	50	10^3	55sec
...	...	..	...	...	...	...	...

Figure 2: Logical-Operator Costing for Join Operator.

Stable workload: Another underlying assumption is to have a roughly consistent workload on the remote system. That is, the workload during the training and model construction phase should roughly remain the same while executing users' queries later. In our experiments, we assume the remote system is dedicated to the submitted queries and no other workloads are running. Supervised ecosystem and stable workloads are the same assumptions used in almost all other related work [15, 21, 30], otherwise it is impossible to predict the remote system behavior.

Integration in bigger query plans: In Teradata, the cost of a SQL query operator includes several low-level factors such as the I/O costs, e.g., index scans, disk page accesses for data, and CPU costs, e.g., hash table creation, hash table lookup, records merge or sort, etc. Ultimately, these costs are translated to an estimated execution time cost per operator. As such, the estimated execution time for the remote operators fit directly in bigger plans.

3 LOGICAL-OPERATOR COSTING

One approach for estimating a remote operator cost is the *Logical-Operator Costing* (*logical-op costing* for short). In this approach, the training and learning phase is performed at the logical operator level, e.g., join and aggregation operators. This is the approach used in other systems, e.g., BigDAWG [13]. The main idea of the logical-op costing is to build a relatively large set of training queries, execute them on the remote system, and build a model for the target operator. The key advantage is that it requires no knowledge about the internal execution of the remote system, e.g., it does not need to know which physical join algorithm is used to perform the join. In other words, the remote system is treated as a blackbox. However, the main drawback is that to build a reasonably accurate model for a given operator, it would require a large number of queries to cover a wide range of possible configurations. This would certainly require a prolonged training phase and potentially consume valuable resources. In the following, we describe in detail the phases involved in this costing approach.

Building a training dataset: In general, the more complex the logical operator and the more variations in physically implementing it, the more training queries are needed to build its corresponding model. We created logical-op training models for

[2] Teradata can estimate the amount of data that need to be sent to the remote system as well as the output size that will be sent back to Teradata. Based on these estimates, other costs such as the the network cost and data transfer are estimated.

the join and aggregation operators. For the join operator, the training model has seven dimensions, which include the *row size* and the *number of rows* in each of the two tables, the *sum of the projected attribute sizes* from the each table, and the *number of output rows* (See Figure 2). For the aggregation operator, the model has four dimensions, which include the *number of input rows*, *input row size*, *number of output rows*, and *output row size*.

Coming up with appropriate training dimensions is crucial and requires some level of expertise. On one hand, we need to minimize the number of dimensions because the number of queries grows exponentially when adding more dimensions. On the other hand, we need to capture enough parameters in order to model the targeted operator accurately. Based on our team's experience with the Teradata query optimizer, we selected the highlighted dimensions as the representative dimensions for the join and aggregation operators.

The next step is to assign for each dimension a domain reflecting the possible training values that this dimension may take. In some applications, there can be samples of existing data or workloads to help selecting the appropriate domain for each dimension. Otherwise, we start with reasonable assignments, and then a continuous learning phase will help to gradually expand the domains as the system observes and executes more queries (as will be explained later).

Assume dimension i has a domain d_i of size $|d_i|$, then the total number of configurations in the training set for one operator is computed as $\prod_{i=1}^{k} |d_i|$, where k is the number of dimensions. For example, as illustrated in Figure 2, each row represents one configuration, which maps to a single query over the remote system. After executing the queries, each configuration will be labeled with the observed execution cost. This step of executing the queries over the remote system can be very expensive, e.g., it may take days if the number of queries is large.

Building a costing model: The next step in the logical-op costing is to build a model from the observed costs. For that purpose, regression or neural network models can be used. We experimented with both, and we found out that linear regression models introduce more errors as will be demonstrated in the experiment section (around three times larger w.r.t the root-mean-square error RMSE). This is primarily because the number of data points can be large, e.g., in thousands, the number of input dimensions can be also large, and the relationship between the inputs and outputs might not be linear—especially for complex operators like join and aggregation. Simple light-weight neural networks tend to be more accurate under such complex modeling. For that reason, we opt for the neural network model in the rest of the paper.

There is no rule of thumb for deciding on the optimal neural network structure. Typically, two or three hidden layers are enough for not highly-complex problems [18]. Therefore, we fix the number of layers to two for both the join and aggregation operators. And then we use a cross-validation technique to determine the number of nodes (neurons) in each layer [16]. More specifically, we vary the number of nodes in the 1^{st} layer between the number of inputs (7 for join, and 4 for aggregation) and the double of that number, and vary the number of nodes in the 2^{nd} layer between three and half the number of the 1^{st} layer's nodes. Then, for each topology, we use a cross validation test involving 70% of data as training and 30% as a test to measure the accuracy of the network. Finally, we select the topology that introduces the least root-mean-square error (RMSE). Figure 2 shows the neural network model over the seven-dimension inputs for the join operator.

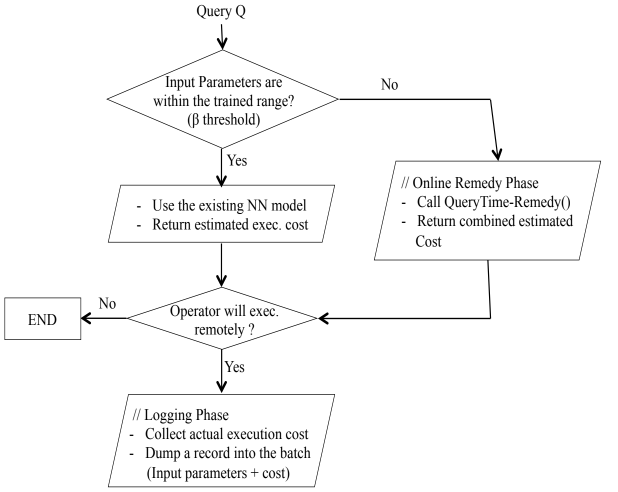

Query Q
Input Parameters are within the trained range? (β threshold)
No
Yes
- Use the existing NN model
- Return estimated exec. cost
// Online Remedy Phase
- Call QueryTime-Remedy()
- Return combined estimated Cost
END
No
Operator will exec. remotely ?
Yes
// Logging Phase
- Collect actual execution cost
- Dump a record into the batch (Input parameters + cost)

Figure 3: Logical-Operator Costing: cost-estimation flow chart at query time.

Usage and model expansion: In the typical scenarios, the constructed model is directly used at query time to estimate the cost of a remote operator. Given an operator that is candidate for execution on a remote system, e.g., a join operator where one of the input relations is on that remote system, the system calculates and/or estimates the input parameters for the operator's model. For example, the seven input parameters illustrated in Figure 2 need to be estimated for the join operator. These parameters are then fed to the neural network model to predict the output value, which represents the estimated cost (See the flowchart in Figure 3).

The estimation process is straightforward as long as all the input parameters fall within (or in the proximity of) the ranges on which the neural network is trained. However, in the cases where one or more parameters are way off the trained ranges, the model may not provide accurate estimation. This is because neural networks are good in capturing complex relationships but not good in extrapolating out-of-range values.

In real deployment systems like Teradata *IntelliSphere* these cases need to be adequately handled. Therefore, we propose a two-phase solution that consists of an *online query-time remedy phase* and an *offline batch tuning phase*. The online remedy phase provides an immediate best-guess estimation to the operator at hand to continue the query optimization and execution. Whereas, the offline batch tuning phase provides a mechanism for readjusting and tuning the neural network from the actual logged executions. Both phases are described in detail below.

Online Remedy Phase: The main idea of the online remedy phase is presented in Figures 3 and 4. Initially, the system maintains metadata information for each input dimension in the training set of a given operator. This metadata includes the covered range using *min* and *max* boundaries and a *stepSize*. For example, Figure 2 shows the metadata of the *Row size* dimension, which indicates the training covers the range from 100 to $1,000$ bytes and the step size is 100. Now, if the current query at hand involves a join where the estimated row size is $10,000$ bytes, the system will detect that this parameter is way off the trained range, and will not get the estimate by relying only on the neural

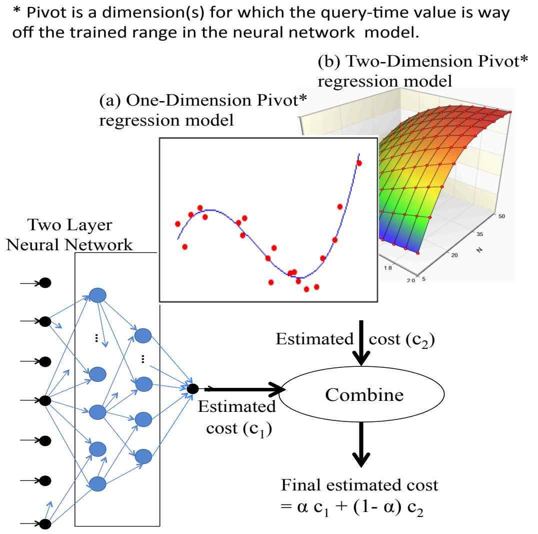

* Pivot is a dimension(s) for which the query-time value is way off the trained range in the neural network model.
(b) Two-Dimension Pivot* regression model
(a) One-Dimension Pivot* regression model
Two Layer Neural Network
Estimated cost (c_2)
Combine
Estimated cost (c_1)
Final estimated cost = α c_1 + (1- α) c_2

Figure 4: Online Remedy Phase: Combining Neural Network and On-the-Fly Regression Model for Cost Estimation. (a) Regression model for one-dim pivot, (b) Regression model for two-dim pivot.

network model, but instead it will trigger the execution of the `QueryTime-Remedy()` procedure (See the top diamond box in Figure 3). More specifically, if the value of a given dimension is outside the *[min, max]* range by more than $\beta * stepSize$, where $\beta > 1$ is a configuration parameter, then that dimension is considered way off the trained range. The procedure on the fly builds a regression model and combine it with the neural network model to come up with the final estimate as illustrated in Figure 4.

Lets illustrate the construction of the regression model using a simple scenario. Assume a join query Q that involves only one dimension, say *Row size of R*, where its estimated value is way off the trained range in the neural network. We refer to that dimension as the *Pivot* dimension. All other dimensions (refer to them as $D_{inRange}$) are within the trained range. The `QueryTime-Remedy()` procedure extracts a set of training records of size k, where k is a system parameter, having the following properties: (1) their values in the $D_{inRange}$ dimensions are matching (or very close) to the corresponding values in Q, and (2) their values in the *Pivot* dimension are the immediate successors and/or predecessors of the corresponding value in Q. This set should represent the closest possible training points to the query point. The pivot values in this set are then extracted and used to build a regression model. The algorithm can be extended to handle more than one pivot dimension as illustrated in Figure 4.

The `QueryTime-Remedy()` procedure uses the constructed regression model to extrapolate on the pivot dimension(s) and predict the cost. This cost is then combined with the estimated cost from the neural network model to come up with the final cost (See Figure 4). The reason we combine the two costs is that they capture different and complementary factors. The neural network captures the complex relationship between the input parameters and the output but cannot extrapolate. In contrast, the regression model can extrapolate but oblivious to the other dimensions. The costs are combined using a weighted factor α $(0 < \alpha < 1)$ as illustrated in Figure 4. Initially, α is set to 0.5, and as the system executes more queries, α gets automatically adjusted to narrow the gap between the estimated and actual execution times.

Offline Tuning Phase: Whenever *IntelliSphere* executes a remote operator on an external system (depending on the optimizer's decision), it captures the actual execution cost and pushes this information to a log (See the bottom diamond box in Figure 3). Periodically, this log is fed to the neural network model to tune its structure with the new observed data.

One interesting detail to highlight here is the mechanism for updating the metadata information of the training dataset at the end of the tuning phase. Recall that a metadata information is maintained for each dimension in a training set including the *min, max, stepSize* values. When a log gets executed to update the neural network model, the metadata gets also updated. More specifically, the *[min, max]* range gets expanded if the log has entries with out-of-range values. However, this expansion takes place only if a continuity in the training points is maintained. For example, referring to the metadata in Figure 2, if the log has some entries with out-of-range values or the 1^{st} dimension like $8,000$ and $10,000$ bytes, then the current range will remain intact because there are many missing points between that range and the new values, i.e., continuity will be broken. Instead, more information is added to the metadata to indicate that training dataset of $8,000$ and $10,000$ bytes

The implication of this expansion strategy is that when a new join query comes and it includes an out-of-range value for the 1^{st} dimension, say $6,000$ bytes, the system will still trigger the *online remedy phase* highlighted in Figure 4 to come up with the final estimated cost instead of relying only on the neural network model. The positive thing is that the prediction from both the neural network and the regression models are getting better because they take into account the previous log records even if the [*min, max*] range has not been modified.

4 SUB-OPERATOR COSTING

Another approach for remote operator costing is what we refer to as *sub-operator costing* (or *sub-op costing* for short). In this approach, the learning and training is performed at the granularity of small building block operators, e.g., scan, shuffle, sort, read, and write operations. And then, the higher-level query operators, e.g., join and aggregation, are expressed as formulas on top of the sub-ops. The main advantage is that learning the cost of each sub-op is relatively straightforward and fast because: (1) The number of dimensions in a training set for each sub-op is typically very small (only two or three), (2) As a result of the low-dimensionality, the number of needed training queries is very small—which saves training time and cost, and (3) The logic and behavior of each sub op is relatively simple and thus linear regression is typically enough to model most of these sub ops.

On the other hand, the main disadvantage of the sub-op approach is that it requires a great deal of knowledge about the remote system, which may not be available in some cases. For example, it requires identifying a set of the building block operators (the sub ops) that is sufficient to accurately model the query operators. It also requires understanding the different algorithms of the physical implementations for the different operators, e.g., a join operator can have four or five different physical algorithms such as broadcast join, re-distribution hash join, etc., and defining a formula to express each algorithm in terms of the sub ops. Evidently, if such level of knowledge is not already available, then it takes a long time to collect with these details.

Identifying sub operators and costing formulas: The first step in this approach is to identify the key sub operators of the

Basic (Mandatory)	Read (DFS) [1]	r_D	Reading a record from dist. file system
	Write (DFS) [2]	w_D	Writing a record to dist. file system
	Read (Local)	r_L	Reading a record from local file system
	Write (Local) [3]	w_L	Writing a record to local file system
	Shuffle	f	Shuffle a record between machine
	Broadcast [4]	b	Broadcast a record to all machines
	Sort	o	Main memory sort cost per record
	Scan	c	Main memory scan cost per record
Specific (Optional)	HashTable Build [5]	h_I	Inserting a record into hash table
	HashTable Probe	h_P	Probing a hash table
	Rec Merge	m	Merging two records

[1] Query that reads from HDFS and does not produce any output.

[2] Query that reads from HDFS and writes back to HDFS. Subtract r_D from the measured values

[3] Query that reads from HDFS and writes content to local file. And then subtract r_D from the measured values

[4] Query that reads from HDFS, produces no output, and broadcasts a file (distributed cache) to all nodes (without reading it). Subtract r_D from the measured values

[5] Query that reads from HDFS, builds a hash table for each HDFS block, and does not produce any output. Subtract r_D from the measured values

Figure 5: List of Common Sub Operators in Remote Systems. Additional sub ops can be defined specifically for certain remote systems.

remote system, which may differ from one system to another. However, in the majority of the modern distributed systems, which have shared-nothing architecture in common, these sub operators typically include: *reading from disk, writing to disk, shuffling across machines, in-memory sorting,* and *scanning a memory block*. Other more specific sub operators include *insertion into a hash table, probing a hash table,* and *merging two records*.

In Figure 5, we highlight a list of the key and common sub operators and categorize them into two categories, namely *Basic* and *Specific*. The sub operators under the *Basic* category are kind of mandatory to learn, otherwise it would not make sense for the corresponding remote system to be costed using this approach. The other sub operators are good to have, but missing them is not a hinder to this approach because either they are specific to few query operators, they are not dominating factors in the cost formulas in which they participate, or *IntelliSphere* can provide rough default values for them. We will provide more details and examples in this section for these sub operators.

It is worth to highlight that Teradata costing mechanism is based on the *sub-op costing* approach. It is highly reliable, efficient to use for estimation, and easy to calibrate and extrapolate whenever needed. Given that all engine details are known, Teradata optimizer maintains a long and detailed list of sub operators. In contrast, for remote systems, it is more practical to assume limited knowledge about them. That is why we try to capture a minimal, but sufficient, set of sub ops as highlighted in Figure 5.

After defining the sub operators, each query operator for which a costing model need to be built, e.g., join and aggregation, need to be expressed as a composition of the sub operators. Since each of these query operators can have multiple physical implementations carrying significantly different costs, it is important for a technical expert to know the list of physical algorithms that are supported by the remote system for a given query operator. For example, Hive supports five types of join algorithms, which are: *Shuffle Join*, *Broadcast Join*, *Bucket Map Join*, *Sort Merge Bucket Join*, and *Skew Join* [19]. Similarly, Spark supports five join algorithms, which are: *Broadcast Hash Join*, *Shuffle Hash Join*, *SortMerge Join*, *Broadcast NestedLoop Join*, and *Cartesian Product Join*. Each of these algorithms need to be expressed in terms of the defined sub operators.

In Figure 6, we provide a detailed example using the *Broadcast Join* algorithm between two relations *R* and *S*, where *S* is assumed to be the small relation. The top part of the figure shows the algorithm workflow while the bottom part shows the corresponding cost formula. The algorithm starts by reading the small relation *S* from the distributed file system, e.g., HDFS, and broadcasting it to all workers, and it gets stored locally on each machine. Then each task—in Hadoop terminology, it is called a *map task*—executes the loop illustrated in Figure 6. Basically, each task reads relation *S* and builds a main memory hash table, and then reads one block from the big relation *R* and for each record in that block, it probes *S*'s hash table for possible joins. The read block from *R* is assumed to be on the local disk because most distributed systems try to achieve data locality by putting the computational task on the machine storing the data. Previous studies have shown that although data locality is a best effort mechanism, it is achieved more than 90% of times. The last step in the workflow is for the task to write its output back to the distributed file system[3].

The costing formula in Figure 6 has almost one-to-one mapping to the steps in the workflow. We use the notation | | to indicate the cardinality (number of records) of an input. The term `NumTaskWaves` represents the number of cascaded tasks executed on a single machine. It is computed as total number of tasks in the join job divided by the total number of parallelism in the system, i.e., the total number of cores. Notice that the values for factors such as `NumTaskWaves`, `|Block(R)|`, and `|TaskOutput|` are calculated and/or estimated by another module in the *IntelliSphere* system different from the costing module and that module is outside the scope of this paper.

Building a training dataset: The upfront effort put in specifying the sub ops and the cost formulas will pay of by simplifying the subsequent phases of the sub-op costing approach. For the training dataset, what is needed is to build a set of queries for each of the sub ops to learn its cost in the remote system. Since each sub op is *primitive*, the number of dimensions in its training dataset is very small. In fact, we found it enough for almost all sub operators under the *Basic* category (Refer to Figure 5) to have only two dimensions in the training dataset, which are the *number of records* and *record size*. The only exception is the *Broadcast* sub operator, which requires a third dimension, which is the *number of machines*.

Since the number of dimensions is small, and additionally the number of values assigned to each dimension is also small (because the sub op models are easy to extrapolate as we will discuss later), the number of records in the training set becomes very small. In fact, it is between one to two orders of magnitudes smaller than that of the logical-operator approach, which introduces a significant reduction in the training time and cost over a remote system.

For measuring the cost (execution time) for each sub-op, we avoided instrumenting and injecting special code inside the remote system since such instrumentation may not be feasible for some remote systems. Instead, we submitted primitive queries that execute specific type of operations, and from that we extracted the values of the individual sub-ops. In Figure 5, we show examples

[3] Cost formulas for other join algorithms can be derived in the same manner. We omitted them from the paper due to space limitations.

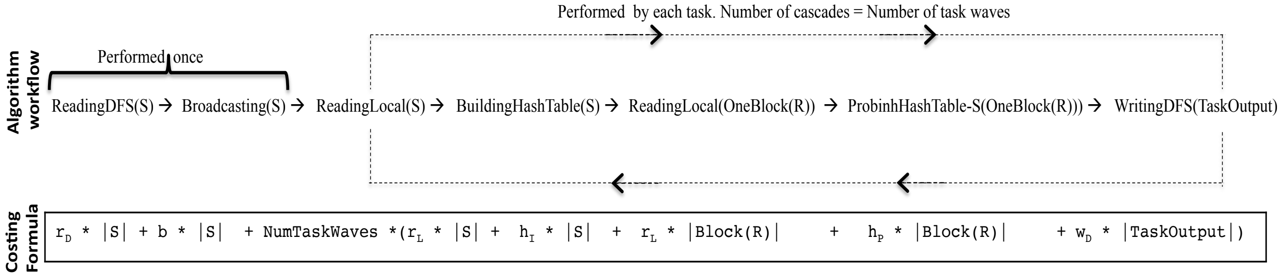
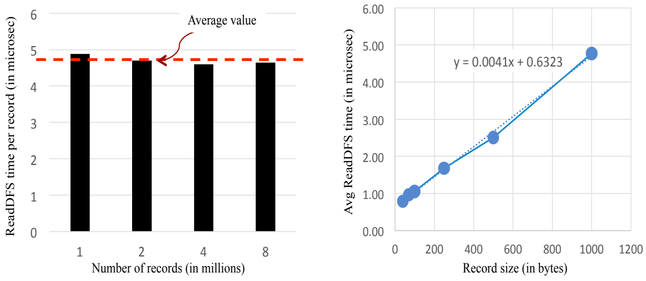

Performed by each task. Number of cascades = Number of task waves
Performed once
Algorithm workflow
ReadingDFS(S) → Broadcasting(S) → ReadingLocal(S) → BuildingHashTable(S) → ReadingLocal(OneBlock(R)) → ProbinhHashTable-S(OneBlock(R))) → WritingDFS(TaskOutput)
Costing Formula
r_D * |S| + b * |S| + NumTaskWaves *(r_L * |S| + h_I * |S| + r_L * |Block(R)| + h_P * |Block(R)| + w_D * |TaskOutput|)

Figure 6: Broadcast Join (R, S) in Hive & Spark (Broadcast Hash Join). The Algorithm workflow and the costing formula in terms of the sub ops. Relation S is the small relation to be broadcasted.



(a) ReadDFS cost for a 1,000 byte record

(b) ReadDFS linear regression model

Figure 7: Sub-Op Costing Model for ReadDFS Operator.

of these queries and how they can be used to measure specific sub-ops.

Building a costing model: In this step, a cost model is built for each sub operator. For simplicity, we will focus our discussion on the majority of the sub ops, which involve two dimensions in the training set, i.e., *number of records* and *record size*. It is possible to consider these two dimensions as separate (orthogonal) dimensions while building the model. However, we experimentally observed that the model can be further simplified because for a given record size, say *s*, the measurements across the other dimension (the records' number) are very similar to each other. Therefore, it is practical to group the measurements by the record size, and compute the average across the varying number of records. In Figure 7(a), we illustrate this observation. The experiment is measuring the *ReadDFS* (Reading from distributed file system) cost for a record size of 1,000 bytes under varying number of records. The dotted line shows the average value. Similar findings are observed for other record sizes and other sub operators.

Based on this observation, a simple linear regression costing model can be built as depicted in Figure 7(b) for the *ReadDFS* operator. As can be noticed a big advantage of the sub-op costing approach is that most sub-ops have simple and tight linear regression models that can be easily learned from small training dataset (more results will be presented in Section 7). Moreover, these models are easy to extrapolate for un-seen values, which is not the case for the more complex neural network models presented in Section 3.

Usage: At query time, lets say a join query between *R* and *S*, the first thing to be done by the *IntelliSphere* cardinality estimation module is to provide the required cardinalities and statistics, e.g., the cardinality of each relation, the number of distinct values in the join keys, the average number of records per key, etc. Then, if the operator at hand has only one physical implementation in the remote system, then *IntelliSphere* uses the corresponding cost formula to estimate the cost. Otherwise, if there are multiple possibilities, which is the case for the join operator (Refer to Figures 6), then *IntelliSphere* needs to predict which algorithm the remote system will use.

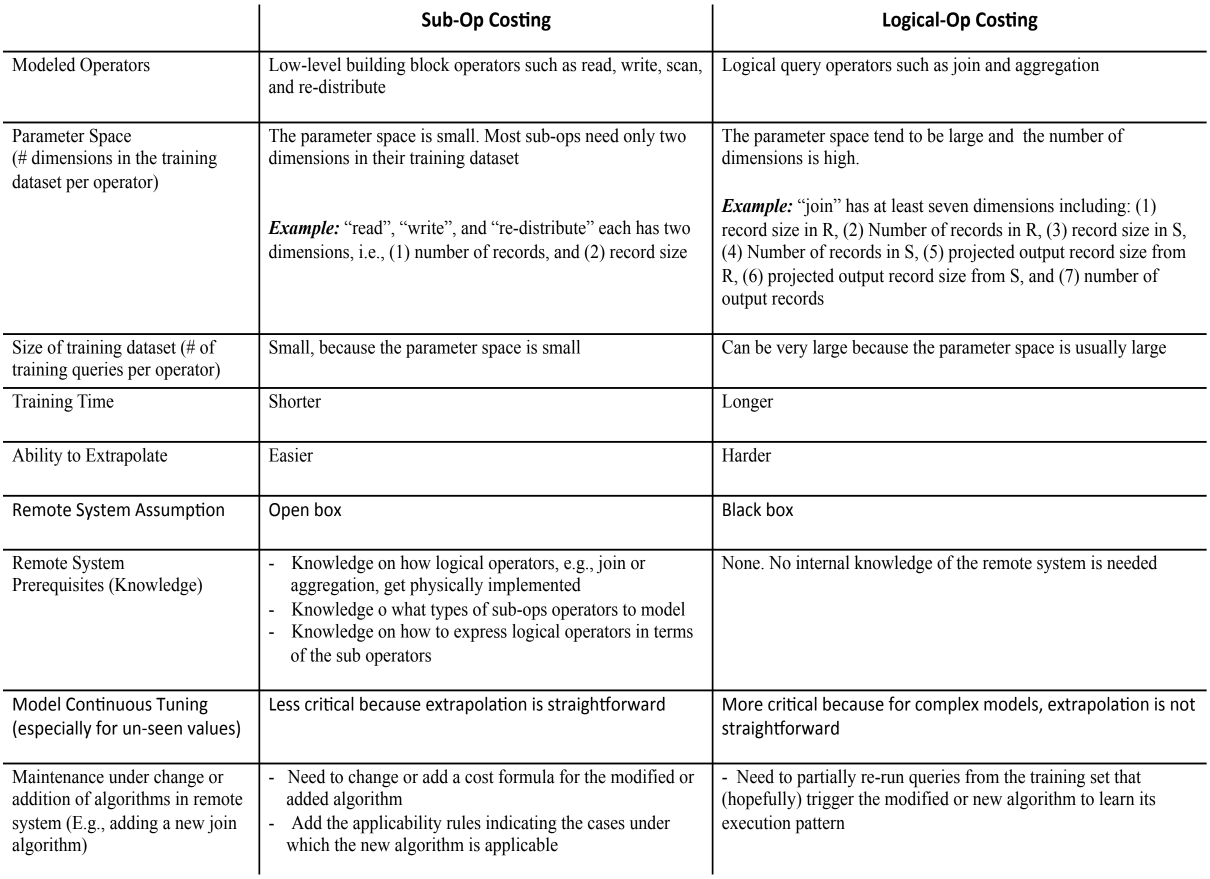
Predicting the remote system choice is tricky, especially for complex systems such as other relational databases, e.g., DB2, SQL Server, or Oracle. Yet, it is more straightforward for systems like Hive and Spark. Lets take the join operator, which has the most algorithmic variations, as an example. Although it has five algorithms in Hive and Spark, most of the choices can be easily eliminated based on some observations. For example, if the relation in Teradata side, say *S*, which will be sent to the remote system is not partitioned by the join key—which *IntelliSphere* should know—then the choices of *Bucket Map Join* and *Sort Merge Bucket Join* in the case of Hive can be eliminated. Even if *S* is partitioned on the join key, but there is no way to tell the remote system such property after the data transfer, then still the two choices above can be safely eliminated. If the join is not Cartesian product, then the choices of *Broadcast NestedLoop Join* and *Cartesian Product Join* in the case of Spark can be eliminated. If both join relations are quite large, then the choices of *Broadcast Join* either in Hive or Spark can be eliminated.

These observations, or what we refer to them as *"Applicability Rules"*, are defined by the technical experts while defining the cost formula for each possible algorithm. *IntelliSphere* uses them at query time to eliminate inapplicable choices based on the cardinalities and statistics at hand. Finally, if there are still multiple possible choices, then the system can either take the highest cost (assuming the worst case scenario), the average cost, or the *"in-house comparable"* cost. The *in-house comparable* cost is applicable when the remote system is another relational database system. In this case, *IntelliSphere* assumes that the remote system will pick the algorithm that Teradata would have picked were the data in-house.

5 HYBRID-OPERATOR COSTING

As highlighted in Sections 3 and 4, each of the sub-op and logical-op approaches has pros and cons. Such tradeoff between the two approaches and the diverse remote systems available nowadays in the Big Data ecosystem call for a hybrid approach that can combine the advantages of both worlds.

In Figure 8, we provide a summary comparison between the two approaches. In general, the sub-op costing model can be significantly superior w.r.t the training cost, training time, and the ease of extrapolation given that a detailed knowledge on the remote systems is already available. Otherwise, the logical-op model would be the favorite.



	Sub-Op Costing	Logical-Op Costing
Modeled Operators	Low-level building block operators such as read, write, scan, and re-distribute	Logical query operators such as join and aggregation
Parameter Space (# dimensions in the training dataset per operator)	The parameter space is small. Most sub-ops need only two dimensions in their training dataset ***Example:*** "read", "write", and "re-distribute" each has two dimensions, i.e., (1) number of records, and (2) record size	The parameter space tend to be large and the number of dimensions is high. ***Example:*** "join" has at least seven dimensions including: (1) record size in R, (2) Number of records in R, (3) record size in S, (4) Number of records in S, (5) projected output record size from R, (6) projected output record size from S, and (7) number of output records
Size of training dataset (# of training queries per operator)	Small, because the parameter space is small	Can be very large because the parameter space is usually large
Training Time	Shorter	Longer
Ability to Extrapolate	Easier	Harder
Remote System Assumption	Open box	Black box
Remote System Prerequisites (Knowledge)	- Knowledge on how logical operators, e.g., join or aggregation, get physically implemented - Knowledge o what types of sub-ops operators to model - Knowledge on how to express logical operators in terms of the sub operators	None. No internal knowledge of the remote system is needed
Model Continuous Tuning (especially for un-seen values)	Less critical because extrapolation is straightforward	More critical because for complex models, extrapolation is not straightforward
Maintenance under change or addition of algorithms in remote system (E.g., adding a new join algorithm)	- Need to change or add a cost formula for the modified or added algorithm - Add the applicability rules indicating the cases under which the new algorithm is applicable	- Need to partially re-run queries from the training set that (hopefully) trigger the modified or new algorithm to learn its execution pattern

Figure 8: Comparison between Sub-Op and Logical-Op Costing Approaches.

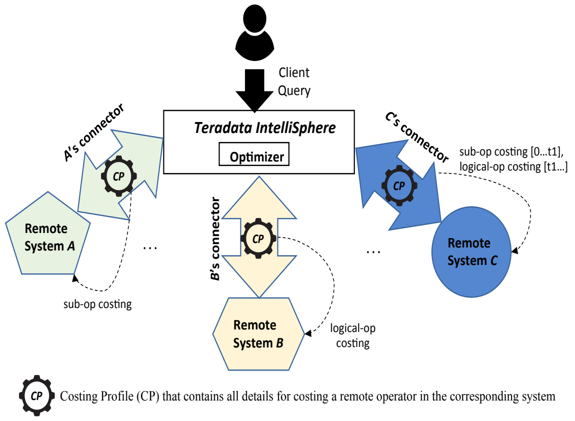


Figure 9: Overview on the Hybrid Costing Model.

The main idea of the hybrid approach is depicted in Figure 9. Basically, the Teradata *IntelliSphere* architecture will connect and communicate with different remote systems using one of the two costing approaches. The choice depends on several factors including whether or not there is enough knowledge about the remote system, and whether or not the resources allow for a prolonged training phases—which can be days in the case of logical-ops .

For example, referring to Figure 9, remote system A can be a well-known openbox system, e.g., Hive or Spark, and in this case the sub-op costing can be the model of choice. In contrast, remote system B is a blackbox and its workload and resources allow for a prolonged training phase, in this case the logical-op costing model is a good choice. On the other hand, remote system C has little knowledge known about it and its workload and resources do not allow for a dedicated several-days training phase (for logical-op training). In this case, an approximate sub-op costing can be applied to C—even if not highly accurate—until the more extensive training for the logical-op costing is performed, which may span weeks, and then C switches from the sub-op costing to the logical-op costing. The *IntelliSphere* architecture provides such flexibility.

As highlighted in Figure 9, each remote system has a *costing profile (CP)* containing all needed details based on its costing model. For example, for the sub-op costing, it includes a list of the sub-ops, a list of the physical algorithms for each logical operator, the costing formula of each algorithm, and the applicability rules for each algorithm. For the logical-op costing, it includes the neural network model for each operator, the metadata information of the training dataset, plus other information. Updating the costing profile information instantaneously reflects on the remote table costing. Although not currently supported in *IntelliSphere*, the hybrid approach is also applicable within a single system. That is, some operators, e.g., selection and aggregation, can be trained using the logical-op approach, while other higher-dimensional operators such as joins can be trained using the sub-op approach. The CP profile is flexible enough to store different costing models for different operators. We plan to explore this extension in the near future.

6 RELATED WORK

Accessing and querying datasets that span multiple heterogeneous data sources is a complex problem, and several systems and architectures have been proposed to address certain aspects of the problem. In this section, we overview these related systems and emphasize the differences to the *IntelliSphere* system.

Federated Systems: Federated systems provide a virtual layer of a unified access and management over a collection of data sources [8, 11, 24]. The federation can be over a collection of homogeneous relational databases, e.g., distributed DBMSs (Category I), and most of the research in this category focuses on distributed transaction processing, replica management, recovery control, and concurrency control [14, 20, 24].

Systems such as [30] belong to Category I, and they address the cost model issue across multi relational databases by dividing the workload into multiple query classes, then sample a subset of queries from each class and submit them to the remote database(s). The objective is to learn the corresponding unknown coefficients of the cost equation using statistical regression models. This approach is similar to our proposed logical-op learning, however in their work they did not consider the sub-op costing, which some times has clear advantage of the logical-op costing especially when dealing with heterogeneous systems.

The federation can also be over a collection of heterogeneous data sources (Category II), and the focus of this category is on building unified data and representation models, query translation, mediation design, data extraction, schema matching and coalescing, and conflict and resolution management [10, 12, 28, 31]. *IntelliSphere* is fundamentally different from these systems because IntelliSphere's focus is on efficient query plan generation and remote operator cost estimation.

Some work under Category II such as that proposed in [21] addresses the costing in such heterogeneous data sources. However, their assumed sources are not limited to SQL-like operators, e.g., the sources can be web search engines, image processing systems, CAD systems, etc. In this setting, the authors proposed wrappers around each source that acts as a *mini-optimizer* and feeds a global optimizer with the estimated costs for a given operation. The developers of the remote systems need to code these wrappers and augment in them optimizer-like logic to derive the cost of the different possible operations on these remote systems. IntelliSphere is fundamentally different from that work because our focus is only on the costing of SQL operators, e.g., selection, projection, join, aggregation, etc. For that, there are no strong justifications for the complexity of adding a wrapper's layer and the non-trivial task of coding a mini-optimizer for each remote system.

Polystore Systems: The key characteristic of the polystore systems, e.g., the BigDAWG system [13], is that they provide transparent access across multiple engines of different data models, e.g., relational, graph, NoSQL, array, and steaming engines. In BigDAWG, the underlying sources are grouped into islands by their data model type, and then each source has a *"shim"* which acts as the source's communication wrapper. BigDAWG addresses issues including location transparency, casting among the different data models, unified query language, and query planning and optimization across the islands.

The *IntelliSphere* system is distinct from the polystore systems in the following: (1) *IntelliSphere* is not a polystore system because it assumes a common relational-like data model for all underlying data sources with a SQL-like interface. Therefore, *IntelliSphere* does not focus on issues such as casting among the different data models and building a unified query language. (2) Although cost estimation is a fundamental issue in BigDAWG, it is briefly touched and the system is currently using primitive approaches as a first step [13]. In contrast, *IntelliSphere* introduces a comprehensive cost estimation module for efficient query plan generation across the underlying systems. The innovations presented in this project can be certainly leveraged by other systems such as the BigDAWG system.

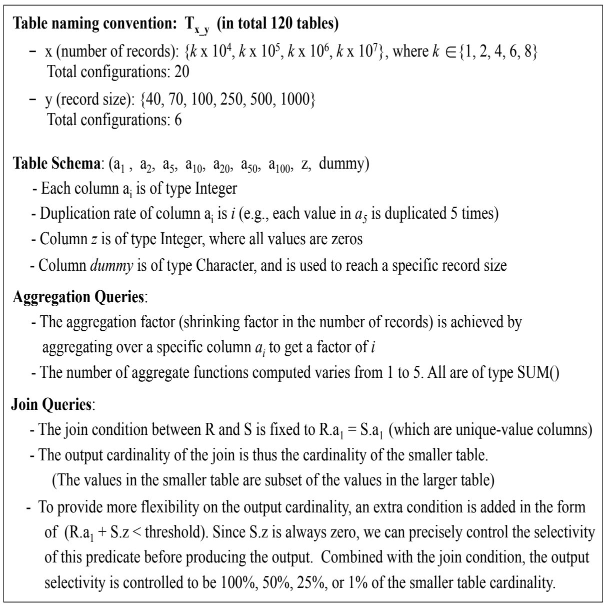


Table naming convention: T_{x_y} (in total 120 tables)

- x (number of records): {k x 10^4, k x 10^5, k x 10^6, k x 10^7}, where $k \in \{1, 2, 4, 6, 8\}$
  Total configurations: 20
- y (record size): {40, 70, 100, 250, 500, 1000}
  Total configurations: 6

Table Schema: (a_1 , a_2, a_5, a_{10}, a_{20}, a_{50}, a_{100}, z, dummy)

- Each column a_i is of type Integer
- Duplication rate of column a_i is *i* (e.g., each value in a_5 is duplicated 5 times)
- Column *z* is of type Integer, where all values are zeros
- Column *dummy* is of type Character, and is used to reach a specific record size

Aggregation Queries:

- The aggregation factor (shrinking factor in the number of records) is achieved by aggregating over a specific column a_i to get a factor of *i*
- The number of aggregate functions computed varies from 1 to 5. All are of type SUM()

Join Queries:

- The join condition between R and S is fixed to $R.a_1 = S.a_1$ (which are unique-value columns)
- The output cardinality of the join is thus the cardinality of the smaller table.
  (The values in the smaller table are subset of the values in the larger table)
- To provide more flexibility on the output cardinality, an extra condition is added in the form of ($R.a_1 + S.z <$ threshold). Since S.z is always zero, we can precisely control the selectivity of this predicate before producing the output. Combined with the join condition, the output selectivity is controlled to be 100%, 50%, 25%, or 1% of the smaller table cardinality.

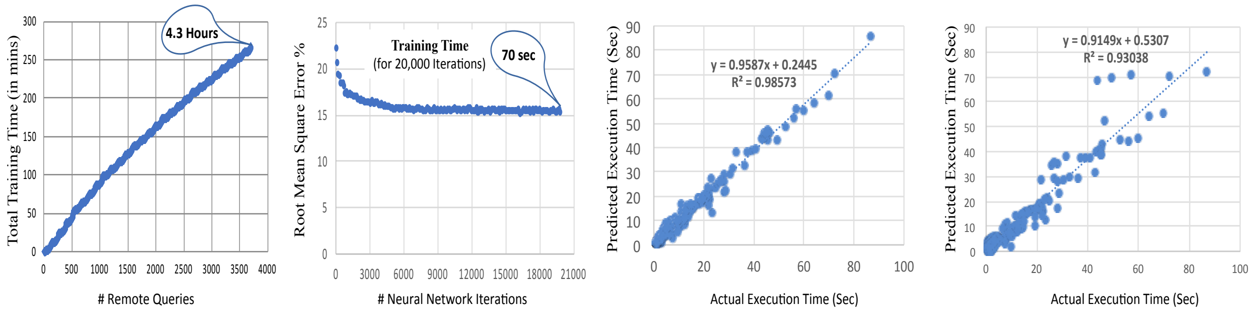
Figure 10: Experimental Setup and Synthetic Dataset Description.

Advanced Learning in Query Optimization: Learning-based models have been studied for both static and dynamic query workloads at coarse-grained plan-level models to fine-grained operator-level models [6]. Machine learning techniques have been also used in the context of query optimizers [17, 29]. The LEO project [17] uses model-based techniques to create self-tuning query optimizer by producing better cost estimates. The work in [29] uses regression techniques to create cost models for XML operators. And the work in [5] proposes building analytical models for query mix interaction to determine good execution schedules. The *IntelliSphere* system combines both the analytical models and machine learning techniques into its cost estimation module.

7 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the various techniques of the *IntelliSphere*'s cost estimation module. As a proof of concept, we study the learning of one remote system, which is Hive/Hadoop. We focus on evaluating the *aggregation* and *join* operators since they are the most expensive operators in the relational model.

Cluster and Dataset Description: The Hive VM cluster has a total of four nodes, one master node and three data nodes. The total HDFS size is 445GBs divided equally across the data nodes. Each node has 8GBs of memory and two CPU cores model Intel(R) E5-2697@2.7GHz. We used synthetic datasets in which we generated 120 tables. The details of the generated tables are summarized in Figure 10. As presented in the figure, we created 20 different configurations for the number of records, and 6 different configurations for the record size. All tables have the same schema as indicated in the figure. The schema is designed such that the



(a) Query Training Cost

(b) Neural Network Conversion Error

(c) Neural Network Model Accuracy

(d) Linear Regression Model Accuracy

Figure 11: Aggregation Logical-Operator: Training Costing & Accuracy over the remote system.

different columns will have different duplication factor, which facilitates the design of the aggregation and join queries to produce specific output cardinalities. Overall, the generated dataset occupies around 45% of the total HDFS capacity (including the default three-fold replication).

Logical-Op Evaluation: In Figures 11 and 12, we present the logical-op evaluation for the aggregation and join operators, respectively. Recall that the aggregation operator has four parameters (four dimensions) training dataset, which are the *number of input rows*, *input row size*, *number of output rows*, and *output row size*. We created a total of approximately 3,700 aggregation queries by varying the target table (from the 120 available ones), and the shrinking factor and the number of computed aggregates as highlighted in Figure 10. Figure 11(a) shows the cumulative training time needed to execute the queries over the remote system (∼ 4.3 Hours).

The collected cost values are then fed to train a neural network model. As discussed in Section 3, the topology of the network has two layers, and the number of nodes in each layer is decided using a cross-validation technique. We omit such details from this section since it is not part of our core contributions. The neural network is trained using 70% of the data points, and then the accuracy is measured using the remaining 30% of the data points. Figure 11(b) illustrates the convergence of the model. It reaches a steady state after 7,000 to 9,000 iterations. The figures shows a total of 20,000 iterations (x-axis), and the y-axis represents the error percentage, which is measured as ($e \times 100/v$), where e is the root mean square error (REMS), and v is the average execution time over all queries. The entire network training takes negligible time (∼ 70 Seconds).

After building the model, the test dataset (30%) is used to test the neural network model accuracy, which is presented in Figure 11(c). The figure shows very high agreement between the actual (x-axis) and estimated (y-axis) execution times. This indicates that the four-parameter model is a good model for the aggregation operator, and that the neural network model can capture the relationship between the inputs and outputs with high precision. In Figure 11(d), we illustrate the model accuracy under a linear regression model instead of the neural network model. For the aggregation operation, the linear regression model shows a reasonable accuracy, although it is still lower than the neural network model.

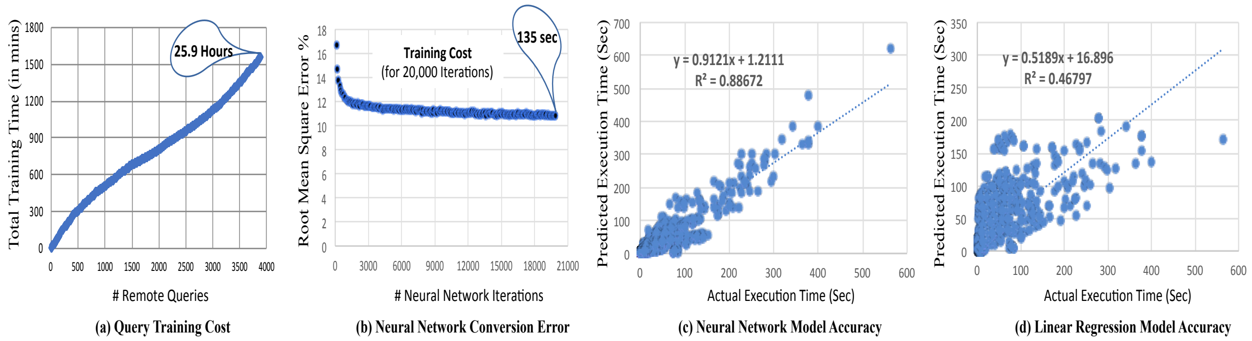
Figure 12 illustrates the training cost and accuracy of the join logical operator. The operator has seven dimensions training set (refer to Figure 2). We created a training set of 4,000 queries by varying the possibilities in each dimension according to the procedure highlighted in Figure 10. Figure 12(a) shows that the training time is really high (∼ 26 Hours). It is worth highlighting that our testing cluster is small, and with bigger clusters, more training configurations need to be covered. Hence, the training time shown in Figures 11(a) and 12(a) can easily grow by an order of magnitude.

In Figure 12(b), we show the convergence and error percentage of the trained neural network model over the training dataset. And Figure 12(c) shows how well the model can learn the execution pattern. We tested the accuracy using the test dataset (30% of the entire data), and the model shows good linear correlation. In Figure 12(d), we illustrate the model accuracy under a linear regression model instead of the neural network model. Unlike the aggregation query type in which the linear regression performed relatively well, in the case of the join queries, the regression model performed poorly and could not capture the execution pattern. Therefore, we believe that for logical operators, it is more accurate and stable to use the neural network model.

Sub-Op Evaluation: For the sub-operator costing approach, the training of each sub-op needs only few number of queries, e.g., in the range of few 10s of queries. As mentioned in Section 4, we did not instrument the remote system to measure the execution time of the sub-op, but instead used primitive queries as presented in Figure 5. Figure 13(a) shows the training time for a number of queries ranging from 6 to 32, which is few minutes. The results from those queries are then used to construct a linear regression model for each sub-op. Figures 13(c), 13(d), and 13(e) illustrate the model of the *WriteDFS*, *Shuffle*, and *Rec Merge* sub-ops, respectively.

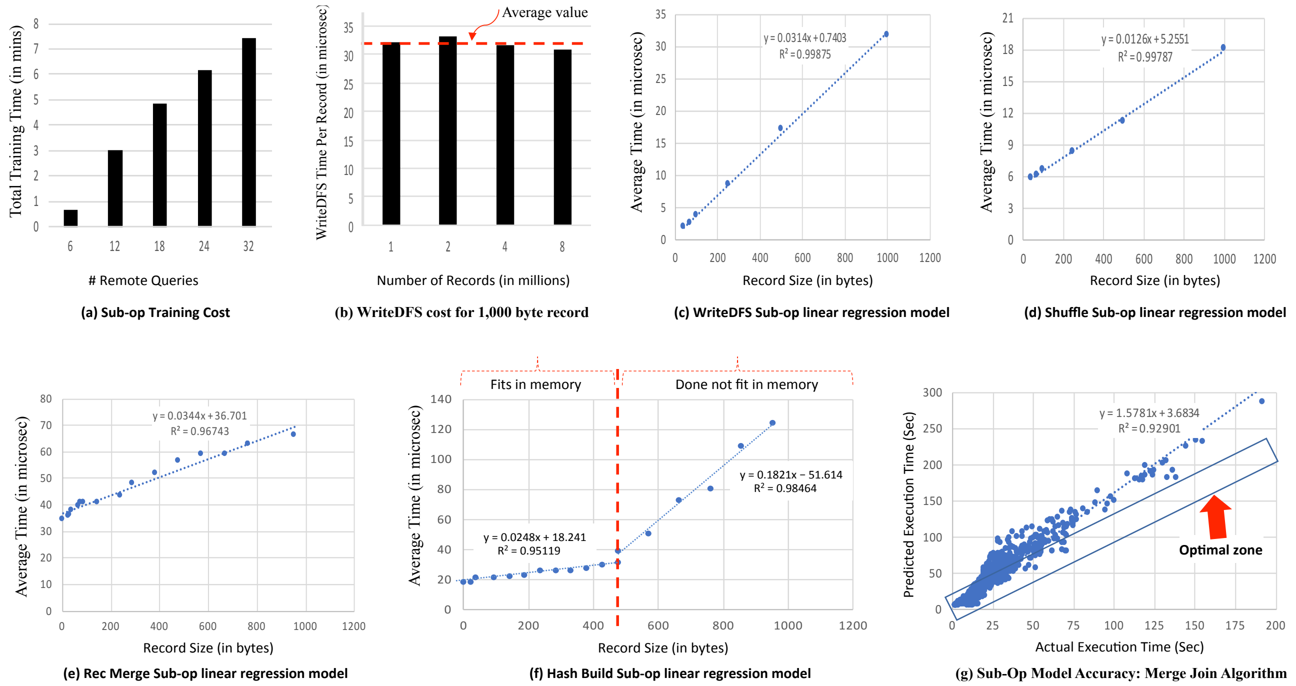
As we discussed in Section 4 while presenting the *ReadDFS* sub-op (Figure 7), we do not construct a separate sub-op model under different dataset sizes (number of rows). Instead, for each record size, say k bytes, we perform four experiments with varying number of rows (1, 2, 4, and 8 millions), and then use the average value to construct a single linear regression model for each sub-op. This average value is shown to be a good-enough representation across datasets as confirmed by the results in Figure 13(b) (for the *WriteDFS* sub-op as an example), and earlier in Figure 7(a) for the *ReadDFS* sub-op.

For the *Hash Build* sub-op an interesting behavior is observed, which is that the results actually resemble two distinct models (See Figure 13(f)). This is because the sub-op is sensitive to whether the hash table fits in memory or not. We experimented with both cases and constructed a model for each case. Recall that the *Hash Build* sub-op is primarily used in the hash join algorithm, where the smaller of the two joined relations is broadcasted to all machines,



(a) Query Training Cost (b) Neural Network Conversion Error (c) Neural Network Model Accuracy (d) Linear Regression Model Accuracy

Figure 12: Join Logical-Operator: Training Costing & Accuracy over the remote system.



(a) Sub-op Training Cost (b) WriteDFS cost for 1,000 byte record (c) WriteDFS Sub-op linear regression model (d) Shuffle Sub-op linear regression model

(e) Rec Merge Sub-op linear regression model (f) Hash Build Sub-op linear regression model (g) Sub-Op Model Accuracy: Merge Join Algorithm

Figure 13: Sub-Op Model: Training Costing & Accuracy over the remote system.

and each machine will build a hash table for this smaller relation. Therefore, given a specific cluster configuration, if the broadcasted relation fits in memory, i.e., falls in the L.H.S area of the vertical dotted line in Figure 13(f), then the corresponding model is used. Otherwise, the system can predict that the broadcasted relation will not fit in memory, and hence the other model is used.

Finally, Figure 13(g) shows the results from combining multiple sub-ops in an analytical formula to estimate the merge join algorithm. Recall that such formula is provided by the domain expert and stored in the remote system profile (Refer to Figures 6). As the results show, the sub-op costing approach provides very good estimation. We found that the sub-op approach slightly tends to overestimate the cost (and similar trend is observed for other algorithms as well), which is a typical trend even within RDBMSs.

Estimation for Out-of-Range Inputs:

In Figure 14, we study the accuracy of the different costing approaches when estimating out-of-range values. This is a typical scenario because an initial training dataset—even if large in size—cannot cover every possible scenario. In this experiment, we studied the merge join algorithm. Both the sub-op and logical-op approaches are trained using datasets of up-to 8x10^6 records with different record sizes. Then, the models are constructed from this training dataset. The figure shows the estimation accuracy for a set of new queries, where the number of input records is 20x10^6, while the record sizes are within the trained ranges. We generated 45 queries with different configurations, e.g., in some configurations only one of the join table is out-of-range and in

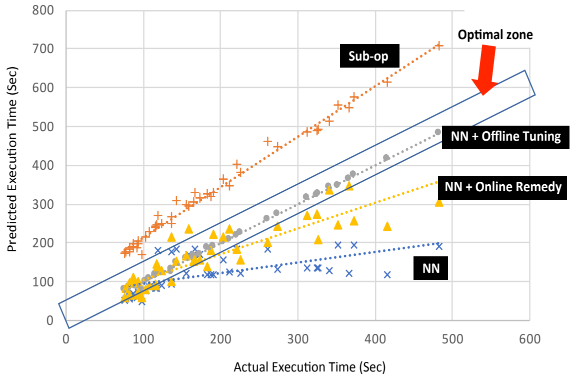


Figure 14: Evaluation of Out-of-Range Prediction Models: Merge Join Algorithm (Fixed α = 0.5).

Table 1: Online Remedy Technique: Automatically Adjusting the Cost-Combining Factor α.

	Batch 1	Batch 2	Batch 3	Batch 4	Batch 5
α	0.5	0.62	0.66	0.57	0.71
RMES%	16.32%	12.6%	12.2%	10.87%	9.1%

other configurations both tables are out-of-range. We compared the estimation accuracy of the *sub-op* approach with that of the *logical-op* approach (the neural network "NN" model).

The results show that the sub-op approach is relatively consistent and can easily extrapolate its trained range to cover out-of-range values. However, due to the non-linearity in the neural network model (the "NN" line), its accuracy degrades and cannot extrapolate well. Interestingly, with the *Online Remedy* technique (Introduced in Figure 4), the accuracy of the estimation improves significantly as depicted in the figure. In this experiment, we fix α (the cost-combining factor) to 0.5.

We also measured the accuracy of the offline tuning phase as follows. We randomly divided the new out-of-range queries (45 in total) into two batches of sizes 70% and 30% roughly. The observed execution times from the 70% batch are added to the neural network model before executing the remaining 30%. And then, the accuracy of remaining 30% is measured. As Figure 14 shows the model adjusts its weights and nicely learns to provide accurate estimations for the new ranges.

Finally, to measure how well the system can adjust the cost-combining factor α in the *Online Remedy* technique (Refer to Figure 4), and its effect on the performance, we performed the following experiment. We initially set $\alpha = 0.5$, and then we randomly divide the 45 out-of-range queries into 5 batches each of size 9. After the execution of each batch, the system adjusts α to minimize the root-mean-square error percentage (RMSE%) of the previously executed batches. The RMSE% is computed as $(e \times 100/v)$, where e is the root-mean-square error (REMS) of a given batch, and v is the average execution time over all queries within that batch. The new value of α is then used for the cost estimation of the subsequent batch. In Table 1, we present the changes of the α values across batches along with the RMSE% for each batch. The results show a trend towards putting a higher weight on the cost factor produced from the neural network, but still the cost produced from the linear regression extrapolation contributes to the final cost by a 30% to 40%.

In summary, as Figure 14 shows, combining the two costs seems effective during the online estimation until the systems collects enough points and applies the offline tuning phase over the neural network model.

8 CONCLUSION

We presented a comprehensive cost estimation module, which is part of the *Teradata IntelliSphere* project. This work addresses a fundamental problem in the modern big data ecosystems, which is the need to efficiently access and query data across multiple heterogenous sources (remote systems). In order to generate efficient execution plans, accurate cost estimation on the remote systems is an essential building block step. We proposed three costing approaches, namely *logical-op*, *sub-op*, and *hybrid* approaches. They cover the spectrum of blackbox, openbox, and a mix of such systems. We demonstrated that none of the *logical-op* or *sub-op* approaches is superior (or practical) in all cases, and thus a hybrid approach should be deployed. We also presented the pros, cons, and applicability cases of each approach. Given the complexity of the problem, we integrated deep learning and analytical models within the proposed cost estimation module. Moreover, we proposed techniques for enhancing the estimation quality for the out-of-range (un-seen) values. As part of future work, we plan to study more types of remote systems such as SparkSQL and Impala.

REFERENCES

[1] MySQL. http://www.mysql.com.

[2] Teradata. *http://www.teradata.com.*

[3] Teradata Query Grid. *Teradata User Group, September 2014.*

[4] A unified software portfolio for a unified analytical ecosystem. Teradata intellisphere. http://www.teradata.com/products-and-services/IntelliSphere.

[5] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Qshuffler: Getting the query mix right. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1415–1417. IEEE, 2008.

[6] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 390–401. IEEE, 2012.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[8] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM computing surveys (CSUR)*, 18(4):323–364, 1986.

[9] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[10] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[11] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogenous information sources. 1994.

[12] H.-H. Do, S. Melnik, and E. Rahm. *Comparison of Schema Matching Evaluations*, pages 221–237. 2003.

[13] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, 44(2):11–16, Aug. 2015.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[15] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, Dec. 2000.

[16] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation and active learning. In *Proceedings of the 7th International Conference on Neural Information Processing Systems*, NIPS'94, pages 231–238, 1994.

[17] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Systems Journal*, 42(1):98–106, 2003.

[18] R. Miikkulainen. *Topology of a Neural Network*, pages 988–989. Springer US, Boston, MA, 2010.

[19] M. Mofidpoor, N. Shiri, and T. Radhakrishnan. Index-based join operations in hive. In *2013 IEEE International Conference on Big Data*, pages 26–33, 2013.

[20] S. Mullender, editor. *Distributed systems (2nd Ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[21] M. Roth, F. Özcan, and L. M. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *VLDB*, 1999.

[22] J. Russell. Couldera-Impala. *O'Reilly Media*, 2013.

[23] M. Stonebraker. The design of the postgres storage system. In *VLDB*, pages 289–300, 1987.

[24] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 5(1):048–063, 1996.

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

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[27] M. Traverso. Presto: Interacting with petabytes of data at Facebook. 2013.

[28] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

[29] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing xml queries. In *Proceedings of the 31st international conference on Very large data bases*, pages 289–300. VLDB Endowment, 2005.

[30] Q. Zhu and P. . Larson. Building regression cost models for multidatabase systems. In *Fourth International Conference on Parallel and Distributed Information Systems*, pages 220–231, 1996.

[31] P. Ziegler and K. R. Dittrich. *Data Integration — Problems, Approaches, and Perspectives*, pages 39–58. Springer Berlin Heidelberg, 2007.