Apache Wayang: A Unified Data Analytics Framework

Kaustubh Beedkar[*,‡], Bertty Contreras-Rojas[◇], Haralampos Gavriilidis[◇], Zoi Kaoudi[⋆,‡], Volker Markl[◇], Rodrigo Pardo-Meza[◇], Jorge-Arnulfo Quiané-Ruiz[⋆,‡]

Indian Institute of Technology Delhi[*] Technische Universität Berlin[◇] IT University of Copenhagen[⋆] Databloom Inc.[‡]

ABSTRACT

The large variety of specialized data processing platforms and the increased complexity of data analytics has led to the need for *unifying data analytics* within a single framework. Such a framework should free users from the burden of (i) choosing the right platform(s) and (ii) gluing code between the different parts of their pipelines. Apache Wayang (Incubating) is the only open-source framework that provides a systematic solution to unified data analytics by integrating multiple heterogeneous data processing platforms. It achieves that by decoupling applications from the underlying platforms and providing an optimizer so that users do not have to specify the platforms on which their pipeline should run. Wayang provides a unified view and processing model, effectively integrating the hodgepodge of heterogeneous platforms into a single framework with increased usability without sacrificing performance and total cost of ownership. In this paper, we present the architecture of Wayang, describe its main components, and give an outlook on future directions.

1. INTRODUCTION

The research and industry communities have developed a variety of data processing platforms (platforms, for short) to enable users to efficiently extract value from their data. Each platform excels in different aspects of the design space. For instance, PostgreSQL performs better than Vertica for OLTP workloads, but Vertica performs better for OLAP workloads. Apache Spark, on the other side, can outperform both database systems for batch data processing on big datasets.

Consequently, users face a *zoo of specialized platforms* to perform data analytics. They typically run their data analytics at a higher cost than necessary, as selecting the right platform is daunting. Furthermore, modern applications often require to perform data analytics that goes *beyond the limits of a single* platform, making the selection of platforms even more difficult.

To ease the platform selection task, we require *unifying data analytics* within a single framework, i.e., applications should run over any set of platforms seamlessly and efficiently. The need for unified data analytics can stem from simple tasks, such as k-means clustering, to very complex tasks, such as a data science pipeline that includes data cleaning, preparation, feature extraction, and model training. Unified data analytics is quickly becoming essential as new applications emerge.

We distinguish between two general cases of unified data analytics: (i) an entire task is executed on a single platform and, based on the circumstance, this platform can vary, and (ii) a task is split into sub-tasks which are executed on multiple platforms. In particular, we identify four situations when unified data analytics is required [10]: **Platform Independence** refers to the situation where one needs to run an entire task on any arbitrary platform. This requires re-implementing applications when new platforms emerge or when the workload changes. **Opportunistic Cross-Platform** refers to the situation where performing a single task using multiple platforms brings significant performance reasons. **Mandatory Cross-Platform** refers to the fact that modern applications need to go beyond the functionalities offered by a single platform. **Polystore** refers to the situation where applications need to access and process data stored in different data stores (data lakes). In all the above cases, developers typically must write ad-hoc programs to wire multiple platforms together. However, integrating platforms is tedious, repetitive, and error-prone. Figure 1 illustrates these four cases with systems that can handle each case.

Therefore, supporting unified data analytics is crucial in many cases. Apache Wayang (Incubating)[1], Wayang for short, is the first open-source framework that provides applications with unified data analytics capabilities. The main goal of

[1]https://github.com/apache/incubator-wayang

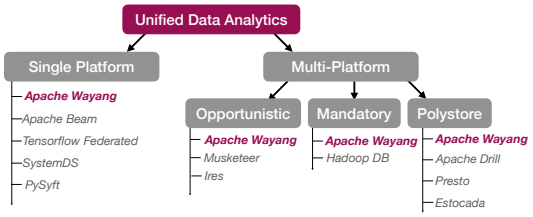

Figure 1: Taxonomy for unified data analytics.

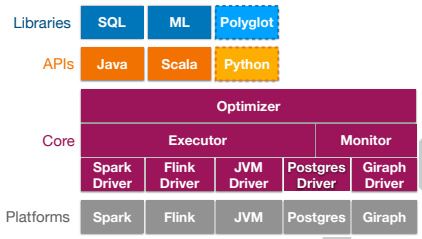

Figure 2: Wayang's software stack.

Wayang is to decouple applications from platforms so that they can run analytics on one or more platforms seamlessly and efficiently. Developers can provide their data analytics tasks programmatically (using Java, Scala, or Python) or declaratively (via the SQL and ML libraries). Wayang, in turn, takes an input task and *optimizes* it to produce efficient execution plans, which might run over multiple platforms. Here, we refer to an efficient execution plan as the plan that allows Wayang to execute a given task with a low execution cost. By default, Wayang considers the cost to be the execution time, but users can provide their own cost function, such as monetary cost.

As of today, Wayang supports a variety of platforms: Spark, Flink, PostgreSQL, GraphX, Giraph, and its in-memory Java-based executor[2]. Wayang presents itself as a full-fledged and efficient cross-platform data processing system for unified data analytics. Wayang originated from the Rheem project [3, 13], is currently incubating in the Apache Software Foundation, and is used by several companies. In particular, Databloom, an AI startup, has been created around Wayang [2].

Our contributions in this paper are as follows. We introduce Apache Wayang (Incubating), which comes with a novel system architecture allowing the integration of different platforms (Section 2). Then, we present the core components of Wayang, including a cross-platform query optimizer that alleviates users from any platform decisions (Section 3). Furthermore, we introduce Wayang's approach to running data analytics on any platform (Section 4). We additionally present the Polyglot module, which allows developers to add support for UDFs in any desired programming language by implementing only two core abstractions. Finally, we briefly discuss Wayang's adoption (Section 5), related work (Section 6), and our current efforts towards Wayang 2.0 (Section 7).

2. OVERVIEW

Wayang's main goal is to unify data analytics by decoupling applications from the underlying platforms and to provide cross-platform data processing. Figure 2 shows the software stack of Wayang.

In the bottom layers, there are the different data storage mediums and the supported data processing platforms. On top of these, Wayang's core consists of the following main components: the optimizer, the executor, the monitor, and platform-specific drivers. Wayang currently supports two main APIs: the Java one and the Scala one. A Python API is currently under development. Besides using any of the supported languages, users can directly input SQL queries via the SQL library, which transforms them into a Wayang plan. Wayang also comes with an ML library for running ML tasks. Users can directly utilize the provided algorithms or can implement their own algorithm using a simple ML abstraction [11]. To enable support for more programming languages in an efficient way, Wayang comes with a Polyglot library (see Section 4.3).

Wayang relies on *data quanta*, the smallest processing units of the input datasets. A data quantum can express a large spectrum of data formats, such as database tuples, edges in a graph, or data points required by machine learning. Wayang's main building block is a *Wayang plan*: a directed dataflow graph whose vertices are platform-agnostic operators and whose edges represent data flowing among the operators. An example Wayang plan for the Wordcount task is depicted in Figure 3(a). A user or application can specify a Wayang plan by using any of the three supported languages (Java, Scala, and Python). Importantly, one does not have to specify the platform on which each plan's operator will be executed. Given a Wayang plan, the optimizer is responsible for determining the platform on which each operator has to be executed, thereby composing a platform-specific execution plan. The executor is then responsible for assigning the operators of the execution plan to the corresponding drivers and coordinating the execution. The monitor checks whether the estimations used during the optimization are correct, and if not, requests a new execution plan from the optimizer.

[2]GraphChi is outdated and is going to be removed in our next release.

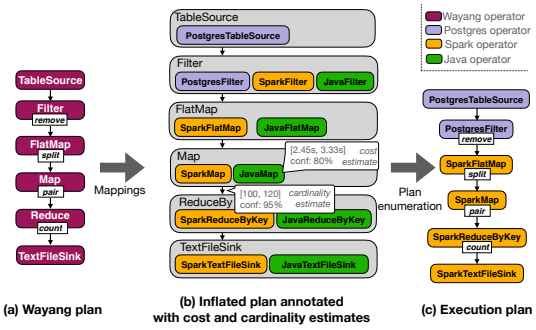

Figure 3: Wayang's optimization phase by example.

3. THE CORE OF WAYANG

Wayang's core comprises a query optimizer and an executor. The former determines which underlying platform each (sub)query has to be executed on to achieve the best performance. The latter is responsible for scheduling and submitting the (sub)queries for execution. The main building block of Wayang is a *Wayang plan*, a graph where each node is a Wayang (platform-agnostic) operator of the following types: unary, binary, loop, source, or sink. In addition, a Wayang operator can be either atomic (e.g., `map`) or composite (e.g., `pagerank`), i.e., composed of many atomic operators.

Query Optimization. The optimizer receives as input a Wayang plan and outputs an execution (platform-specific) plan with the goal of minimizing the total execution cost. To achieve this, the optimizer first "inflates" the plan (Figure 3): For each node that corresponds to a Wayang operator, it adds all the corresponding execution operators. This is done via *flexible graph-based mappings* that map one or more Wayang operator(s) to one or more platform-specific execution operator(s). A composite Wayang operator, such as `pagerank`, is mapped either to a platform-specific composite operator (e.g., MLlib's pagerank) or to a graph of platform-specific operators that perform the required functionality.

Once the inflated plan is created, the optimizer attaches not only the operator's costs but also the costs for moving intermediate data from one platform to another (omitted in the figure for simplicity reasons). Currently, Wayang uses linear cost formulas to estimate these costs. The system administrator needs to fine-tune the coefficients of these formulas. Although Wayang comes with profiling tools to facilitate this tuning effort, our near plans include the ability to easily port machine learning models for estimating the costs, as discussed in [12].

At the last step of query optimization, an enumeration algorithm considers available options to output the optimal execution plan w.r.t. a defined cost. The metric for the optimization cost can be anything, from runtime to monetary cost or energy consumption. As the search space is exponential (a plan with n Wayang operators, each having k execution operators, leads to k^n possible execution plans), pruning is crucial. Wayang's enumeration algorithm is based on an algebra consisting of two main operations: *Join* for concatenating subplans and *Prune* for pruning subplans that lead to inferior execution plans. Importantly, the pruning strategy is lossless. It is based on the notion of boundary operators which are the start and end operators of a subplan and is guaranteed to not prune a subplan that is part of the optimal execution plan.

Note that users can control the optimizer by specifying in their code where an operator has to be executed via the `withTargetPlatform(plat)` call on the desired operator. Then, the optimizer takes into consideration the decisions of the user and outputs an execution plan by navigating a reduced search space during the plan enumeration.

Data movement. We now detail how Wayang computes the costs incurred when moving data from one platform to another during the query optimization process. As there may be multiple ways to move data from platform A to platform B, Wayang represents the space of different communication ways as a channel conversion graph. This graph contains the different data types (channels) as vertices (e.g., RDD or Relation). Two channels are connected with a direct edge denoting that the source channel can be converted to the destination channel via one or more conversion operators. Conversion operators can be the standard source and sink operators of the underlying platforms. During query optimization, Wayang finds the optimal communication path from one channel to another by formulating the problem as a minimum conversion tree problem (proved to be NP-hard [14]). The interested reader is referred to [13] for more details.

Execution. Given an execution plan output by the query optimizer, the executor of Wayang is responsible for scheduling its execution. First, it divides the plan into stages so that each stage forms a subplan where all its execution operators are of the same platform. Stages are connected by data flow dependencies. The executor dispatches a stage to the relevant platform driver, which in turn submits the sequence of operators as a job to the underlying platform. If there are stages with no dependencies, the executor dispatches them in parallel. In any other case, it dispatches a stage once its input dependencies are satisfied. After each stage, the plat-

form gives back the control to the executor so that it either initiates the next stage or it materializes the output in the case of the final stage. The executor may also create more than one stage for a sequence of execution operators of the same platform in cases where the executor needs the control (e.g., when the executor needs to evaluate the loop condition in an iterative operator).

Re-optimization. It is well-known that poor cardinality and cost estimates can negatively impact the effectiveness of an optimizer. This is even worse in our setting where the semantics of UDFs and data distributions are usually unknown because of the little control over the underlying platforms. For this reason, Wayang's optimizer allows for re-optimizing a plan whenever observed cardinalities greatly mismatch the estimated ones. Similar to [15], it achieves this by adding optimization checkpoints between stages in the execution plan whenever the cardinality estimates have low confidence or the data is at rest (e.g., file). When the executor encounters a checkpoint between stages, it pauses the plan execution and gives the control to the optimizer to consider a re-optimization of the plan beyond the checkpoint. The optimizer uses the observed cardinalities and recomputes the most efficient plan since the last optimization checkpoint. It then gives the new execution plan to the executor, which resumes execution considering the new plan.

4. ANY ANALYTICS ANYWHERE

We now describe the libraries that Wayang currently supports (SQL and ML) and one that is under development (Polyglot). These libraries are built atop the native Java API of Wayang. Note that Wayang also provides a Scala API and a Python API is currently under development. In the following, we first describe its SQL and ML libraries. We, then, detail the Polyglot library, which is the Wayang's approach to support UDFs coded in different programming languages.

4.1 SQL analytics anywhere

A feature of Wayang is its unified SQL interface for cross-platform data processing. The SQL library allows users to embed SQL queries in their applications via the `SqlContext` object, which holds the configurations about different data sources. The following snippet shows how users can specify SQL queries using its Java API.

```
SqlContext sqlContext = new SqlContext(conf);
Collection<Record> result = sqlContext.
        executeSql("SELECT ... FROM ...");
```

The `sqlContext` provides methods that return

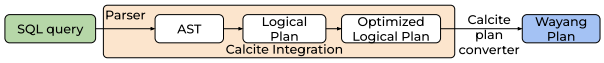

Figure 4: SQL query preparation in Wayang.

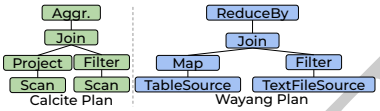

Figure 5: Example Calcite plan converted to a Wayang plan.

the result of the `SELECT` statement as a collection of Records, which can be converted to a data quanta or used in subsequent SQL queries. This allows Wayang to seamlessly integrate SQL queries into applications and holistically optimize them.

Wayang's SQL library utilizes Apache Calcite [4] to support the SQL standard. Yet, to execute an SQL query, we first have to translate it into a Wayang plan, as shown in Figure 4.

Wayang comes with a Calcite-based SQL parser and optimizer. The SQL query is first translated into a Calcite logical plan from its AST, which is then optimized and subsequently converted into a Wayang plan. Our Calcite integration facilitates database optimizations, such as operator pushdown, reordering, and elimination.

Converting a Calcite plan into a Wayang plan is done on a per-operator basis. Figure 5 shows an example translation. Common translations include: `tableScan` operators to Wayang source operators; `project`, `filter`, `join`, and `aggregation` operators to Wayang's `Map`, `Filter`, `Join`, and `ReduceBy` operators, respectively. During the plan conversion step, a SQL operator is translated into Java functions, which are then wrapped by a single UDF.

Wayang currently offers support for developing applications with the SQL interface in Java. Our future work will include bindings for Scala as well as for Python. Moreover, the SQL language support in Wayang only includes support for `SELECT` statements as the core focus of the Apache Wayang project is to enable cross-platform data analytics (rather than data management). In future, we also plan to include a JDBC client in Wayang. This will enable Wayang's compatibility with other external BI tools, such as Tableau.

4.2 Machine learning anywhere

Wayang also comes with a machine learning (ML) library, which allows users to create ML pipelines using Wayang's operators and/or write their ML algorithms using a predefined small set of primitives [11]. This set of primitives is sufficient for implementing a wide variety of iterative ML algorithms, such as any gradient descent algorithm, k-means clustering, or expectation-maximization

algorithms. After analyzing such ML algorithms, we found that they all can be split into three different phases: preparation, processing, and convergence. Wayang's ML library, thus, provides the following seven operators:

(1) `Transform` receives a data point to transform (e.g., normalize it) and outputs a new data point.

(2) `Stage` initializes all the required global parameters (e.g., centroids for the k-means algorithm).

(3) `Compute` performs user-defined computations on the input data point and returns a new data point. For example, it can compute the nearest centroid for each input data point.

(4) `Update` updates the global parameters based on a user-defined formula. For example, it can update the new centroids based on the output computed by the `Compute` operator.

(5) `Sample` takes as input the size of the desired sample and the data points to sample from and returns a reduced set of sampled data points.

(6) `Converge` specifies a function that outputs a convergence dataset required for determining whether the iterations should continue or stop. For example, it can be the difference between two subsequent values of each centroid.

(7) `Loop` specifies the stopping condition on the convergence dataset.

All the above operators serve as UDFs. While we provide reference implementations for common algorithms, users can easily customize or override them. The first two operators are used in the preparation phase, while `Compute`, `Update`, and `Sample` are used iteratively in the processing phase. The interested reader can find more details in [11].

Once these operators are defined, the ML library transforms them into a Wayang plan. The plan is then passed to the optimizer to determine the right platform. Thus, data scientists can use Wayang to develop new algorithms and test them with small datasets, which will be run locally. The same code can seamlessly be used on deployment for larger datasets potentially running in a big data platform. However, the data scientist does not have to worry about the underlying deployment of a plan.

4.3 UDFs coded anywhere

Besides its Java, Scala, and Python APIs, which are dedicated interfaces for programming languages, Wayang provides a library to support UDFs coded in different programming languages, such as Go. Continuing with its search for platform interoperability, Wayang goes one step forward when it comes to supporting UDFs. It offers the Polyglot library to support the execution of UDFs in any programming language and in an efficient way. The execution of UDFs in other programming languages is crucial for exposing Wayang to other programming languages besides Java, Scala, and Python.

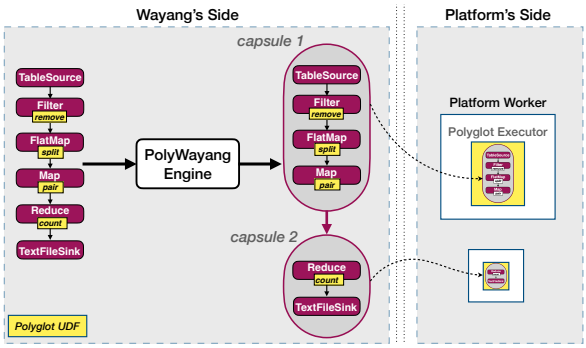

Figure 6: Execution flow with Polyglot.

The execution flow with Polyglot is shown in Figure 6. To support UDFs in a particular programming language, Wayang developers must implement a `FunctionWrapper` and a `WorkerManager`. While the former allows Wayang users[3] to provide their UDFs in any supported programming language (e.g., Go), the latter allows Wayang to launch the required runtime (e.g., the Go runtime). As a result, Wayang can invoke the programming language runtime (e.g., the Go runtime via a `GoWorkerManager` implementation) with the UDF (e.g., the Go UDF via a `GoFunctionWrapper`). For performance reasons, Wayang does so by *encapsulating* operators that belong to the same stage (i.e., pipelined operators) into a `MapPartition` operator. This results in a single call to the programming language runtime for the entire data quanta of the `MapPartition` instead of having a runtime call per input data quantum.

5. ADOPTION

Wayang is increasingly gaining traction in both industry and academia. In academia, Wayang has fostered several database research projects in query optimization, data integration, and polyglot data management. In industrial settings where multi-platform infrastructures are routine, Wayang has provided a cost-effective alternative to run complex analytics without having to develop platform-specific solutions. Wayang is being used, among others, in machine learning, data cleaning, and data analytics applications. For instance, an airline company is assessing Wayang to carry out large-scale

[3]We distinguish between developers and users in the way they interact with Wayang: while the developers write code to extend or fine-tune Wayang, the users only use it via its libraries and APIs.

data analysis for optimizing air cargo revenues [16]. NADEEF [5], a commodity data cleaning system, uses Wayang to boost its performance through platform independence. Wayang has also led to the creation of Databloom [2], an AI startup that aims at providing an easy-to-use, cost-efficient, and data-compliant solution to companies having distributed and heterogeneous data infrastructures.

6. RELATED WORK

Open source. To our knowledge, Wayang is the only open source system that not only decouples applications from the underlying platforms but also provides a way for automatically determining on which platform(s) a given task should be executed. Perhaps, most related to Wayang is Apache Beam [1], which focuses on providing a unified model for batch and streaming data processing and being portable to any data processing platforms. The latter means that a user's pipeline is entirely executed on one data platform (runner), which users need to specify. In contrast, Wayang users are free to either not specify at all where their pipelines are executed or create hybrid pipelines integrating multiple data processing platforms.

Academic. There have been early efforts to unify data analytics in a systematic way [9, 7]. However, while [7] requires expertise from users for deciding when to use a data processing platform, the design of [9] is not flexible enough to allow for continuous extensions with new platforms, i.e., developers have to modify the source code of the platforms. IReS [6] on the other hand, goes one step further and provides a flexible and automatic way to choose data processing platforms. However, in contrast to Wayang, it focuses more on coarse-grained operators (e.g., `k-means`, `tf-idf` instead of `filter`, `map`) and provides 1-to-1 mappings from abstract to execution operators, which may lead to suboptimal execution plans. For instance, for a simple stochastic gradient descent algorithm that could be viewed as an operator by itself, Wayang can provide significant performance benefits by splitting it into more fine-grained operators [3].

7. TOWARDS WAYANG 2.0

Apache Wayang (Incubating) facilitates automatic cross-platform data processing. Its extensible framework integrates various data processing platforms, decoupling applications from specific platforms. Wayang provides a unified framework for analytics and aims to support fully decentralized applications through a *delegation phase* and *direct communication channels*.

Task Delegation The goal is to decentralize execution by offloading processing and communication tasks to underlying platforms, avoiding a centralized Executor. This involves introducing delegation tasks (akin to [8]) that combine data manipulation and movement instructions. Task delegation will reduce the need for resource-intensive execution coordination and platform communication.

Direct Communication Channels To enable task delegation, we will prioritize direct communication between platforms and develop new abstractions within the Wayang operator model. These abstractions will enhance platform interoperability and eliminate the need for generic conversion channels where communication operators require intermediate mediums (e.g., CSV files or Java collections). Wayang can then also optimize data movement on a platform level by implementing techniques such as data layouts and compression.

8. REFERENCES

[1] The Unified Apache Beam Model. https://beam.apache.org. Retrieved May, 2023.
[2] Databloom ai: https://www.databloom.ai/, 2022.
[3] D. Agrawal et al. Rheem: enabling cross-platform data processing – may the big data be with you! In *PVLDB*, 2018.
[4] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, page 221–230, 2018.
[5] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, pages 541–552. ACM, 2013.
[6] K. Doka, I. Mytilinis, N. Papailiou, V. Giannakouris, D. Tsoumakos, and N. Koziris. Multi-engine analytics with ires. In *BIRTE*, pages 133–154, 2016.
[7] 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 Record*, 44(2):11–16, 2015.
[8] H. Gavriilidis, K. Beedkar, J.-A. Quiané-Ruiz, and V. Markl. In-situ cross-database query processing. In *ICDE*, 2023.
[9] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: all for one, one for all in data processing systems. In *EuroSys*, 2015.
[10] Z. Kaoudi and J. Quiané-Ruiz. Unified data analytics: State-of-the-art and open problems. *Proc. VLDB Endow.*, 15(12):3778–3781, 2022.
[11] Z. Kaoudi, J.-A. Quiané-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal. A cost-based optimizer for gradient descent optimization. In *SIGMOD*, 2017.
[12] Z. Kaoudi, J.-A. Quiané-Ruiz, B. Contreras-Rojas, R. Pardo-Meza, A. Troudi, and S. Chawla. ML-based Cross-Platform Query Optimization. In *ICDE*, pages 1489–1500, 2020.
[13] S. Kruse, Z. Kaoudi, B. Contreras-Rojas, S. Chawla, F. Naumann, and J. Quiané-Ruiz. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *VLDB J.*, 29(6):1287–1310, 2020.
[14] S. Kruse, Z. Kaoudi, J.-A. Quiané-Ruiz, S. Chawla, F. Naumann, and B. Contreras-Rojas. Optimizing Cross-platform Data Movement. In *ICDE*, 2019.
[15] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.
[16] S. G. Rizzo, Y. Chen, L. Pang, J. Lucas, Z. Kaoudi, J. Quiané-Ruiz, and S. Chawla. Prescriptive learning for air-cargo revenue management. In *ICDM*, pages 462–471, 2020.