THE UNIVERSITY OF CHICAGO

IMPROVING DATA'S UTILITY

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
TAPAN SRIVASTAVA

CHICAGO, ILLINOIS
JUNE 2026

ORCID iD: https://orcid.org/0009-0000-8592-4218

To my Baba, the kindest person I've ever met. I hope I am like you in all the most important ways.

And to my four nieces, Ananya, Anisha, Asha, and Nithya, who bring me so much joy and who I hope I get to share my favorite movies with for many years to come.

"I do not know which one of us has written this page."

— Jorge Luis Borges, *Borges and I*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I want to first thank my advisor, Raul Castro Fernandez, for his time and mentorship over these years. I have grown as a researcher and worker in Raul's lab, and he inspires me to aggressively pursue the highest possible standards for myself and my work. Most of the projects I worked on during this degree have been primarily just the two of us, and I have grown significantly as a result of working so closely with Raul. I also want to thank my committee members, Aaron Elmore and Michael Franklin, both of whom provided me with valuable feedback. I have learned a lot TA'ing classes for both of them and getting to work with them in teaching environments. I appreciate your help and guidance through my degree.

I am thankful for my lab mates, who provided companionship and compassion during my degree. In particular, talking with Chris Zhu has calmed me down from many spikes of anxiety, and being able to give Chris advice during his degree has been very rewarding for me and has acted as a way to see how far I'd come myself.

Finally, I am thankful for my family and the many friends I've made. To my partner, Hartrich, who I simply would not have survived without. I have cherished every moment of our seven years together, and I am incredibly excited for the many more years together to come. There are more friends than I can possibly name, but I'll try: to Lauren, Jake, Pouya, Anup, Renée, Andy and Evie and Cosmo, Laura and Hope, Jill, Dan, Owen, Ted, Ethan, Arjun, Noah, Zene, Sheldon, Trevor, Mihir, and all the other friends whom I haven't named. Your friendship means more to me than can be expressed, and I love you all deeply.

And to my family: my Amma and Papa, my Dada and Didi, Bhavya and Travis, and most especially to my four nieces, Ananya, Anisha, Asha, and Nithya, I am eternally grateful for your love and support. I would not be here without you all.

Any accomplishments in my life are due to the strong support system I have around me, tremendous luck and privilege, and hard work. I am deeply indebted to those who prop me up and enable me to be better every day. Thank you, all.

ABSTRACT

Users must make many different data management decisions about how to store, transform, and process their data in the cloud to best serve organizational goals. Because cloud services employ on-demand pricing models, dimensions such as monetary costs are just as important as task performance when evaluating the efficacy of a potential data management decision.

These problems each require different expertise and concern different data systems and are thus often solved in isolation using a mixture of heuristic signals, monitoring tools to detect issues, and manual investigation and tuning. We argue that across seemingly disparate data management problems, these approaches could be improved by identifying a common signal that directly indicates the benefit an organization achieves from its data. This benefit, called data utility, is a function of the value achieved from tasks run on data and the costs of running tasks and maintaining data. Though the particular utility function may vary between data managers and organizations, data's utility can be increased by reducing costs or improving the value achieved from tasks on that data.

This dissertation presents solutions that improve data's utility by building cheaper execution plans across multiple cloud data warehouses for analytical workloads and which provide users with an indicator for the value they have achieved from their data as a signal to make better data management decisions. These solutions together provide concrete mechanisms to enable users to directly interact with and improve their data's utility.

CHAPTER 1

INTRODUCTION

1.1 Managing Data in the Cloud

Cloud vendors, such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP), provide computing resources over the internet like scalable storage, elastic virtual machines, and platform-as-a-service offerings which enable efficient data processing while fully managing all underlying infrastructure. For example, Amazon Redshift [Gupta et al., 2015] or Google BigQuery [BigQuery] provide fully managed analytical databases which organizations can pay for on-demand to run SQL queries.

Users, ranging from individuals and small startups to large, international companies, configure and use these cloud services to store, transform, and analyze their data in the cloud. For instance, users may decide to store data in centralized repository like a data lake for ease of access, to manually prepare data using cloud compute resources to ensure accuracy for key tasks, to allocate some resources to a cloud data warehouse to run analytical SQL queries, and so on.

Users aim to efficiently decide what services to use and how to configure them in ways that best serve organizational goals. For instance, for users with strict monetary budgets, a cloud setup that keeps costs low even at the cost of some performance degradation is more efficient than an expensive cloud setup that achieves the best performance.

Because users run a wide range of tasks on their data—such as batch workloads like fraud detection pipelines or interactive queries run by business analysts exploring datasets—determining a cloud setup which efficiently serves organizational goals is a complex problem. Users' goals may also change, requiring them to identify how to efficiently modify their cloud setup to meet these new goals better.

Each decision about what system to use, how to configure it, and what processes to

implement on it, are seemingly disparate decisions requiring different expertise. For instance, deciding how to run analytical tasks, like SQL queries or Python notebooks, requires an understanding of cloud data warehouses, data lakes and storage formats, and data ingestion and transformation pipelines along with how to configure these complex systems to best meet organizational goals. In contrast, a user deciding what data should be prepared manually requires understanding what tasks are sensitive to data quality, the quality of the data, and how many resources are required to prepare a piece of data, among other factors.

Consequently, data managers—or those users responsible for making these decisions such as engineers, administrators, or analysts—often approach each decision individually, applying bespoke solutions to each which often cannot be applied to other problems.

Making efficient data management decisions is critical for organizations. For instance, companies that use AWS S3 Intelligent tiering, which automatically archives data based on data age and access frequency, to solve their data lifecycle problems have saved $6 billion since 2018 compared to those who used AWS S3 standard storage [Amazon S3 Intelligent-Tiering]. However, poor solutions to these problems can similarly hurt an organization; for example, organizations lose $12.9 million annually on average because data quality was poor for key tasks [Gitnux, 2026].

However, data managers often cannot feasibly solve these problems perfectly; for instance, identifying what storage tier data should be stored in to minimize costs while maintaining performance for key tasks would require evaluating many different assignments of data to storage tiers, subdividing data into different pieces to assign to different tiers, and measuring query performance and overall costs to ensure that goals are met. This approach would likely require an infeasible amount of human time and resources.

Data managers thus rely on techniques that aim to approximate an efficient solution to these complex problems, such as signals like access frequency to guide data lifecycle decisions or cloud tools like those for data monitoring and observability. However, these may guide

data managers to inefficient decisions which do not best align with organizational goals. For instance, relying solely on a proxy heuristic like data access frequency to decide whether to archive data may significantly slow down a key task that accesses data infrequently even though maintaining its performance is a major goal for the organization. Furthermore, these techniques often still require data managers to perform significant manual work, such as defining specific data lifecycle rules to govern when data moves to archival storage.

These approaches could be improved by identifying a common signal applicable across these problems which directly indicates how well a data management decision achieves organizational goals. This signal could be used to find more efficient solutions and can be applied across a range of different data management problems.

1.2 Current Approaches to Data Management Problems

We now summarize some current approaches for specific data management problems and discuss the limitations present with these approaches. One key data management problem is identifying and removing bottlenecks in data governance. Architecturally, the data mesh paradigm [Dehghani, 2020] delegates data ownership to domain teams rather than small, centralized teams to remove bottlenecks from a tremendous amount of data governance work being handled by a small team. However, this approach requires each domain team to have data engineers capable of handling governance and significant organizational effort to implement this paradigm, making it infeasible for many organizations to adopt.

Some approaches similarly aim to mitigate or remove bottlenecks in data migration and transformation. The data lakehouse [Armbrust et al., 2021] provides a unified platform for both data lake and data warehouse which provides several benefits, including reducing storage costs, data redundancy, and costs of extract, transform, load (ETL) processes. The zero-ETL paradigm [Heroor, 2024] similarly aims to reduce the latency and complexity of ETL pipelines, which may be brittle to changes in the data source or downstream tables,

by using real-time updates from data sources to update downstream tables via change data capture (CDC) systems. However, both of these approaches have downsides: the lakehouse paradigm may not be able to achieve the same performance as a data warehouse for key workloads, and the zero-ETL paradigm may not be able to handle all transformations required by workloads and can introduce latency to source systems that have many updates. Thus, data managers must still manually weigh the tradeoffs of these approaches in the context of their own data, tasks, and organizational goals.

Another key data management problem is data lifecycle management. Cloud vendors offer multiple storage tiers which trade off monetary costs for data access latency. Data managers must decide what data to store in what storage tier. One approach is to perform rule-based static tiering [Khan et al., 2023], where rules are defined as if-then statements based on signals such as data age or access frequency to decide what tier to store data in. These rules are either manually written by data managers or inferred based on proxy heuristics like access frequency. Other methods like AWS Intelligent Tiering [Amazon S3 Intelligent-Tiering] use access frequency and predict future access patterns to make more efficient storage decisions. However, these approaches all rely on signals that can be imperfect proxies for organizational goals and thus could lead to inefficient decisions or require significant manual effort by data managers to make these decisions.

Beyond concerns about data storage or migration between systems, data managers often simply want to reduce monetary costs spent in the cloud. However, these costs are incurred from many services, so there are various existing approaches to save money focusing on a single source of cloud cost. To reduce storage and maintenance costs, data managers apply deduplication techniques which use hash functions to find duplicate data blocks and compress data [Chen, 2024]. However, deduplicating data is a computationally expensive process, so data managers must still manually decide which datasets would be most efficient to deduplicate. To reduce SQL query costs, data managers may manually try to rewrite SQL queries

more efficiently to reduce the amount of data scanned and thus reduce costs [BigQuery Optimize Query Computation]. Data managers may rely on consulting groups like the DuckBill Group, CloudZero, or McKinsey [The Duckbill Group, CloudZero, McKinsey], which provide setup-specific recommendations like turning off unused resources or using reserved instance pricing, to save costs; or, data managers may build in hard budget constraints to each team in their cloud setup and use cloud vendors to enforce these project quotas to prevent teams from exceeding their budget [Williams and Joshi, 2023]. These approaches all ultimately require data managers to manually understand the needs of the organization and the tradeoffs between costs and value for different tasks to identify an efficient decision. We illustrate this in the following example.

Example 1.1: *An organization stores multiple relational tables in the standard Amazon S3 blob storage tier, with each table stored in multiple Apache Parquet files, and databases directly scan data from S3 to run queries. Observing that the total cloud expenditure on storage and data access in the last two quarters has exceeded budget estimates, a financial officer asks a data manager to reduce monetary costs for storage for the next quarter.*

The data manager aims to save money by moving some data to cheaper storage tiers, such as S3 Glacier, which charges less for storage but increases latency to retrieve data from milliseconds to minutes or hours. The data manager chooses to use access frequency to guide their decision, storing the least frequently used tables in cheaper storage until the savings goal is met. Some tables, which are only stored for financial compliance but are not accessed by queries otherwise, are moved to Glacier, while tables accessed multiple times every day to run key services are kept in the standard S3 storage tier. However, there is one table that is accessed infrequently, only a few times a month, that is moved to Glacier storage. Soon after, though the monetary savings goal is achieved, the data manager receives an alert that a system that makes recommendations to users on what content to watch has been disrupted. Upon investigating, they find that the service retrains less frequently but

uses the infrequently accessed table and long-term historical data, which have been moved to Glacier. The increased read latency delayed the retraining process, leading to a degradation in the quality of recommendations and a loss in customer satisfaction and retention. [McDowell, 2025, Noble, 2025]

Example 1.1 illustrates how relying on a simpler proxy heuristic like access frequency as a signal can lead to decisions that are detrimental to an organization. In this case, the data manager did not understand the importance of the tasks using the infrequently accessed table, and as a result negatively impacted an important task unintentionally.

In addition to relying on signals, data managers also rely on monitoring and observability tools [Uber Monitoring Data Quality], like Monte Carlo [Monte Carlo] or Bigeye [Bigeye], which capture metadata, trace data lineage, and handle anomaly detection. These tools are used in part to alert data managers when a potential problem arises, often in the context of data quality, such as a drift in a data distribution that could require manual preparation. However, these alerts, such as anomaly detection, are themselves still an imperfect signal for potential data quality issues; for instance, the distribution of a dataset may have genuinely drifted over time or the anomalies observed may be valid data points rather than being a sign of deteriorating quality. While these tools can direct data managers to potential problems and provide key information to help to investigate these problems, data managers must still do significant manual work to identify if a problem truly exists, develop a potential solution, and implement that solution.

Though tools exist to address specific data management problems, such as data lifecycle management, cost reduction, or data quality monitoring, these tools still require significant manual effort from data managers, and this manual effort cannot be easily amortized across different problems—e.g., AWS S3 Intelligent Tiering may be an effective solution for data lifecycle management but cannot easily identify what data to prioritize for preparation.

1.3 Gaining Utility from Data

We argue that across the many data management problems they face, users aim to balance the costs paid with the value extracted from using their data to complete tasks. Data utility, as defined in [Castro Fernandez, 2025], is a function of the value achieved from running tasks on data and the costs paid to achieve that value. This utility function may vary between organizations and individuals, as different entities may differ in how they aim to balance costs and value.

When a task executes, such as a SQL query used to populate a dashboard, an organization gains some insight from that dashboard, and those insights have some value to the organization. This value could be strictly monetary—such as the revenue achieved from a service, or the money saved from reducing costs—or could be represented in other units, such as customer satisfaction or human time expended.

The value of these insights depends on how well the task is written and the data it accesses. For instance, if a dashboard is used to decide where to place an advertisement to increase revenue, the quality of the insight depends on the dashboard being well-designed and on the quality of data used to build the dashboard, such as whether the data is relevant, clean, and up-to-date. Thus, some of the value of a task is determined by the data used to complete it.

There are costs incurred from gaining these insights. For instance, running a SQL query incurs monetary costs from the cloud database, storing data incurs storage costs, manual preparation incurs human time, and so on. Thus, the utility of a task, or of data, represents the net benefit of that task or data to an organization, factoring in both the value gained and the costs incurred.

Solutions that aim to improve data's utility by balancing value and costs can guide more efficient data management decisions, which we illustrate in the following example.

Example 1.2 *A team executes a nightly batch workload of many SQL queries to detect*

fraudulent transactions from the day's transaction logs. The engineering manager for this team observes an opportunity to improve the utility gained from this workload. The results of this workload are read manually by analysts the next morning at 10 am, but the workload typically completes around 5 am. This workload incurs significant query costs on cloud data warehouses. The engineering manager thus observes that reducing costs, even if doing so increases the runtime of the workload by a few hours, would not impact the value of the workload but could significantly reduce costs, dramatically improving utility. [BigQuery Reliability]

Example 1.2 illustrates how using utility can guide data managers to a more efficient data management solution, in this case finding an execution plan for a latency-insensitive batch workload. Beyond this example, we observe that utility captures a holistic picture of the benefit that data has contributed to all tasks across an organization, so utility is applicable in multiple different data management scenarios, such as data lifecycle or data preparation decisions. We now illustrate how the data lifecycle solution described in Example 1.1 can be improved with insights into the utility of data.

Example 1.1 (Continued) *In Example 1.1, an analyst placed key data that is accessed infrequently in Glacier storage, impacting the performance of training for a recommendation system. While costs decreased, the corresponding loss in value from that task outweighed the benefit from cost reduction. Instead, the analyst could place the data with the highest utility in the most expensive storage tier and the data with the lowest utility in Glacier storage. Utility would guide the analyst to keep that key data in faster storage, as the tasks accessing it are extremely valuable. This approach would migrate data used rarely by unimportant tasks to Glacier.*

Problem Statement. We now formally present the problem we address. We aim to identify a signal that guides data managers to more effective data management decisions across a range of problems, and we aim to implement tools that leverage this signal to

enable data managers to better serve their organizational goals.

1.4 Thesis Statement

Efficiently deciding how to configure and use cloud services to store, transform, and analyze data requires balancing the value gained from tasks and the costs of gaining that value. In this dissertation, we argue that data's utility, which is a function of costs and the value achieved from tasks, is a signal that can more directly guide efficient data management decisions. We present solutions that build cheaper execution plans across multiple cloud data warehouses for SQL workloads and which provide an indicator for the value of data to enable better data management decisions. These solutions provide concrete mechanisms for users to directly interact with and improve their data's utility.

1.5 Dissertation Contributions and Outline

This dissertation proposes solutions that rely on data utility to guide more efficient data management decisions, which in turn can improve the utility of an organization's data. The remainder of the dissertation is organized as follows:

- **Chapter 2: Data Management and Data Utility.** We first illustrate how data management is handled at organizations today, exploring the tools that exist, the techniques used by data managers, and what challenges data managers face. We then formally define data value and data utility and discuss how these definitions interact with data management.

- **Chapter 3: Saving Money for Analytical Workloads in the Cloud.** Chapter 3 exploits an opportunity to improve utility for users running SQL workloads in cloud data warehouses by utilizing multiple pricing models across different cloud data warehouses to find general cost savings opportunities. We propose two algorithms that

exploit monetary savings opportunities and build a prototype, ARACHNE, which implements these algorithms and achieves significant savings using multi-cloud execution plans. These solutions improve utility for users by reducing cloud costs under a runtime SLA to ensure that users still gain the same value from their tasks [Srivastava and Fernandez, 2024].

- **Chapter 4: Better Data Management Through an Indicator for the Value of Data.** In Chapter 4, we propose an indicator for the value of an organization's data, which data managers can leverage as a signal to guide better data management decisions compared to existing approaches. We present a relational interface through which data managers can access the indicator, summarize a model for the value and utility of data, and describe how we compute the indicator. We implement the value of data indicator on Databricks' Unity Catalog and Apache Spark and illustrate empirically that the value of data indicator can identify more efficient data management solutions while its implementation adds minimal overhead to query performance.

- **Chapter 5: Other Works Which Explore Utility in Data Management Problems.** In Chapter 5, we present two works that explore different data management problems and how organizations today approach these problems. We first present *Eudoxia* [Srivastava et al., 2025], a scheduling simulator that builds a simulator to evaluate how different scheduling algorithms would impact the cost and performance for a composable lakehouse running on cloud infrastructure. This simulator enables developers to identify what design choice would yield the best utility for their system and customers. We then present *Programmable Dataflows* [Xia et al., 2024], which defines a data sharing abstraction called the contract to facilitate data sharing. This abstraction enables users to weigh the risks and benefits of sharing data with other entities, to more effectively identify when data sharing would benefit them, and to find consensus between all relevant users. These works illustrate concrete data management

scenarios today and present bespoke solutions designed specifically for these scenarios.

- **Chapter 6: Conclusion.** Finally, in Chapter 6, we summarize the contributions of the dissertation and outline avenues for future work in building tools which enable users to observe, interact with, and improve the utility of their data.

CHAPTER 2

DATA MANAGEMENT AND DATA UTILITY

2.1 Data Management in the Cloud

Organizations gain insights from their data by executing tasks, such as code or manual processes, over their data. These tasks could be as simple as printing the schema of a relational table or as complex as a pipeline constructed of many subordinate tasks that run together to detect fraudulent transactions.

Organizations must decide how to store, transform, and process data in the cloud to gain these insights. These decisions include managing costs, allocating resources to computing tasks, and deciding what data to manually prepare, among many others. Efficient decisions to these problems balance the value of tasks run on data with the costs of doing so to best serve task-specific or organizational goals, such as latency constraints for a workload or broader goals to reduce cloud costs. For a small organization with only one major product, it may be easier for a single data manager to understand how each decision will impact organizational goals. For a larger organization, where each team may be unaware of other team's goals and what tasks those teams run, efficiently managing data is a complex problem for data managers to solve.

Data management problems are not specific to cloud environments, though specific features of the cloud present new dimensions to data management problems. In on-premise environments, where organizations purchase and operate infrastructure to store and process data, data managers face important and complex data management problems, such as deciding what data to store in limited memory, how to maintain high resource utilization, or how to maximize performance. However, we study data management problems in cloud environments because of the widespread use of the cloud today. In cloud environments, these data management problems have evolved because of the pay-per-usage pricing model offered

by cloud vendors in contrast to the up-front costs of on-premise data management. In these environments, data managers are interested in dimensions beyond just query performance or resource utilization, such as monetary costs, in addition to the quality of insights extracted from their data by tasks.

The types of problems encountered also depend on the workloads being run and the cloud services chosen by data managers. For instance, organizations running exclusively analytical SQL queries on static data may face problems primarily focused on configuring and optimizing the use of their cloud online analytical processing (OLAP) database. In contrast, organizations that execute expensive ETL pipelines and repeatedly train machine learning models may primarily face problems concerning data quality and freshness.

Thus, addressing these problems requires data managers to understand many pieces of information: how a potential solution impacts their data, how that will in turn impact tasks run on that data, and finally how that will impact the organizational goals. Gaining the information needed to precisely understand those relationships is complex—it requires mechanisms to understand what data is relevant for a given task, how a change to that data could impact the usefulness of a task, and how useful a task is in general to an organization. Gathering some of these pieces of information may need to occur during task execution. For instance, identifying the rows of a table relevant to a SQL query may need to be collected during query execution. For larger organizations running many different tasks across many different teams, gathering this context manually becomes increasingly complex, making it difficult for data managers to make efficient decisions. We illustrate this point in the following two examples.

Example 2.1: *A mid-sized online platform operates several independent services: a booking system, a search service, and a mobile app. Each system emits its own logs and transactional records, designed primarily for operational needs rather than analysis. Over time, these data sources are connected to a shared data platform. A small infrastructure team*

builds ingestion pipelines that extract data from production databases and logs, normalize schemas, and load them into a central data lake. However, the data is still not fully compatible because schemas evolve independently, timestamps are inconsistent, and identifiers do not always align. As a result, the platform team maintains a growing collection of manually written ingestion and cleaning jobs, each tailored to a specific source. Failures are common when upstream systems change, and fixes are often reactive.

Example 2.2: *Analysts and engineers at a mid-sized company begin creating derived datasets from their centralized data stored in a data lake, such as a daily bookings table, a user engagement metric table, or a denormalized search sessions table. These derived datasets emerge organically as different teams address different data management problems individually; for instance, two analysts may independently define slightly different versions of "active users," each backed by separate transformation pipelines. As derived datasets become widely used, and tasks or other data come to depend on these derived datasets, data managers must manage data stored on the shared centralized platform to ensure that these derived datasets are easily accessible and all ETL/ELT pipelines run efficiently and correctly; further, these managers may wish to identify and delete duplicate datasets to reduce costs and confusion and perform a range of other optimizations.*

In Example 2.1, the infrastructure team must make a range of decisions about how to best transform operational data and store the resulting derived data in a data lake to best serve the analytical tasks that will run on this transformed data. In Example 2.2, we see that data managers also manage data produced by many different teams and tasks, adding further complexity to the many dimensions they already have to consider.

Existing solutions, such as heuristic signals like access frequency, to these problems still require data managers to manually solve many problems, and these solutions may guide decisions which do not best meet organizational goals. As such, using the utility of data as a signal when approaching data management problems can yield more efficient solutions that

more directly benefit organizations.

2.2 A Model for Data Value and Utility

We now present a concrete model for reasoning about the value and utility of data as proposed by Castro Fernandez [Castro Fernandez, 2025]. Under this model, an **agent** is an entity with the capacity to represent, store, and process data. Agents execute **tasks**, which are some computation that runs over data. Tasks could be completed manually by humans or by computer programs and may be composed of many subordinate tasks. For instance, a task could require a SQL query run over relational data, and the results of that are then used by a Python notebook to train a machine learning model. Tasks are executed to fill an **information need** for the agent. In other words, every task is ultimately used to gain some insight or answer a question posed by an agent. Certain information needs are more important to agents than others. For instance, identifying a set of fraudulent transactions may be far more important to an organization than generating a report of social media interactions on advertising for a single month. As such, different tasks have different levels of importance to agents.

Under the model proposed by Castro Fernandez, there are two types of data value, **intrinsic value** and **instrumental value**. Data's intrinsic value is what it is worth in and of itself. This is the value of that data simply existing. Data's instrumental value is its value as it is used to solve tasks. In this dissertation, when we discuss value we are referring to instrumental value, as we are interested in how organizations gain information from their data and what data is most useful for tasks.

Under the instrumental view of data value, data accrues value when it is used by tasks to fill an information need. This data value is a function of its contribution to a task and the importance of that task to an agent, such as an engineer, team, or organization. However, there are costs incurred to compute tasks and prepare data for those tasks, such as query

costs in a data warehouse, storage costs, or the human time needed to manually clean and prepare data. Thus, in this model the **utility** of data is the net benefit that data provides to an organization. Utility is a function both of the value that data has contributed to a task and the costs incurred in extracting those benefits.

In the context of these definitions, an efficient data management decision improves data's utility. For some tasks, this may involve simply improving performance by reducing query latency or improving the throughput of a data pipeline. However, especially in cloud environments, this may involve reducing costs for tasks, especially those that do not have tight latency constraints like nightly batch workloads where analysts simply need the workload results by the next morning. In other cases, agents may primarily be interested in improving the quality of a task's output, such as improving the accuracy of a machine learning model by improving the quality of certain dimensions of training data or purchasing higher-quality data. Across these scenarios, however, more efficient data management decisions are those that balance costs and value to improve utility.

2.3 Conclusion

In this dissertation, we argue that approaches to data management problems which rely on utility can better serve organizations, and in the remainder of this dissertation, we present solutions that enable data managers to improve their data's utility by more efficiently solving data management problems. First, in Chapter 3, we present solutions that enable users to improve utility by finding general cost savings opportunities for analytical SQL workloads by executing these workloads across multiple cloud data warehouses employing different pricing models. In Chapter 4, we propose an indicator for the value of each piece of an organization's data and show how this indicator can be used as a signal to enable more efficient data management decisions. After presenting these two methods, we illustrate in Chapter 5 two concrete data management scenarios and build bespoke solutions to enable

data managers to make more informed decisions in these scenarios.

CHAPTER 3

SAVING MONEY FOR ANALYTICAL WORKLOADS IN THE CLOUD

3.1 Introduction

As analytic workloads migrate to cloud data warehouses, users care just as much about saving money as they do about optimizing workload runtime. Even modest savings on one workload become significant over time if the workload runs repeatedly. For example, saving $140 on an analytics workload that runs twice a day to update recommendations will save $100,000 a year, and organizations may have many such workloads that power different applications such as filling dashboards to visualize complex data patterns or managing ETL pipelines [What is OLAP?, Tejada, Snowflake ETL, Goff, Martinez, 2021, Google ETL, Dashboard].

While cloud providers offer many tools to tune database performance, there are no mechanisms to directly save money. Thus, users are increasingly employing setup-specific solutions to save costs such as working with consulting groups like McKinsey, the DuckBill group, or CloudZero [The Duckbill Group, CloudZero, McKinsey] to tune databases, turn off unused resources, and optimally utilize allocated resources.

In this paper, we identify opportunities to reduce the monetary cost of running analytical workloads in the cloud. The key insight is simple. Queries consume IO *and* CPU; but cloud databases' pricing models charge for IO *or* CPU time, to keep pricing simple. This opens an opportunity to save money by cleverly scheduling queries in a cloud database with a favorable pricing model.

The two most prominent pricing models for cloud databases are *pay-per-compute* and *pay-per-byte*. In a *pay-per-compute* model the user pays for computation time, e.g., in AWS Redshift [Armenatzoglou et al., 2022], while in *pay-per-byte* the user pays for the amount of data scanned irrespective of compute time, e.g., in Google BigQuery [BigQuery].

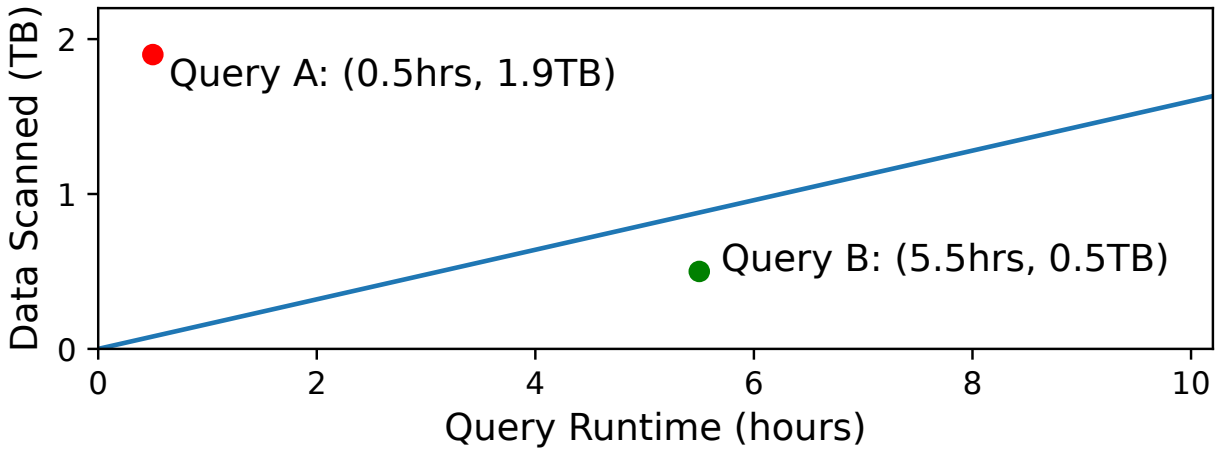


Figure 3.1: Size scanned (TB) vs. runtime (hours). Boundary for \$6.25/TB (pay-per-byte) vs. \$1/hour (pay-per-compute).

CPU-bound queries execute cheaper in pay-per-byte models, vice versa for IO-bound queries. Figure 3.1 plots query runtime (hours) on the x axis and the amount of data scanned (terabytes) on the y axis. We consider one cloud database charging \$6.25/TB (pay-per-byte) and one charging \$1/hour (pay-per-compute). Queries on the blue line cost the same in both databases, e.g. one that runs for 6.25 hours and scans 1TB. Query A runs fast but reads 1.9TB (e.g., a simple scan query). It costs less in the *pay-per-compute* database, so it is above the line. Query B scans 0.5TB but runs slower (e.g., window operations), so it costs less in the *pay-per-byte* database. Running each query in the pricing model most beneficial for it is cheaper than running both queries in a single cloud database.

All workloads have runtime constraints, even if they are loose. For example, a user with a nightly workload that usually finishes by 2 am may prefer to delay the completion time up to 8am if they can save money [BigQuery Reliability]. This distills the problem we explore in this paper: given an analytical workload (a set of queries and data) and a workload runtime constraint, we propose two algorithms to exploit money saving opportunities that arise from the observations above:

- **O1: Inter-Query Algorithm.** We propose an algorithm that takes a set of queries, data, a workload runtime constraint, and a set of cloud databases and identifies which

queries should execute in which databases to save money within the runtime constraint.

- **O2: Intra-Query Algorithm.** We propose an algorithm that, given a *single* query, a query runtime constraint, and a set of target cloud databases finds subqueries to execute in each cloud database to reduce query cost within the runtime constraint.

Where these complementary techniques are best applied is workload dependent, e.g., a workload with 3 expensive queries may benefit more from **O2** than **O1**, so we consider each in isolation. However, both **O1** and **O2** require moving data, ensuring cloud database SQL syntax compatibility, and, more importantly, managing the costs of data movement. To exploit **O1** and **O2** without modifying user setups, we implement middleware called Arachne between users and the cloud that takes a workload and runtime constraint, executes **O1**–**O2**, migrates data, and yields cheaper execution plans when these exist under the runtime constraint.

We use Arachne to study the impact of **O1**–**O2** on workload costs under runtime constraints. We build execution plans across Amazon Redshift (pay-per-compute) [Armenatzoglou et al., 2022] and Google BigQuery (pay-per-byte) [BigQuery] to evaluate Arachne, and we carefully configure these systems to the best of our ability [Scaer et al., 2020, Burman et al., 2022, BigQuery Optimizing, Sharma and Fu].

Our results show that there are massive opportunities for saving money. We achieve up to 57% savings (we run workloads for $104 while the original cost $243) with an inter-query plan across pricing models that has a nearly 10 hour slowdown. On another workload, an inter-query plan saves 55% with a 3-hour speedup. We also achieve up to 90% savings on a query via an intra-query plan.

We simulate varying cloud prices and study multi-cloud savings and runtime-cost tradeoffs as prices change. We see that savings are robust to price changes and that varying egress fees–what cloud vendors charge to move data out of a cloud–can aid or fully prevent data movement. In summary, the contributions of this paper are:

- We observe cost saving opportunities for analytical workloads by exploiting different cloud pricing models.
- We design two algorithms to exploit those opportunities.
- We implement the algorithms on a system called Arachne, to realize savings for real workloads.
- We evaluate Arachne (and the algorithms it implements) using prominent cloud databases and provide a simulation to shed light on the impact of market prices on saving opportunities.

Next, Section 3.2 presents background and the cost saving opportunity. Sections 3.3 and 3.4 present the inter- and intra-query algorithms to exploit **O1** and **O2**, and Section 3.5 presents Arachne. We evaluate savings opportunities in Section 3.6 and present related work in Section 3.7 and conclusions in Section 3.8.

3.2 Background and Opportunities

We provide background on analytical workloads in the cloud in Section 3.2.1, characterize the savings opportunity in Section 3.2.2, and present the problem statement in Section 3.2.3.

3.2.1 Executing Analytics Workloads in the Cloud

Cloud Data Warehouses and Pricing Models

We differentiate *infrastructure-as-a-service* (IaaS) from *platform-as-a-service* (PaaS) and within PaaS, we separate *pay-per-compute* from *pay-per-byte*.

IaaS vs PaaS. In IaaS, OLAP databases (e.g., Trino [Sethi et al., 2019] or Apache Hive [Thusoo et al., 2010b]) are manually deployed and maintained on virtual machines and billed by compute time. In contrast, PaaS (e.g., Google BigQuery [BigQuery], AWS Redshift [Armenatzoglou et al., 2022], Microsoft Azure Synapse [Azure Synapse], or Snowflake [Dageville

Table 3.1: Prices for pay-per-compute (PPC), pay-per-byte (PPB), storage, and egress. Systems in evaluation are bolded.

Pricing Model	Database	Cost
PPC	**Amazon Redshift–ra3.xlplus**	**$1.086/hr**
PPC	Amazon Redshift–ra3.4xlarge	$3.26/hr
PPC	Azure Synapse 100 DWU	$1.20/hr
PPC	Azure Synapse 500 DWU	$6/hr
PPC	Snowflake Small (AWS US-East)	$4/hr
PPC	n2-standard-32 VM (GCP)	$1.55/hr
PPC/PPB	Amazon Redshift Spectrum	RS + $5/TB
PPB	**Google BigQuery**	**$6.25/TB**
PPB	Amazon Athena	$5/TB
PPB	Azure Synapse Serverless	$5/TB

Cloud Vendor	Storage	Writes	Reads	Egress
GCP (us-east1)	$0.023/GB-mo	$0.05/10k ops	$0.004/10k ops	$120/TB
S3 (us-east)	$0.023/GB-mo	$0.05/10k ops	$0.004/10k ops	$90/TB
Azure	$0.018/GB-mo	$0.065/10k ops	$0.005/10k ops	$87/TB

et al., 2016]) deploy and maintain databases for users to directly use. PaaS charges more than IaaS for these services.

Pay-per-compute vs Pay-per-byte. Two of the most common PaaS pricing models are *pay-per-compute*, which charges for the amount and duration of computing resources, and *pay-per-byte*, which charges for the bytes read by a query regardless of runtime.

Breakdown of Cloud Costs

Loading data into a cloud database and executing a workload incurs the following costs:

- **Blob Storage cost**: Most cloud databases access data from blob storage (e.g., AWS S3), and storing data there has a cost (see Table 3.1). In S3 (us-east) it costs $23/month to store 1TB of data.
- **Read/Write cost**: Blob storage systems charge for API calls, e.g., read/write calls to store and retrieve data from blob storage. For instance, it costs $0.05 to perform 10,000

write operations in S3.

- **Loading cost**: While data is loaded into a machine, such a machine must be operative and thus consuming compute resources.
- **Egress cost**: Transferring data out of a cloud or between regions incurs per-byte charges. A 1TB transfer from GCP costs $120.
- **Query Processing cost**: Query execution can be billed per-byte or per-compute. BigQuery charges $6.25/TB scanned.

Running an analytics workload in the cloud involves four steps. In **Step 1**, data is collected from sources (e.g., on-premise repositories, sensors) and moved to the cloud, incurring data transfer and storage costs. In **Step 2**, data is loaded into a cloud database incurring read/write and loading costs. Moving data between clouds exacerbates these costs, so our algorithms account for this potential cost increase. In **Step 3**, users pay to execute queries against the data in the cloud database. Finally, in **Step 4** query results are returned to users to use in downstream tasks, e.g., reporting, filling dashboards [BigQuery analytics, Datta, 2022], potentially incurring egress costs. While all four steps incur costs, costs for **Steps 1 and 4** depend on the input and output data, which are mostly fixed for a given workload. We concentrate on **Steps 2 and 3** which involve cloud databases and dominate the total cost. We specifically focus on reducing the cost of **Step 3** and, to the extent that data movement is needed, **Step 2**.

3.2.2 Cost Saving Opportunity and Challenges

We now exploit the insight that migrating CPU- or IO-bound queries or subqueries to an analytical system with a *beneficial* pricing model presents an cost saving opportunity.

Given the size scanned by a query (S), query runtime (R), per-byte cost (α_S), and per-compute cost (α_R), we observe that $\alpha_S \times S = \alpha_R \times R \implies S = \frac{\alpha_R}{\alpha_S} \times R$. So a query that runs for R seconds costs the same in pay-per-compute as one that reads $\frac{\alpha_R}{\alpha_S} \times R$ bytes in

pay-per-byte. This equation represents the blue line in Figure 3.1 that delineates the most *beneficial* pricing model for a query.

Arachne needs query runtimes to exploit savings opportunities within runtime constraints. Unfortunately, there are few accurate approaches to estimating query runtime (R) Instead, Arachne collects query runtime via a *profiling* stage described in Section 3.5.2.

Adapting to Cloud Vendor Pricing. This analysis only requires query cost and runtime, so Arachne can support other pricing models. For tiered pricing models–such as egress where in AWS the first 10TB/month cost \$90/TB and the next 40TB/month cost \$85/TB–Arachne can track usage and adjust pricing constants accordingly.

Arachne must also track cloud price changes to keep its analysis accurate, as users do today. However, pricing changes happen rarely and are announced well in advance. Google announced a recent BigQuery price increase 3 months in advance [BigQuery Pricing Change] while Redshift pricing for current generation hardware has not changed for years.

3.2.3 Problem Statement and Approach

We now formally present the goal of this work. Given a workload of queries Q and tables T that execute on a source execution backend X_s under a runtime constraint, we consider a set of execution backends (each of which may offer a different pricing model) and find inter- and intra-query plans (**O1**–**O2**) that save money within the runtime constraint. Users choose which algorithms to run on their workload, as **O1** and **O2** do not need to be deployed together.

Approach. To solve the problem statement, we build a proof-of-concept, Arachne, which implements the inter- and intra-query algorithms and shows empirically that it is possible to save money on analytical workloads by scheduling queries and subqueries across clouds. Arachne does not require modifications to existing setups, e.g., where data is stored, and it handles all needed data movement. Arachne relies on an offline *profiling* stage to gather

query plans, runtimes, and costs to save money and meet runtime constraints.

We note that these algorithms can only honor runtime constraints up to the accuracy of the profiles, e.g., if cloud databases dramatically vary a query's runtime between iterations, the algorithms cannot compute accurate runtime or cost estimates for plans. We discuss this further with profiling in Section 3.5.2.

3.3 Inter-Query Execution Plan

We now present the inter-query algorithm to exploit **O1**. We discuss the setup (Section 3.3.1) and algorithm intuition (Section 3.3.2), present the algorithm (Section 23), and compare it to an optimal inter-query algorithm (Section 23) to understand its quality.

3.3.1 Algorithmic Setup and Goal

We consider an analytics workload with a set of tables T, queries Q, and a workload runtime constraint. We assume that initially a user employs a *source execution backend* X_s, paying storage costs for T in X_s, and paying execution costs for Q in X_s.

Given a second execution backend, X_d, the algorithm chooses a subset of queries $W \subseteq Q$ to execute in X_d such that the overall workload costs are reduced without breaking the runtime constraint. To run a query in X_d, all tables that the query scans must migrate from X_s to X_d, so the algorithm must account for migration costs. The algorithm requires the following inputs:

- The set of tables T and queries Q in the workload.
- $DEADLINE$ is an optional workload runtime constraint.
- The source X_s and destination X_d execution backends e.g., AWS Redshift or Google BigQuery.
- A set of *cloud prices* $\mathrm{P} = (p_{blob}, p_{read}, p_{write}, p_{sec}, p_{byte})$. p_{blob} per-byte for blob storage,

p_{read} read and p_{write} write cost, and execution backend costs p_{sec} for per-compute pricing models and p_{byte} for per-byte pricing models (example prices in Table 3.1)

- The egress cost e to move from X_s to X_d e.g., \$90/TB out of AWS.
- A function s which returns the size of a given table. This is measured via cloud storage APIs and is defined for the sake of notation.
- Functions C_{X_i} and R_{X_i} which take a query q and return the cost and runtime respectively of q in an execution backend X_i.

All the above inputs are easy to obtain except for C_{X_i} and R_{X_i}, which are obtained during a profiling stage, explained in Section 3.5. We now formalize the algorithm's goal.

Considering the Problem as a Bipartite Graph. We construct a bipartite graph $G = (T, Q, E)$, with tables T and queries Q. We draw an edge $(t \in T, q \in Q) \in E$ if query q scans base table t.

We next assign weights σ_q to each query $q \in Q$ and μ_t for each table $t \in T$. σ_q represents the *query savings* achieved by moving query q to the other execution backend, i.e.,

$$\sigma_q = C_{X_d}(q) - C_{X_s}(q) \tag{3.1}$$

μ_t represents the *migration costs* for a table, which is the cost of moving t from X_s to X_d, loading t into X_d, reading and writing t from blob storage, and temporarily storing t in blob storage. If each table requires K read/write operations, we can express μ_t as:

$$\mu_t = e \times s(t) + (p_{read} + p_{write}) \times (s(t)/K) + p_{blob} \times s(t) \tag{3.2}$$

In Figure 3.2 we show an example of this model with three tables $T = \{t_1, t_2, t_3\}$ and three queries $Q = \{q_1, q_2, q_3\}$. We draw edges to represent query dependencies e.g., q_3 scans tables t_2, t_3.

The algorithm's goal is to find a subset of tables that maximizes *query savings*. Con-

cretely, for $S \subseteq T$ let $N(S)$ be the set of queries scanning tables in S and let $N^{-1}(q \in Q)$ be the set of tables q scans. Our goal is to find S_θ in Equation 3.3. For example, in Figure 3.2 we move tables t_2, t_3 and queries q_2, q_3 to X_d, saving (3+4)-(2+4)=\$1.

$$S_\theta = \underset{S \subset T}{\text{argmax}} \sum_{q \in N(S)} \sigma_q - \sum_{s \in S} \mu_s \tag{3.3}$$

3.3.2 Inter-Query Algorithm

Now we provide the intuition for our greedy strategy to exploit **O1** (Section 3.3.2), and present the inter-query algorithm (Section 23). Finally, we assess the quality of the greedy algorithm by comparing it to an optimal (but much slower) min-cut algorithm (Section 23).

Intuition for the Greedy Strategy

At each iteration, the greedy algorithm computes the maximum savings achievable (upper bound) by moving queries depending on t to X_d for each table t. It removes the table with the least upper bound and records the cost and runtime of the resulting inter-query plan. When no tables remain, it chooses the cheapest plan with runtime under the runtime bound.

Concretely, we define $v_t = (\sum_{q \in N(t)} \sigma_q) - \mu_t$ as the sum of *query savings* for all queries that scan t minus the *migration cost* of t. As an upper bound on savings, if $v_t < 0$ it will *never* be beneficial to move t to the destination backend, so we remove nodes for t and all queries scanning t from the bipartite graph.

Analogously, we define a lower bound on savings generated from a single query, v_q, as *query savings* of q minus the *migration costs* of the tables that q requires, or $v_q = \sigma_q - \sum_{t \in N^{-1}(q)} \mu_t$. As a lower bound on possible savings for q, if $v_q > 0$ it is strictly beneficial to move q to X_d. To represent this we add q and $N^{-1}(q)$ to the final set of queries and tables to move to X_d, we remove the nodes representing q and all $t \in N^{-1}(q)$ from the bipartite graph, and remove all outbound edges from $N^{-1}(q)$. In Figure 3.2 we compute v_t

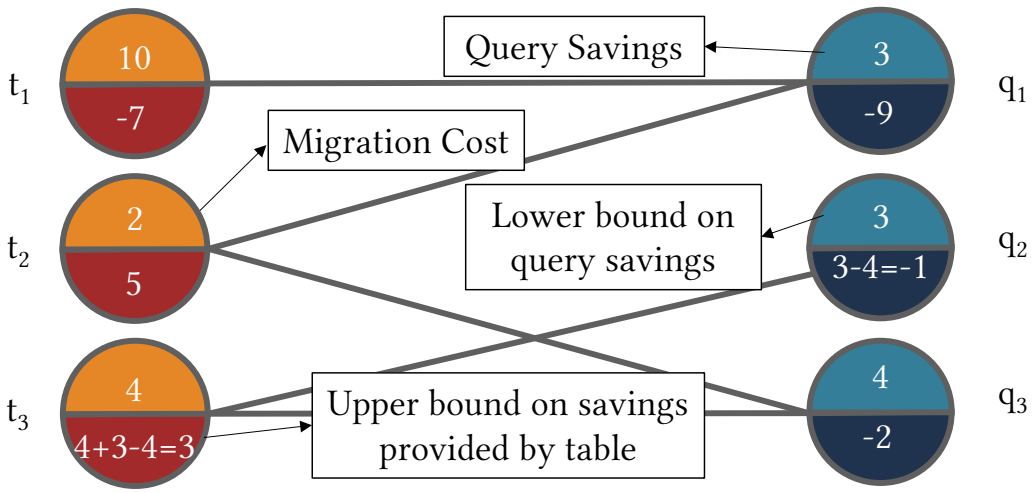


Figure 3.2: Bipartite model. Node top is query savings σ_q or migration cost μ_t. Node bottom is upper bound v_t or lower bound v_q. We show how this value is calculated for t_3 and q_2. and v_q and present them in the lower half of each node, e.g., v_{t_2} is the savings of q_1 and q_3 minus the cost of t_2, or $3 + 4 - 2 = 5$.

Algorithm Example. In Figure 3.3 the algorithm considers three plans for the given workload. The first plan (left) migrates all tables and queries, saving $65 but violating the runtime constraint, so runtime is colored red. Removing t_3 yields the second plan (middle), saving $40 and running in 2.5 hours, so both savings and runtime are colored green. Finally, removing t_1 yields the baseline (right) which saves no money. The second plan is chosen as it saves the most money under the runtime constraint, so it is colored green.

The Inter-Query Algorithm in Depth

The greedy algorithm, shown in Algorithm 1, first invokes ReducePlan (line 2), which computes v_t and v_q (lines 14–15), removes tables with $v_t < 0$ from consideration, and migrates queries with $v_q > 0$ to X_d (lines 17–22). This loop repeats until there are no more candidates or there are no tables left (line 16). ReducePlan returns the tables and queries to migrate and the remaining tables and queries to consider (line 23).

The algorithm then removes all outbound edges from the tables migrating to X_d (line 3) and proceeds to greedily remove tables from consideration. While there are still tables

1 **Function** `InterQuery` ($\{X_s, X_d\}$, *prices P, e, T, Q, s, C, R, DEADLINE*):
2 T', Q', T_f, Q_f = ReducePlan(X, P, e, T, Q, s, C) ;
3 Remove all outbound edges from T_f ;
4 **while** $T' \neq \emptyset$ **do**
5 For $t \in T'$, $v_t = (\sum_{q \in N(t)} \sigma_q) - \mu_t$;
6 $t' \in T'$ be the table with minimium $v_{t'}$;
7 $T' = T' - \{t'\}$; $Q' = \{q | N^{-1}(q) \subset T'\}$;
8 $T', Q', T'_f, Q'_f = ReducePlan(X, e, T', Q', s, f)$;
9 $T' = T' \cup T'_f$; $Q' = Q' \cup Q'_f$;
10 planCosts$[T', Q']$ = $\langle$cost(T', Q'), runtime$(T', Q')\rangle$;
11 Return the min cost plan in planCosts within $DEADLINE$;
12 **Function** `ReducePlan` (X, P, e, T', Q', s, C):
13 $Q' = \{q | \sigma_q > 0\}$; $T_f, Q_f = \{\}, \{\}$;
14 For $t \in T'$ $v_t = (\sum_{q \in N(t)} \sigma_q) - \mu_t$;
15 For $q \in Q'$ $v_q = \sigma_q - \sum_{t \in N^{-1}(q)} \mu_t$;
16 **while** $T' \neq \emptyset \wedge \exists v_t < 0 \wedge \exists v_q > 0$ **do**
17 $U = \{t | v_t < 0\}$; $V = \{q | v_q > 0\}$;
18 $T' = T' - U$; $Q' = Q' - \{q | q \in N(t) \forall t \in U\}$;
19 $T' = T' - \{t | t \in N^{-1}(q) \forall q \in V\}$; $Q' = Q' - V$;
20 $T_f = T_f \cup \{t | t \in N^{-1}(q) \forall q \in V\}$; $Q_f = Q_f \cup V$;
21 For $t \in T'$ $v_t = (\sum_{q \in N(t)} \sigma_q) - \mu_t$;
22 For $q \in Q'$ $v_q = \sigma_q - \sum_{t \in N^{-1}(q)} \mu_t$;
23 return T', Q', T_f, Q_f;

Algorithm 1: Inter-query greedy algorithm

(line 4), the algorithm computes v_t (line 5), assigns t with minimal v_t to remain in X_s, and removes nodes in $N(t)$ (line 6–7). The algorithm calls ReducePlan again to prune away tables with $v_t < 0$ and identify queries with positive lower bounds (line 8). The algorithm records the cost and runtime of the current plan using query costs and runtimes C and R, cloud prices P, and table sizes s (line 10).

Finally, the algorithm chooses the cheapest plan in *planCosts* with runtime less than $DEADLINE$ (line 11). The baseline plan– migrating no tables or queries–will be cached in *planCosts* and will be chosen if no cheaper plans exist under the runtime constraint.

Optimal Algorithm and Complexity Analysis

To evaluate the greedy algorithm we: i) present an optimal min-cut based inter-query algorithm; ii) compare its runtime complexity to the greedy approach; iii) and show the greedy algorithm's accuracy in practice.

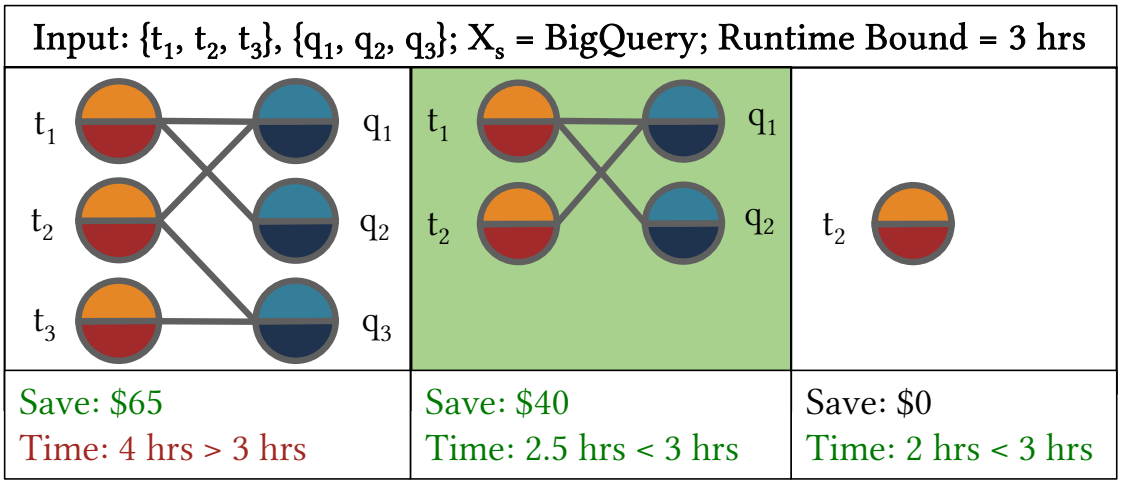


Figure 3.3: The plans considered with a 3 hour runtime bound. Plan B saves the most money within 3 hours so it is chosen.

Optimal Solution. As in [Heller et al., 2021], we build a capacity function c and a bipartite graph G with tables on the left and queries on the right. Edges with infinite capacity connect tables and queries by query dependencies. We add a source node a and draw edges from a to every table t_i; $c((a, t_i)) = \mu_{t_i}$. We add a sink node b and draw edges from every query q_j to b; $c((q_j, b)) = \sigma_{q_j}$. The algorithm finds the min-cut (A, B) and migrates the queries and tables in B to X_d.

Complexity Analysis. Using a min-cut algorithm [Edmonds and Karp, 1972, Dinitz, 2006], the optimal algorithm has complexity $O(|V|^2|E|)$. For the greedy approach, (note: $|V| = |T| + |Q|$), computing v_t or v_q is $O(|V|)$ with at worst $|T|$ iterations, yielding a worst-case complexity of $O(|T||V|)$. The optimal algorithm is both an order of magnitude less efficient and depends on the number of relationships between queries and tables. The greedy algorithm is independent of query complexity.

Practical Significance. This difference in complexity between the optimal and greedy algorithms has practical significance when the number of queries and tables is large. For example, when using the TPC-DS workload as input (24 tables and 100 queries), the difference is insignificant, with both algorithms running in less than 0.3 seconds. With 1000 queries and 100 tables, the optimal runtime jumps to 3.4 seconds, while the greedy remains under

0.3 seconds. With 2500 queries and 400 tables, the optimal algorithm takes 2.1 minutes while the greedy takes 1.2 seconds.

Greedy Algorithm Accuracy. To evaluate accuracy, we use 72 workloads at 1TB and 2TB, both with and without IaaS, and using both internal and external BigQuery storage, producing 576 workloads. We run the greedy and optimal algorithms on each workload. Our greedy strategy finds the optimal solution for all workloads.

3.4 Intra-Query Execution Plan

We now present the intra-query algorithm to exploit **O2**. We first present the setup (Section 3.4.1) and how to identify where to make cuts (Section 3.4.2) before presenting the algorithm (Section 3.4.3).

3.4.1 Algorithmic Setup and Goal

The setup is similar to that in Section 3.3.1, but this algorithm takes a single query q and a runtime constraint for q and finds a subquery that saves money when migrated from X_s to X_d after accounting for migration costs while honoring the runtime constraint.

Formalization and Goal. A query plan is a directed acyclic graph where leaves represent base tables and edges represent data flow from upstream tables to downstream operators. Removing all outbound edges from a node partitions the graph into two disjoint subgraphs; we call this process making a *cut* in the query plan at a node. The goal of the intra-query algorithm is to find a *profitable* cut of q so that one subquery executes in X_s and the other in X_d so that the total query cost, including migration cost, is lower than running the entire query in X_s while adhering to the runtime constraint.

3.4.2 Identifying Profitable Cuts

We now present the insight we use to identify profitable cuts. Let $T = (V, E)$ be a query plan. Let p_{byte}, p_{sec}, and e be per-byte, per-compute, and egress prices. Let migration cost μ_t be as in Equation 3.2 and let μ_v for $v \in V$ be the migration cost of the data output from v. Let s be the table size and $rs(v)$ be the row size for v. Finally, when a cut is made at a node v, the subgraph **u**pstream of v is $S_{\mathbf{u}}(v)$–including v and all base tables that flow into v–and the subgraph **d**ownstream of v is $S_{\mathbf{d}}(v)$. We now show some key definitions.

- $f_w(v)$ returns the output cardinality of $v \in V$
- $f_r(v)$ returns the runtime of $S_u(v)$
- $DEADLINE$ is an optional runtime constraint for q.
- $\mathcal{L}(v)$: the base tables in the downstream subquery $S_d(v)$.
- $C_s(v) = \alpha_s \sum_{u \in \mathcal{L}(v)} f_w(u) rs(u)$: the cost of $S_d(v)$ per-byte.
- $C_r(v) = p_{sec} f_r(v)$: the cost of $S_u(v)$ per-compute.
- $C_m(v) = \mu_v + \sum_{t \in \mathcal{L}(v)} \mu_t$: the cost to migrate all necessary data, including the output of v, to X_d.

These values require query costs C_{X_i} and runtime R_{X_i} in all execution backends. The algorithm's goal is to find $v \in V$ such that:

$$C_r(v) + C_m(v) + C_s(v) < C_{X_s}(q) \tag{3.4}$$

This finds the cheapest plan, represented on the left, that costs less than simply executing the query in X_s.

Insight. Naively, we could find the optimal execution plan by making a cut at every operator and executing each resulting plan. This approach requires we pay query processing and migration costs for each possible plan, and the number of possible plans grows with the size

of the query. Our goal is to find cheaper execution plans while minimizing the incurred overhead costs.

To achieve this goal, we assign a value to each operator in q corresponding to the *savings opportunity* of making a cut at that operator. Using the inputs to the algorithm, we reorder Equation 3.4 and compute the maximum savings achievable by an intra-query plan. We consider those operators with positive savings opportunity and use this value to guide what candidate operators we evaluate.

Calculating Savings Opportunity. The *savings opportunity* o_v is the right hand side of $p_{sec}f_r(v) < C_{X_s}(q) - (C_m(v) + C_s(v))$, derived from Equation 3.4, which we compute using C_{X_i} and f_w. We draw two conclusions. First, if $o_v < 0$, the plan produced from a cut at v will cost more than the baseline. Second, the only way to determine if a plan will cost less is to pay to compute f_r, so the algorithm aims to reduce the number of times it computes f_r.

Function `IntraQuery` ($T = (V, E)$, X, P, e, f_w, C, R, $DEADLINE$)**:**
  for $u \in V$ **do**
    $o_u = C_{X_s}(q) - (C_m(u) + C_s(u))$;
  candidates $= \{v \in V | o_v > 0\}$;
  while *candidates* $\neq \emptyset$ *or number of iters* $< K$ **do**
    Pick u from *candidates* such that o_v is largest ;
    Compute $f_r(u)$;
    Compute $a_u = o_u - p_{sec}f_r(u)$;
    for $v \neq u \in$ *candidates* **do**
      If $o_v < a_u$ remove v from *candidates* ;
    for $v \in$ *candidates; v downstream of u* **do**
      $o_v = o_v - p_{sec}f_r(u)$;
      If $o_v < 0$ remove v from *candidates* ;
  Return u with maximum $a_u > 0$ under $DEADLINE$ or baseline;

Algorithm 2: Intra-query algorithm

Updating Opportunities. Measuring f_r allows us to update the opportunities for other nodes. For example, if a node u_1 sits downstream of another node u_2, $f_r(u_1) \geq f_r(u_2)$, so if we compute $f_r(u_2)$, o_{u_1} must decrease by $p_{sec}f_r(u_2)$ since it must pay at least that much in runtime cost. If we measure the savings for a plan a_u, we can remove all candidates with $o_{u_1} < a_{u_2}$ because a cut at u_1 cannot produce a cheaper execution plan. We can then remove multiple candidates per iteration, reducing the number of invocations to f_r.

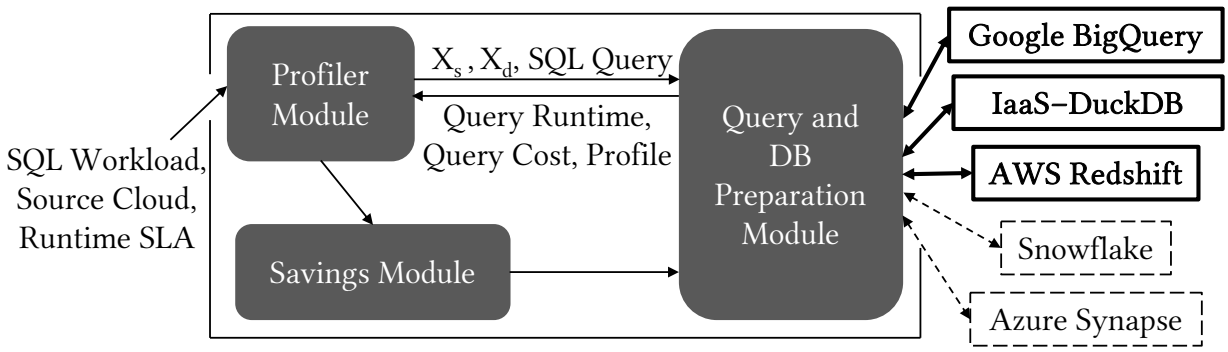


Figure 3.4: Arachne architecture and execution backends.

3.4.3 Intra-Query Algorithm

The intra-query algorithm, in Algorithm 2, computes the opportunity o_u for every node (lines 2-3) and picks candidates with $o_u > 0$ (line 4). It iterates over the candidates from largest to smallest o_u (line 6). Computing f_r for all candidates may be expensive, so iterating from largest to smallest opportunity ensures that the potential savings lost by not iterating over all candidates are minimized.

For each candidate u, the algorithm computes $f_r(u)$, the real savings, a_u, (lines 7–8) and updates the opportunity for other candidates (lines 10–16). It repeats this until all candidates have been checked or after K iterations (line 5). The algorithm chooses the cut that yields the most savings within the runtime constraint or the baseline if no such cut exists (line 14).

Complexity and Discussion of Optimality. The algorithm removes at least one candidate per iteration, so the worst-case complexity is $O(|V|)$. The algorithm by default considers all cuts in a query plan that could yield savings and chooses the optimal cut. The algorithm only parses a single query plan, but it will never choose an execution plan more expensive than the baseline.

3.5 Arachne Overview

We implement Arachne, including the inter- and intra-query algorithms, in 8k lines of Python and Java. We present Arachne's architecture in Section 3.5.1, the *profiler* in Section 3.5.2, and other cost-relevant implementation decisions in Section 3.5.3.

3.5.1 Overview

Figure 3.4 shows Arachne's architecture. Users first execute INITIALIZE, which submits an unmodified SQL workload, stored as individual SQL files, the source execution backend where data starts, and the optional runtime constraint. Arachne implements the inter- and intra-query algorithms in the **savings module**. The **profiler module** gathers the query costs and query runtime inputs (C_{X_i} and R_{X_i} from Sections 3.3–3.4) so the savings module can save money and meet the runtime constraint. The **preparation module** prepares queries for execution in target backends. It 1) ensures that SQL queries are compatible with the syntax requirements of execution backends; 2) submits queries for execution; 3) orchestrates materializing data into portable, open-format Parquet files; and 4) migrates those Parquet files between execution backends when needed.

Execution Backends. Arachne supports two PaaS backends–AWS Redshift and Google BigQuery–so we can study per-compute and per-byte pricing models. It can also deploy DuckDB on IaaS. These backends are bolded in Figure 3.4. Arachne can be easily extended to support new backends, such as those in dashed lines in Figure 3.4.

Minimizing Infrastructure Changes. Arachne is designed to be compatible with existing ETL pipelines (and downstream BI tools and dashboards), so it moves data between source backends *at runtime* whenever this saves money and does so transparently to the end user. Consequently, Arachne moves data every time a workload executes (incurring costs) but does not need to handle data inconsistencies that may arise if, for example, the data were

replicated in several backend systems. Exploring tradeoffs of the spectrum of solutions (keep ETLs intact and pay repeated migration costs or change ETLs, duplicate data, and incur double storage costs) is beyond the scope of this paper. Despite minimizing infrastructure changes, Arachne still requires configuration changes, e.g., to access cloud accounts. We believe this is not an important roadblock to deployment, as such credentials are frequently shared with other tools. These changes add one-time costs; in exchange, Arachne achieves large recurring savings (see Section 3.6).

Compliance. At a high level, Arachne is best seen as an alternative interface to the source backend. When data (or queries) cannot move (or execute) between vendors, e.g., for compliance requirements preventing data migration to a geographical area, such data can be excluded from Arachne and run using the usual interface.

Implementation. Arachne uses Apache Calcite [Begoli et al., 2018] to convert between SQL and query plans and perform query optimization. Arachne uses Apache Arrow [Apache Arrow] to sample tables; the Redshift-Data API to use Redshift clusters and S3; and the Google Cloud client for Storage and BigQuery. It migrates all data at runtime.

3.5.2 Profiler Module

The profiler module provides the inter- and intra-query algorithms with accurate runtime, cost, and cardinality information for each query. We consider two approaches: prediction and profiling.

Why We Do Not Use Prediction in Arachne. Existing approaches to estimate operator cardinality f_w, query cost C_{X_i}, and query runtime R_{X_i} are noisy. Cardinality estimates from query optimizers remain inaccurate [Lohman, 2014, Yang et al., 2019, Perron et al., 2019]. The "cost" produced by query optimizers for a given query plan has only a relative meaning to the cost associated to other plans rather than to absolute monetary or runtime cost, making query optimizers' cost a poor proxy for estimating runtime [Leis et al., 2015, Wu

et al., 2013]. Last, there are few approaches for estimating query runtime given a query plan, and existing ones often underperform, as we show in Section 3.6.6. However, we anticipate that continued advances in this area could replace the profiling approach we now explain.

The Profiling Approach. The profiler executes each query in each execution backend, obtaining its runtime and cost. It also gathers the output cardinality for each query operator via query profiling in DuckDB, and provides the inputs to the inter- and intra-query algorithms (C_{X_i}, R_{X_i}, and f_w as defined in Sections 3.3–3.4). Profiling is more accurate and more costly than prediction. Profiling cost is amortized if the profiled queries execute several times without major runtime changes. This is the case for most periodic workloads [Tan et al., 2019, AWS Batch Processing, BigQuery Reliability], which happen to be the most relevant for saving money. Furthermore, stale profiles can still expose savings opportunities since small errors in costs do not greatly alter what queries migrate in an inter-query plan, as we illustrate in Section 3.6.6.

Profiling Over Samples. It is possible to reduce profiling costs by measuring runtime, cost, and operator cardinality on a workload sample and then extrapolating to the original workload size without introducing significant error. It is difficult to extrapolate query runtime from samples, which is related to the difficulty of join sampling. If the output of a join over data samples is not a sample of the true join result, we cannot accurately extrapolate runtime for the full join [Chaudhuri et al., 1999]. However, profiling costs are largely in *pay-per-byte* pricing models and depend only on the data size. We show empirically in Section 3.6.6 that the profiling cost is quickly compensated by workload savings, often in less than 5 executions of the cheaper execution plan when profiling over the entire dataset and in 1–2 iterations when using a sample. For periodic workloads, we can collect profiles during iterations of the workload to reduce the net profiling costs. We see then that profiling costs are quickly compensated by workload savings.

Assigning Operator Cardinalities. To provide operator cardinality f_w (Section 3.4.2),

Arachne profiles queries in DuckDB as cardinality is independent of execution backend. However, DuckDB's physical plan may not match Arachne's internal query plan, so Arachne cannot directly match operator profiles from DuckDB onto its query plan. We observed that DuckDB does not re-order operators between SQL subqueries, so Arachne writes its query plan into SQL, with all operator trees as nested subqueries, so DuckDB's physical plan exactly matches Arachne's and Arachne can assign cardinalities to operators in its internal query plan.

3.5.3 Cost-Relevant Implementation Details

Data Transfer Between Clouds. Cloud vendors provide easy-to-use tools (e.g., AWS DataSync) to transfer data into their clouds. Beyond the egress cost of doing so, users have to pay for these tools too. Arachne implements a simple cloud transfer tool using blob storage APIs. Our tool on a GCP n2-standard-32 VM transferred a 615GB dataset for $0.58; that transfer in AWS DataSync costs $7.69.

SQL Compatibility. Arachne builds and applies text rules to ensure Arachne-produced SQL is compatible with different backend dialect rules, e.g., BigQuery requires column names to contain only alphanumerics or underscores.

Calcite Query Operators. Arachne implements its own physical node subclass and uses Calcite libraries to perform heuristic optimizations like predicate pushdown, make cuts in query plans, and assign cardinalities collected during profiling to query operators.

3.6 Evaluation

In this section, we answer the following research questions:

- **RQ1:** Does the inter-query algorithm save money? (**O1**)
- **RQ2:** Does the intra-query algorithm save money? (**O2**)

- **RQ3:** How does pricing (chosen by cloud vendors) affect inter-query savings and the runtime-cost tradeoff?
- **RQ4:** Profiler Microbenchmarks: How does sampling impact profiling costs and accuracy and how quickly are they compensated? Does using stale profiles diminish savings? How are savings impacted by noisy runtime estimates versus profiles?

Because Arachne can deploy DuckDB on IaaS, we also evaluate how utilizing cheaper IaaS impacts inter-query savings.

3.6.1 Workloads

Resource Balance Workloads. The specific *balance* of CPU- and IO-queries in a workload will impact savings opportunities. We use the well-known TPC-DS [Nambiar and Poess, 2006] benchmark to create three workloads each with a different balance of CPU- and IO-bound queries to explore the design space (in the original TPC-DS benchmark nearly all of the 99 queries are IO-bound). We adapt queries from LDBC, a well-known business intelligence benchmark [Szárnyas et al., 2022], to work on data from TPC-DS: we create queries to find customers related to each other by purchase history and queries to find connected components of customers for recommendation algorithms. We combine the authored CPU- and some existing IO-bound queries to create three workloads over 17 tables with different characteristics:

- W-CPU: 46 queries, about 40% of which are CPU-bound.
- W-Mixed: 49 queries, about 30% of which are CPU-bound.
- W-IO: 46 queries, about 20% of which are CPU-bound.

While there are more IO-bound queries in each workload, the CPU-bound queries consume a large amount of CPU, so overall resource consumption for each workload reflects the

workload's name. We make all workloads publicly available [1] for reproducibility and because they may be of independent interest to others.

Read-Heavy Workloads. We also explore skewed workloads. While nearly all TPC-DS queries are IO-bound with runtime dominated by table *reads*, these queries differ in runtime and complexity. To explore savings opportunities on a range of IO-bound workloads we create 24 workloads called the Read-Heavy workloads from TPC-DS, which contains 24 tables and 99 queries, by removing one table from the TPC-DS dataset. This creates a 23-table dataset with a subset of the original 99 queries, on average about 80 queries. Each workload is named by the alphabetical order of the table that was removed to generate it, e.g., the workload created by removing the first table alphabetically, *call_ center*, will be named **Read-Heavy 0**.

LDBC. Additionally, we use queries written on the LDBC Social Network Benchmark-Business Intelligence (SNB-BI) dataset [Szárnyas et al., 2022].

For **RQ1** and **RQ3** we use the Resource Balance and Read-Heavy Workloads. For **RQ2**, we use TPC-DS queries and author queries on TPC-DS and LDBC. We explain these in detail in that section.

3.6.2 Experimental Setup

PaaS Execution Backends. We use Google BigQuery (*pay-per-byte*) and AWS Redshift (*pay-per-compute*) as popular representatives of each pricing model. In Redshift cluster size impacts cost and performance, so we explore the G→A1, G→A4, G→A8 setups where data starts in BigQuery (G) and we consider migrating to a 1-, 4-, or 8-node ra3.xlplus Redshift cluster respectively (→A1, →A4, or →A8; the arrow indicates the migration direction). We explore the A4→G setup where data starts in a 4-node ra3.xlplus Redshift cluster and could migrate to BigQuery. We optimize our Redshift and BigQuery setup per docs and best

1. https://github.com/tapansriv/resource-balance-workloads

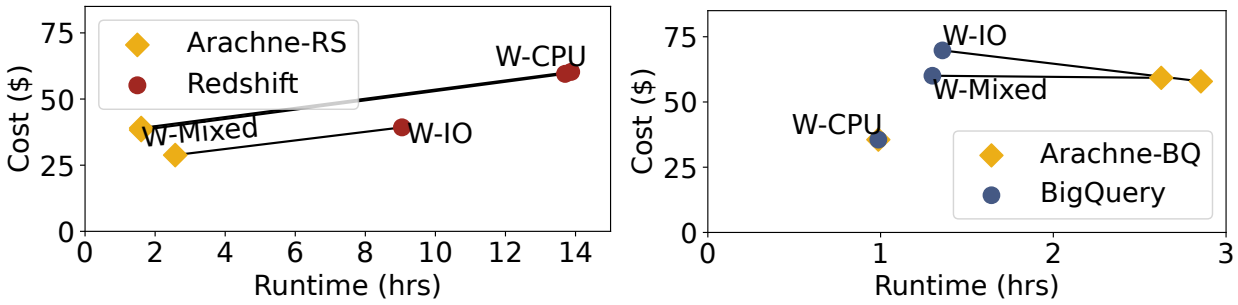
practices [Scaer et al., 2020, Burman et al., 2022, BigQuery Optimizing, Sharma and Fu].

Data Format and Storage. We store intermediate data in Parquet files [Apache Parquet] with Snappy compression [Snappy]. All cloud databases we use are compatible with open data formats [Databricks Data Lakes, Prasad, 2022]. We create external tables in BigQuery pointing to the Parquet files in blob storage. We also consider data loaded into BigQuery. Redshift loads Parquet files from S3. The compression of data saves migration costs, and all in-flight compression occurs during materialization from pay-per-compute databases and is billed as runtime cost.

Where Data Is Initially Stored. For the Resource Balance Workloads, we consider G→A4 and A4→G. With current prices, Redshift is significantly cheaper than BigQuery for IO-skewed workloads and queries. As such, we consider only G→A1, G→A4, G→A8 for the Read-Heavy workloads to evaluate **O1** and only consider DuckDB and BigQuery on data stored in GCP to evaluate **O2**, as there are few savings opportunities if data starts in Redshift.

How Workloads Are Executed. Batch, analytic workloads like those in this evaluation are often executed serially but can also be submitted all at once to a system [IBM Serial Batch, AWS Batch]. For pay-per-byte systems like BigQuery, this has no impact on workload cost, only runtime. For pay-per-compute systems like Redshift, this also impacts costs. Redshift's REST API, the Redshift Data API [BatchExecuteStatement] offers BatchExecuteStatement which executes a list of SQL statements one at a time in a single transaction.

Metrics. We measure monetary cost in US dollars and runtime in time units. We account for all applicable cloud costs (see Section 3.2.1) and use cloud prices as of February'24 in Table 3.1. We validate our results by checking the breakdown of charges in our account from cloud vendors. We create VMs in the same cloud region as blob storage buckets to avoid regional transfer costs and utilize data compression to reduce migration costs.



(a) 1TB A4→G

(b) 1TB G→A4

Figure 3.5: (1TB A4→G, G→A4) Cost (USD) vs runtime (hours) for W-CPU, W-IO, and MIXED workload compositions

Runtime Constraint. We assume that there are no runtime constraints and concentrate on exploring cost savings without imposing arbitrary runtime constraints that do not add any additional insight.

3.6.3 RQ1. Inter-Query Processing

We now explore **O1**. First, we evaluate the cost opportunities of the inter-query algorithm across the three Resource Balance Workloads (Section 3.6.3) and the IO-skewed Read-Heavy workloads (Section 3.6.3). Then, we leverage that Arachne can deploy DuckDB on IaaS to evaluate how that impacts savings (Section 3.6.3).

Resource Balance Workloads

We run the inter-query algorithm on W-CPU, W-IO, and W-MIXED in G→A4 and A4→G and evaluate multi-cloud savings. First, we provide an overview of the results before presenting an in-depth breakdown of costs.

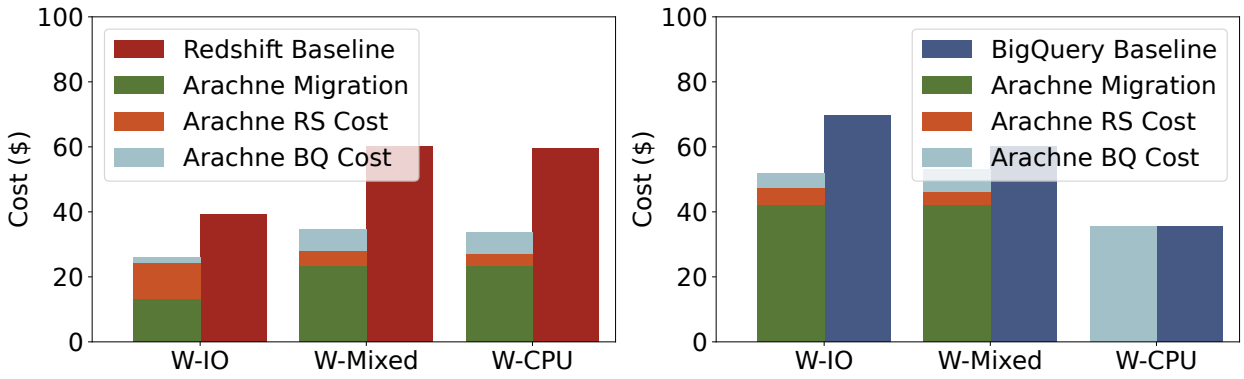
Resource Balance Workload Overview. In Figure 3.5 we compare the runtime (hours) on the x-axis and cost (USD) on the y-axis of Arachne's execution plan to a baseline that executes the workload in the starting backend. In Figure 3.5a, data starts in Redshift. There are three red dots, one for each workload, which represent the cost and runtime of executing

that workload in Redshift. The yellow dots represent the runtime and cost of executing that workload with Arachne. A line connects each yellow dot to its corresponding red dot according to the workload. In Figure 3.5b, data starts in BigQuery (blue dots) instead of Redshift.

In 5 out of the 6 workloads Arachne finds cheaper plans: all lines decrease from the starting cloud baseline unless Arachne has chosen the baseline execution plan, and the degree of its reduction corresponds to monetary savings. For A4→G in Figure 3.5a, Arachne chooses multi-cloud plans for all three workloads as there are enough CPU-bound queries to make migration cost-effective. Arachne saves 27% on W-IO and 35% on W-Mixed and W-CPU over the Redshift baseline. Arachne saves less money for W-IO because there are more IO-bound queries that favor Redshift's per-second pricing model. In G→A4 in Figure 3.5b, Arachne executes W-CPU entirely in BigQuery, so those two dots are directly on top of each other. Since W-Mixed and W-IO contain more IO-bound queries, Arachne saves 1.35% on W-Mixed and 17% on W-IO by migrating IO-bound queries to Redshift. Because W-IO has *more* IO-bound queries, the margin of savings is larger for W-IO than for W-Mixed.

Both W-Mixed and W-CPU include a very CPU-bound query, which groups customers by spending history for recommendations. It runs in 6 hours and costs $25.84 in Redshift, while in BigQuery it runs in 3.5 minutes and costs $1. Other CPU-bound queries are similarly faster *and* cheaper in BigQuery. Consequently, in the A4→G setup in Figure 3.5a, baseline costs for W-Mixed and W-CPU are similar, and Arachne chooses similar multi-cloud plans for both workloads that are faster *and* cheaper than the Redshift baseline.

Resource Balance Workload Cost Breakdown. We divide costs for Arachne into (1) migration costs, (2) the cost of queries which migrated, and (3) the cost of queries which remained in the source backend. In Figure 3.6a we breakdown costs for plans shown in Figure 3.5a and in Figure 3.6b we breakdown costs for plans in Figure 3.5b. Diagonal lines are drawn through bars showing Arachne's plans.



(a) A4→G: Data Starts in Redshift

(b) G→A4: Data Starts in BigQuery

Figure 3.6: (1TB) Arachne migration costs, cost of queries moved, and cost of queries remaining versus baseline.

Table 3.2: Inter-query plan-type by setup at 1TB and 2TB.

Setup	Arachne	Multi	GCP	AWS	Total
1TB, G→A1, →A4, →A8	**23**	6	1	17	24
2TB, G→A1	**22**	4	1	19	24
2TB, G→A4	**22**	3	2	19	24
2TB, G→A8	**22**	4	2	18	24

In Figure 3.6b, W-CPU Arachne's bar is equal to the BigQuery baseline. If the source pricing model is already favorable for a workload, Arachne will keep all data in the starting cloud. For all other plans in Figure 3.6, multi-cloud savings are driven by the significantly lower execution cost for queries that migrated to a favorable pricing model versus their baseline cost. In Figure 3.6, the difference in query execution costs is the baseline (right) bar minus the blue portion of Arachne's bar, which represents the cost of queries remaining, presenting an enormous savings opportunity. Exorbitant migration costs make up the majority of Arachne's costs for multi-cloud plans. Egress is 90% of all migration costs–note that egress out of GCP is $120/TB while egress out of AWS is $90/TB–loading data into Redshift is 5–8% of migration costs, and the rest is the cost of blob storage and data retrieval.

Read-Heavy Workloads

Does Arachne save money on skewed workloads? We first summarize the results before zooming in on a few interesting workloads to understand the cost and runtime.

Read-Heavy Overview. Table 3.2 presents the outcomes in setups G→A1, G→A4, G→A8. The ARACHNE column indicates workloads where Arachne saves money over the BigQuery baseline. Across 144 workloads–48 workloads in 3 setups–only 6/144 remain in BigQuery where the Arachne saves no money. For workloads with cheaper plans, in MULTI plans some tables migrate to Redshift, and in AWS plans *all* tables migrate to Redshift. Since Read-Heavy workloads are IO-bound, the savings of moving queries to Redshift compensate for migration costs. That even 6 of these IO-skewed workloads remain in BigQuery demonstrates that egress costs are a massive barrier to data movement.

At 2TB, we see that from G→A1 to G→A4 one MULTI plan flips to GCP and from G→A4 to G→A8 one AWS plan flips to MULTI. The 4× cost of G→A4 doesn't reduce runtime by 4×, decreasing savings. If clusters are overprovisioned or underutilized, query savings will diminish even with highly IO-bound workloads.

Arachne saves up to 57.4% on a single workload.Of the 29 multi-cloud plans, most achieved 35%–50% savings. 9 saved 2%-8%, while 1 saved less than 1%. These plans save money by migrating queries to their most beneficial pricing model.

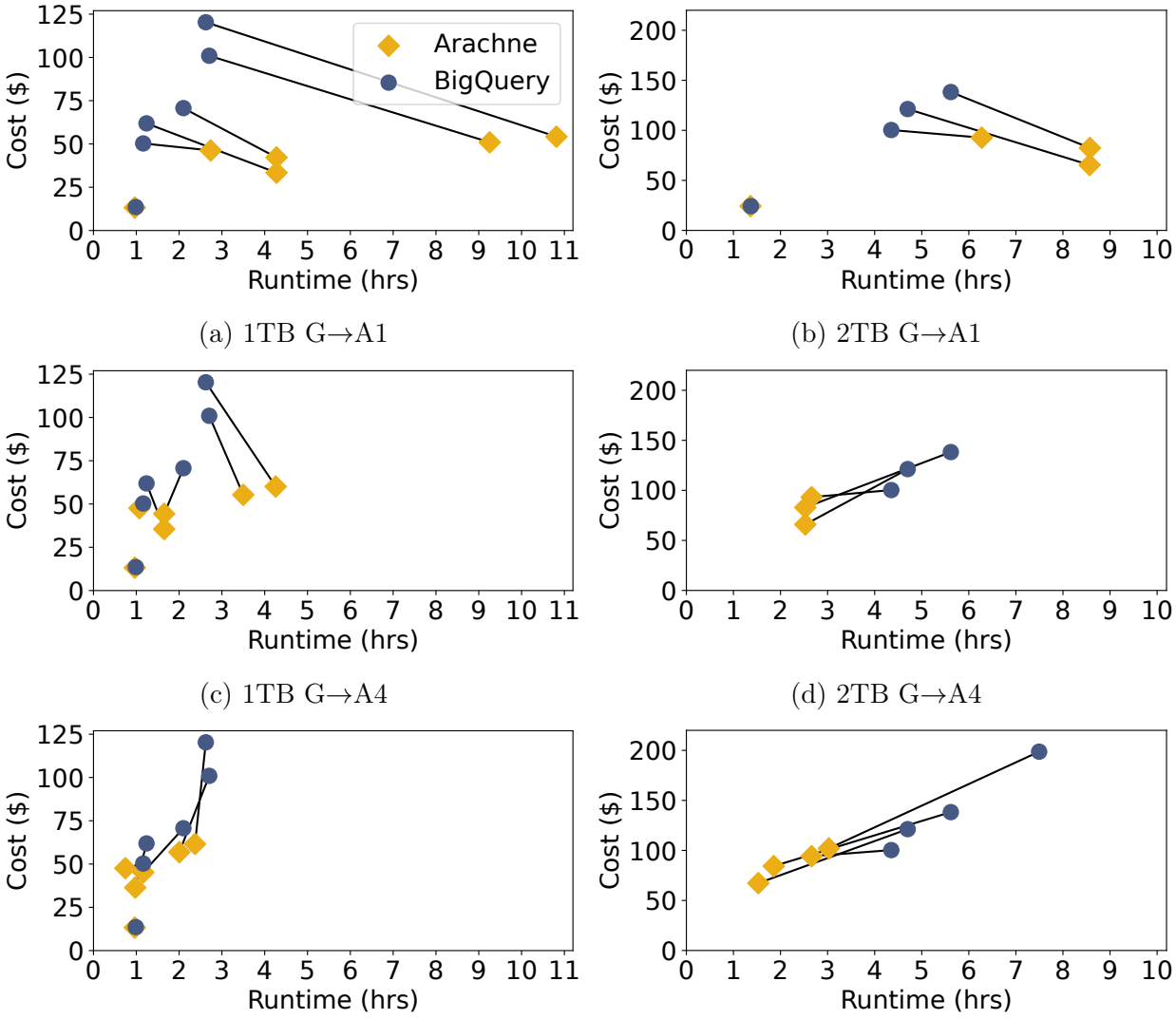
MULTI Plan Analysis. We now focus on MULTI plans which run queries on both BigQuery and Redshift to show the opportunities of combining *price-per-byte* and *price-per-compute* pricing models.

As in Figure 3.5, Figure 3.7 plots workload cost versus runtime for MULTI plans. Blue dots represent BigQuery baseline plans while yellow dots represent Arachne plans. Dots corresponding to the same workload are connected with a line. These plans achieve up to 54% savings and over 35% for most workloads. 2 workloads at 1TB and 1 workload at 2TB save between 2 and 8%. 1 workload at 2TB G→A1 saves less than 1%, as Arachne migrates very few queries and tables which yield marginal savings.

In Figure 3.7, the BigQuery baseline is faster than Arachne's plan because G→A1 does not exploit all the parallelism available in the workload. At G→A4 we see that Arachne's plan is cheaper and closer in runtime to the baseline, and is both cheaper and faster in G→A8. In larger clusters, loading times decrease, and Redshift completes the workloads faster (and cheaper) than BigQuery. At 2TB, the trends are similar to 1TB except that now Arachne is both cheaper and faster than BigQuery even in the G→A4 case.

BigQuery Internal Tables. Loading data into BigQuery is free but significantly increases runtime; loading 1TB took 12 minutes whereas creating external tables took only 20 seconds. Data is stored in a closed format and only accessible via their SQL interface, which incurs query processing costs, or BigQuery's Storage Read API [Google BigQuery Storage Pricing]; both are much more expensive than the blob storage API costs.

For queries over internal tables, BigQuery charges once for each table scanned, even if the table is scanned multiple times in the query. For queries over external tables, BigQuery charges for *each* table scan operator, even if multiple operators scan the same table. So the



(a) 1TB G→A1

(b) 2TB G→A1

(c) 1TB G→A4

(d) 2TB G→A4

(e) 1TB G→A8

(f) 2TB G→A8

Figure 3.7: Cost (USD) vs runtime (hours) for 1–2TB datasets with MULTI plans over Redshift and BigQuery.

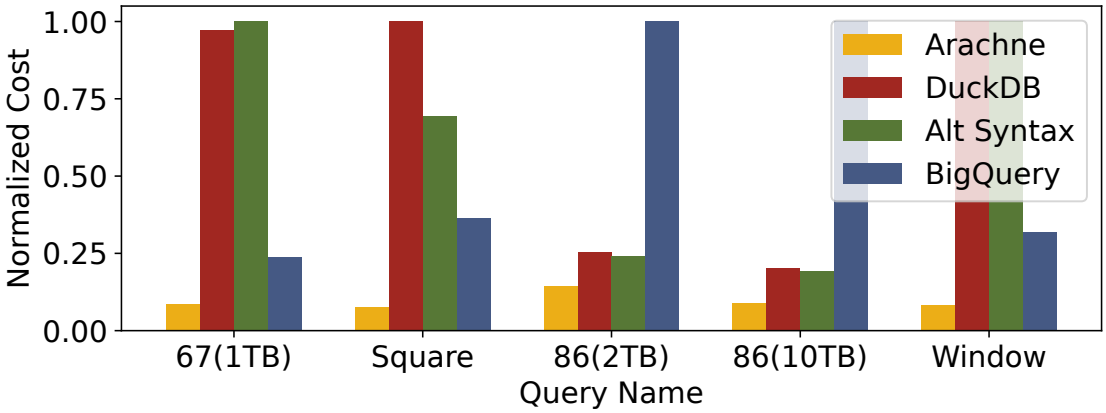


Figure 3.8: Query costs normalized to most expensive plan for Arachne's intra-query plan vs BigQuery, DuckDB, and DuckDB with Arachne-produced text (Alt Syntax).

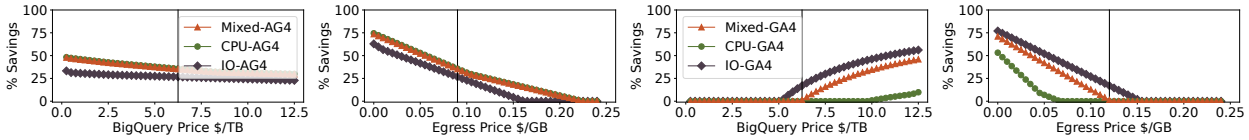
same query will scan fewer bytes and cost less when data is stored internally. We run the inter-query algorithm on G→A1, G→A4, and G→A8 with data stored internally. There are 3-5 multi-cloud workloads in each setup saving 3–20% and 2 with negligible savings, similar to the external case, because of high BigQuery prices and the large IO-bound savings in the Read-Heavy workloads. These savings margins are smaller due to the fewer bytes billed.

Extending Arachne with IaaS+DuckDB

We now explore the cost differences between PaaS (the databases evaluated above) and a new execution backend that deploys DuckDB on a GCP VM, where data starts, to avoid egress costs. The VM costs $1.49/hour with 16 vCPU, 190GB RAM, and 1TB disk to run memory-intensive queries.

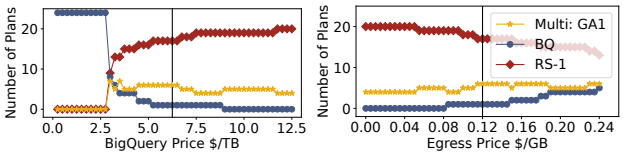
This VM did not have enough memory to run all queries. To concentrate on studying the effect of IaaS, we edited the queries slightly, replacing WITH clauses with CREATE TABLE AS clauses so that intermediate tables are offloaded to disk. While this may increase query runtime it allowed the query to complete so we could proceed with our study. We only consider queries which execute in all execution backends.

In this setup, Arachne does not migrate any queries to Redshift, as IaaS lowered query costs enough that migration is not worthwhile. However, Arachne still achieves up to 55% sav-



(a) A4→G BigQuery Price (b) A4→G Egress Price (c) G→A4 BigQuery Price (d) G→A4 Egress Price

Figure 3.9: (G→A4, A4→G 1TB) % savings for inter-query plans vs. BigQuery or egress prices on Resource Balance Workloads



(a) 1TB Varying BigQuery Price (b) 1TB Varying Egress Price

Figure 3.10: (G→A1 1TB) Inter-query results varying either BigQuery or egress price on Read-Heavy Workloads

ings over the pay-per-byte BigQuery (PaaS) baseline by utilizing cheaper, pay-per-compute IaaS which dramatically reduces costs for the IO-bound workloads. Hence, Arachne can identify opportunities and achieve significant savings with a transparent deployment of DuckDB on IaaS, all without separate user setup, deployment, or maintenance.

Summary

Arachne successfully exploits the inter-query algorithm (**O1**), even in highly skewed workloads if the source pricing model is ineffective for it. Arachne chooses multi-cloud plans saving 35%–56% in most cases by using multiple pricing models.

Table 3.3: Absolute baseline and Arachne intra-query costs.

Query	Arachne	BigQuery	DuckDB	Alt Syntax
67	$1.83	$4.9981	$20.4027	$21.0109
Square	$0.005507	$0.0156	$0.07321	$0.05069
86 (2TB)	$0.089574	$0.62853	$0.1605	$0.15144
86 (10TB)	$0.278728	$3.142	$0.63195	$0.60877
Window	$0.311999	$1.1791	$3.7159	$3.7159

3.6.4 RQ2. Intra-Query Processing

In this section, we explore opportunity **O2** via the intra-query algorithm. These opportunities are significant but occur less often than **O1**, so we report results for five queries that produced a cheaper intra-query plan and analyze the characteristics of these queries.

Queries. Query 67 and WINDOW are run on a 1TB TPC-DS dataset and query 86 is run on a 2TB and 10TB TPC-DS dataset. The WINDOW query performs several joins and group-bys and executes a complex window operation on the result. The SQUARE query, run on 100GB LDBC-SNB dataset, finds squares in social media graphs, e.g., a path from person A to B to C to D and back to A.

Experimental Setup. Data starts in BigQuery, and we consider intra-query plans between BigQuery (pay-per-byte) and DuckDB (pay-per-compute) on a GCP VM. Expensive egress fees restrict data movement and eliminate intra-query opportunities across multiple clouds, so we consider GCP-only intra-query plans. We pay profiling costs to copy data to the VM and execute queries in BigQuery and DuckDB, gathering cost, runtime, and operator cardinality. Arachne converts its internal query plan into SQL to execute subqueries, but we observe that alternate SQL texts for the same logical query can cause the optimizer to choose different physical plans, affecting runtime and cost. To isolate this factor, we consider a third baseline, called **Alt Syntax**, which is the cost of executing the initial query rewritten by Arachne in DuckDB.

Results. Figure 3.8 shows the costs for Arachne's intra-query plan and the baselines normalized to the most expensive baseline for each query. Arachne's plans save 2–5× compared to the next cheapest baseline and orders of magnitude compared to the most expensive baseline, showing the cost saving potential of **O2**. We normalize values to emphasize the relative savings as the absolute savings are small because out-of-memory errors on larger datasets prevented us from using longer-running queries. We show the absolute numbers in Table 3.3 and note that relative costs better represent total savings, as the total savings are

Table 3.4: Runtime (seconds) for baselines and Arachne plans.

Query	Arachne	BigQuery	DuckDB	Alt Syntax
67	4059.96	555.333	50655.30	51107.595
Square	188.226	14.569	168.727	113.961
86 (2TB)	171.051	206.045	381.271	359.023
86 (10TB)	580.898	423.063	1527.825	1471.458
Window	624.970	82.155	9038.641	8954.334

query savings multiplied by the number of times a query executes. While intra-query plans sometimes run slower than the fastest baseline (shown in Table 3.4), this potential slowdown is tolerable if the query is run in a latency-insensitive, periodic workload such as a nightly analytics workload.

Common Characteristics. These queries first join many tables (IO-bound) followed by a window or self-join (CPU-bound). Queries with these stages are good candidates for the intra-query algorithm.

Summary. There are fewer situations where **O2** saves money versus **O1**, but when opportunities exist, the relative savings are significant, especially for queries with the structure discussed above. Profiling costs for 3/5 queries are earned back in under 25 iterations. Query 67 and WINDOW are earned back in 28 and 46 iterations and cost $85.68 and $40.18. The savings achieved are significant and compensate for incurred profiling costs. Re-profiling may be required more frequently for **O2** than **O1**, increasing costs. However, the sizable savings margin can quickly earn back that up-front cost.

3.6.5 RQ3. Simulating Different Cloud Costs

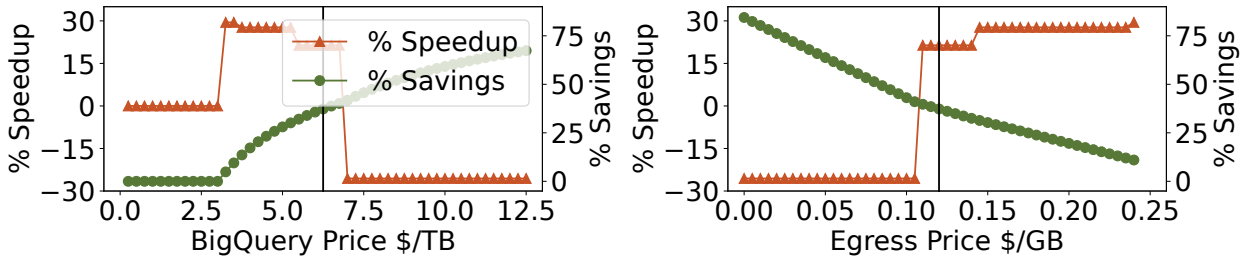
So far, the results shown assume cloud vendor prices as of February'24. In this section, we use profiled inputs (discussed in Section 3.5.2) that are not affected by cloud vendor prices and simulate cloud prices by varying the price inputs to the inter-query algorithm. We vary the price-per-byte (BigQuery price) and egress price from the source execution backend and run the inter-query algorithm on the Read-Heavy (RH) workloads and Resource Balance

Workloads (RBW) to see how varying prices impacts savings. Vertical lines indicate the current price. We use the plan types as in Section 3.6.3–GCP, AWS, or MULTI. For RBW, we show percent savings (Figure 3.9) versus the price being varied in A4→G and G→A4. Figure 3.10 shows plan types for RH in G→A1 at 1TB; trends are similar in G→A4 and G→A8 and at 2TB. The main takeaways are:

- Inter-query savings (**O1**) are robust to changes in prices even for heavily IO-bound workloads.
- Reducing BigQuery price by 40% to \$3.75/TB keeps most RH plans in BigQuery. At 2TB the necessary price reduction increases because potential savings grow as dataset size grows.
- For RBW, in A4→G BigQuery price does not impact plan type but slightly reduces savings. In G→A4, reducing prices by 20% to \$5/TB keeps all plans in BigQuery.
- At low egress prices, MULTI plans are the cheapest option for 4/24 RH workloads in 1–2TB and all RBWs in A4→G. Even if cloud vendors lower financial barriers to data movement, money-saving, inter-query opportunities (**O1**) still exist.
- High egress prices lock-in all RBW plans to their starting cloud. For RH, multi-cloud plans still exist, so savings achievable by using multiple pricing models outpace egress costs.

Runtime vs. Cost Tradeoffs. To observe how cloud prices impact cost and runtime tradeoffs, we zoom in on **Read-Heavy 22** at G→A4 1TB and show the percent savings and percent speedup over the baseline vs. BigQuery and egress prices in Figure 3.11. Negative percent speedup indicates that Arachne's plan is slower.

A small increase in BigQuery price from \$6.25/TB to \$7/TB causes Arachne to migrate more tables in a multi-cloud plan, which runs longer than the baseline but achieves greater savings. A slight decrease of egress cost to \$0.105/GB from \$0.12/GB yields a similar result for the same reasons, as migrating more tables increases savings but also runtime. At many other prices the Arachne's plan is both cheaper and faster. Figure 3.11 illustrates how the



(a) 1TB G→A4 Varying BigQuery Price

(b) 1TB G→A4 Varying Egress Price

Figure 3.11: (1TB G→A4) % Savings and speedup of Arachne for Read-Heavy 22 vs. BigQuery and egress price.

specific tradeoff of runtime and cost is impacted by cloud vendor prices.

Conclusions. Overall, we see that our results are not brittle to price changes: some queries are simply cheaper in different pricing models and prices dictate how large savings are and how large barriers to migration are. More importantly, they show the power platforms have to lock in workloads by adjusting prices slightly; this anti-competitive restriction should be concerning to all of us.

3.6.6 RQ4: Profiling Cost Microbenchmarks

Gathering inter-query inputs (Section 3.5.2) incurs significant *profiling* costs. We show how stale profiles impact savings (Section 3.6.6), how profiling over samples lowers profiling costs (Section 3.6.6), and how noisy runtime estimates impact savings (Section 3.6.6)

Impact of Re-profiling

We first study the savings impact of using stale profiles as data changes. We create 7 datasets with sizes from 100–1200GB with the official TPC-DS generator. A TPC-DS dataset reflects a database at a moment in time for a retail supplier. While tables tracking sales and returns only grow with overall data size, other tables tracking inventory or customers grow and shrink as overall data size increases, per the TPC-DS specification [TPC-DS].

We let each data size be the state on a given day over 7 days. In Figure 3.12 we compare

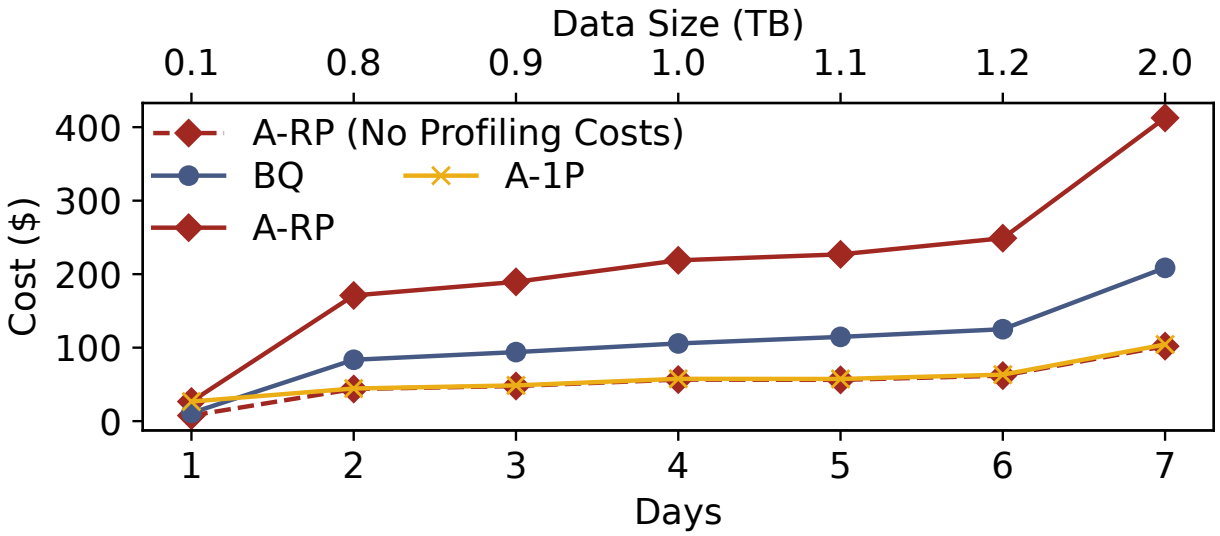


Figure 3.12: (G→A4) Cost (USD) vs days and data size (TB)

four execution strategies in G→A4 as data changes. We compare the BigQuery baseline (BQ), against two Arachne strategies: profiling only on day 1 (A-1P) and using that profile for days 2–7; and profiling as soon as data changes (the solid red line, A-RP). A-1P and A-RP include profiling costs. Finally, we show the cost of A-RP without profiling costs as the dotted red line.

We compare these strategies on Read-Heavy 2 which showed the largest gap between A-1P and A-RP. While A-RP is cheaper over time, it is at most 2% cheaper than A-1P. Daily re-profiling costs make A-RP far more expensive than BQ. A-1P quickly compensates for profiling costs and saves significantly over BQ. A-1P's stale profile still captures which queries are cheaper in which backend, so small errors in the profile do not appreciably diminish savings.

Sampling

We now show how sampling reduces profiling costs with low error. While estimating runtime from samples is difficult for some queries [Chaudhuri et al., 1999, Liang et al., 2021, Yang et al., 2019], most profiling cost are from *pay-per-byte* pricing models, where runtime does

not affect cost.

We show cost and estimation error for samples of 15, 25, 50, and 100% of data in Table 3.5. When profiling over all data, 20/24 workloads earn back profiling costs in 4 iterations. Small samples estimate the inputs well and in most cases lower the number of needed iterations to 1–2. Read-Heavy 7 chooses a GCP-only plan so achieves no savings. Read-Heavy 17 only achieves marginal savings and needs many iterations, though sampling lowers the net cost of profiling. We do not claim that sampling is the best approach, only that it cheapens profiling. More sophisticated approaches, e.g., using parameterized cost models to sample non-linear operators [Li et al., 2019] can further reduce error, which we leave for future work.

Profiling versus Runtime Estimation

We estimate query runtimes in Redshift by training a Kernel Canonical Correlation Analysis (KCCA) model, as proposed by Ganapathi et. al. [Ganapathi et al., 2009] in 2009[2]. KCCA finds correlated clusters of training features and labels to make predictions. However, hardware advances over 15 years mean that queries run much faster, so most training points are clustered together despite the size and diversity of the training set, lowering the reproduced model's accuracy. Nonetheless, we created 2842 training queries as the original 3102 queries used were not available after reaching out to the authors. We used a 1GB TPC-DS dataset and also ran some queries on 100GB and 1TB datasets to get a broader range of runtimes. We make a significant effort to replicate the setup and create a runtime estimation method for SQL queries.

Inter-query plans using runtime estimates are 66% more expensive than plans using profiles on W-Mixed in the A1→G setup and 13% more expensive in G→A1 when they cost the same as the baseline. Noisy estimates result in Arachne missing valuable savings opportunities and greatly diminish savings margins, illustrating the detrimental impact of

2. The authors of the original paper could not provide the materials to reproduce their work; we replicated their model effort to the best of our ability

estimation on inter-query savings.

3.7 Related Work

Other Cloud Databases. Many other cloud and third-party databases scan cloud storage like Snowflake, Azure Synapse, Trino, Apache Hive, Amazon Athena, and SparkSQL [Dageville et al., 2016, Azure Synapse, Aguilar-Saborit et al., 2020, Presto, Thusoo et al., 2010a, Amazon Athena, Armbrust et al., 2015]. These databases use per-second billing (Presto, Hive, SparkSQL), per-byte billing (Athena), or some combination of the two. While we could have used other systems in our evaluation, Google BigQuery and AWS Redshift effectively represent both pricing models across clouds, enabling us to evaluate opportunities **O1**–**O2**.

Cloud Cost Savings. Prior work on money savings focuses on scheduling algorithms [Wu et al., 2015], exploring cost sources in different execution backends [Tan et al., 2019], or using S3 Select [Hunt, 2017] to speed up queries and lower costs [Yu et al., 2020]. Leis and Kuschewski model per-second costs for cloud workloads [Leis and Kuschewski, 2021]. Recent work has also focused on achieving savings using spot instances [Wu et al., 2024], minimizing network egress prices [Wooders et al., 2024], and finding cheaper configurations for cloud deployments [Bang et al., 2024]. To the best of our knowledge, Arachne is the first effort to systematically explore savings opportunities for analytical queries by using multiple databases with different pricing models.

Other Complementary Optimizations. Other prior work saves money through semantic caching and distributed query optimization techniques [Dar et al., 1996, Durner et al., 2021, Kotidis and Roussopoulos, 1999, Perez and Jermaine, 2014], optimizing data placement [Kossmann et al., 2000, Polychroniou et al., 2018], and view selection and materialization in data warehouses [Nadeau and Teorey, 2002, Chirkova et al., 2002, Auto-MV, Redshift Spectrum]. These efforts save costs within a single pricing model and can be ap-

Sample %	15			25			50			100		
Dataset	Cost	Iter	Error	Cost	Iter	Error	Cost	Iter	Error	Cost	Iter	Error
Read-Heavy 0	30.39	1	0.02	49.33	1	0.03	94.2	2	0.03	177.19	3	0.0
Read-Heavy 1	30.04	1	0.02	48.71	1	0.03	92.97	2	0.03	174.93	3	0.0
Read-Heavy 2	27.41	1	0.02	44.49	1	0.03	84.54	2	0.04	157.69	3	0.0
Read-Heavy 3	17.54	1	0.03	28.17	1	0.04	53.45	2	0.05	97.78	4	0.0
Read-Heavy 4	25.6	1	0.03	41.33	2	0.04	78.14	2	0.04	143.91	4	0.0
Read-Heavy 5	26.12	1	0.03	42.24	2	0.03	80.01	2	0.04	148.96	4	0.0
Read-Heavy 6	27.55	1	0.02	44.49	1	0.03	84.55	2	0.04	157.9	4	0.0
Read-Heavy 7	13.86	N/A	0.06	21.87	N/A	0.09	39.3	N/A	0.1	65.17	N/A	0.0
Read-Heavy 8	28.25	1	0.02	45.62	1	0.03	86.77	2	0.03	162.65	4	0.0
Read-Heavy 9	30.0	1	0.02	48.62	1	0.03	92.77	2	0.03	174.48	3	0.0
Read-Heavy 10	30.16	1	0.02	48.91	1	0.03	93.36	2	0.03	175.64	3	0.0
Read-Heavy 11	19.26	5	0.04	30.62	8	0.05	57.13	15	0.06	101.91	26	0.0
Read-Heavy 12	28.97	1	0.02	46.91	1	0.03	89.32	2	0.03	167.45	3	0.0
Read-Heavy 13	29.81	1	0.02	48.28	1	0.03	92.13	2	0.03	173.19	3	0.0
Read-Heavy 14	30.42	1	0.02	49.44	1	0.03	94.21	2	0.03	177.39	3	0.0
Read-Heavy 15	22.77	2	0.03	36.53	2	0.04	68.43	4	0.04	126.6	7	0.0
Read-Heavy 16	26.54	1	0.02	42.94	1	0.03	81.52	2	0.04	152.01	4	0.0
Read-Heavy 17	8.65	26	0.05	13.47	40	0.06	24.84	74	0.08	42.85	127	0.0
Read-Heavy 18	29.78	1	0.02	48.31	1	0.03	91.96	2	0.03	173.15	3	0.0
Read-Heavy 19	30.06	1	0.02	48.85	1	0.03	93.03	2	0.03	174.9	3	0.0
Read-Heavy 20	30.31	1	0.02	49.17	1	0.03	93.88	2	0.03	176.83	3	0.0
Read-Heavy 21	28.35	1	0.02	46.01	1	0.03	87.54	2	0.03	164.02	3	0.0
Read-Heavy 22	20.58	1	0.03	33.3	2	0.04	63.02	3	0.05	114.34	5	0.0
Read-Heavy 23	29.89	1	0.02	48.48	1	0.03	92.51	2	0.03	173.99	3	0.0

Table 3.5: Profiling costs, iters to earn back profiling costs, and est. error for 15, 25, 50, and 100% samples (G→A1 1TB).

plied to databases *prior* to our analysis across multiple pricing models; for that reason they complement our research.

Cloud-Agnostic Query Execution. Recent position papers have emphasized the need to build cloud-agnostic data infrastructure. Berkeley's Sky Computing vision outlines opportunities for multi-cloud workload execution [Chasins et al., 2022]. Our work on Arachne emphasizes cost savings across pricing models.

Federated Query Execution. Some prior work improves performance for federated queries by using metadata from federated sources [Charalambidis et al., 2015], by improving query planning for federated queries [Ewen et al., 2006], or by building full federated query systems [Josifovski et al., 2002]. These works do not aim to save money and consider a single execution backend and multiple storage endpoints, while Arachne uses multiple execution

backends with different pricing models to save money.

3.8 Conclusion

This paper presents, exploits, and evaluates two money saving opportunities for cloud analytical workloads. The key is to schedule queries based on the resources they consume onto *beneficial* pricing models offered by cloud vendors. We measure hard-to-estimate query information, implement the inter- and intra-query algorithms, and use IaaS to save money, all while honoring runtime constraints.

We hope this work will encourage further investigation into multi-cloud savings opportunities. Ideally, this line of work fosters competition between cloud vendors, driving down prices and benefiting users. Cloud vendors may, however, simply modify prices to prevent data movement and lock-in users. Even in extreme situations multi-cloud opportunities exist, and we hope that cloud vendors choose to reduce costs for users and pay for the revenue loss by becoming more energy-efficient to lower internal costs.

CHAPTER 4

MORE EFFICIENTLY MANAGING DATA WITH A VALUE OF DATA INDICATOR

4.1 Introduction

Users leverage cloud data warehouses and lakehouses, such as Amazon Redshift [Armenatzoglou et al., 2022] or Databricks [Databricks], to store relational data and execute tasks like SQL queries over that data. Users must make many decisions about how to configure and use these systems to store, transform, and process their data, such as deciding what cloud storage tier to place their data lake in (data lifecycle problems), how many compute resources to allocate to a data warehouse, what data to manually clean, and so on. Data managers, or those users responsible for making these decisions such as engineers, administrators, or analysts, aim to make these decisions to best serve workload-specific and organizational goals, such as latency constraints, accuracy requirements, or monetary cost budgets.

Making efficient decisions is vital for organizations. For instance, companies who solve data lifecycle problems by using AWS S3 Intelligent tiering, which automatically archives data based on data age and access frequency, saved $6 billion since 2018 compared to those who used AWS S3 standard storage [Amazon S3 Intelligent-Tiering]. However, inefficient decisions can be equally detrimental—indeed, poor data quality for key tasks costs organizations on average $12.9 million annually [Gitnux, 2026].

Scenario 1. *A data lake contains millions of records of transactions of various retail products originating from different source systems. An analyst is asked by a product manager to produce a report on clothing sales, and later is asked by a marketing executive for a separate report about wristwatch sales. Each report requires that the analyst manually clean the subset of transaction data relating to each product. What task should the analyst prioritize, and what data should be cleaned first, to best serve the organization?*

Scenario 2. *A cloud data warehouse charges users based on the amount of data read per query. An administrator aims to reduce cloud costs by clustering tables [Cloud, 2026a] on certain columns to reduce IO. What columns should the administrator cluster each table on?*

Scenario 3. *An engineer setting up a transactional database to run operational workloads uses a buffer pool to cache data pages in memory to reduce query latency. Given limited available memory, how should the engineer prioritize what data pages to cache?*

Data managers often cannot feasibly solve these problems perfectly; for instance, in a large data lake of potentially low-quality records, identifying what data to clean, how to implement cleaning pipelines, and what resources to allocate to those pipelines requires understanding how each piece of data impacts the performance, cost, and accuracy of each task, whether cleaning that data would improve those metrics, and the importance of each task to an organization in order to build an informed solution. This approach would likely require an infeasible amount of human time and resources.

As such, data managers tend to rely on limited signals and tools that aim to approximate efficient solutions, but these can lead to decisions that do not align with organizational goals while still requiring data managers to perform significant manual work. For instance, relying solely on a proxy heuristic like data access frequency to decide whether to archive data may significantly slow down a key task which accesses data infrequently, even though maintaining its performance is a key goal for the organization.

Furthermore, because each data management decision requires different expertise and involves different data systems, data managers often approach each decision individually. These bespoke solutions, which rely on limited signals or tools, are often only applicable to a narrow set of data management scenarios. Data managers thus need a signal that is more widely applicable and better aligns cloud setups with organizational goals.

We claim in this paper that data managers can make more efficient data management decisions across a wide range of problems with access to the **value** that data has to tasks,

like detecting fraudulent transactions, in their organization. Existing systems like data warehouses and lakehouses do not provide access to the value of data, but new services like the Unity Catalog which provide a centralized interface for data governance and management present an opportunity to capture the value of all data managed by an organization and expose it to decision makers across an organization.

The value of data signal is *instrumental*, as data gains value if and only if it helps solve tasks important to an organization. For instance, the data used for a vital fraud-detection pipeline is very valuable because it serves an important task, while data never used has no value. The instrumental value of data can guide more efficient decisions across a wide range of data management problems because it better captures the importance of data to an organization than individual dimensions such as access frequency, as the value of data also considers the importance of each task to an organization.

For specific data management problems, approaches have been proposed that use instrumental data signals, but these are still only narrowly applicable and organizations have not integrated this signal as a first-class citizen in their data management processes. For instance, the five-minute rule computes the value of a data page from its access frequency, its read time from disk, and the monetary cost of memory, and makes caching decisions based on this signal. However, this applies primarily to caching decisions for on-premise computing setups rather than elastic cloud setups.

Other proposals aim to integrate instrumental data signals into underlying systems, such as Glavic et. al. [Glavic et al., 2024] who propose a frequency-based metric for the value of data. However, relying solely on access frequency to compute the value of data can lead to inefficient data management decisions, and these approaches have not been broadly adopted by underlying systems in part to avoid impacting the performance of tasks on critical infrastructure. Thus, it is imperative that the benefit of the value of data be well justified and the cost of its implementation minimized.

To make the value of data accessible to data managers via underlying systems, we propose a general indicator for the value of data, which computes a number for each piece of relational data in an organization, such as a table, column, or row. We call each such piece of data a data element. We first present a relational interface that exposes the value of data indicator to illustrate how data managers can access and use the indicator to better solve a broad range of data management tasks, regardless of the implementation of the indicator.

We then present the instrumental model for data value and data utility proposed by Castro Fernandez [Castro Fernandez, 2025], and we present how we compute the value of data indicator. We discuss the limitations of more limited heuristic signals that do not consider the value of tasks to an organization to motivate our approach, and we argue that task values should be provided by data users to best capture the importance of tasks to organizational goals. We also discuss how task values can be initially incomplete and improved over time to refine the quality of the indicator.

Finally, we present a prototype implementation of the value of data indicator on the Databricks Unity Catalog, which presents a unified interface through which users can run both transactional workloads through the Postgres-based Lakebase and analytical queries through the Lakehouse without needing to run ETL pipelines to transform data. This implementation supports tracking the value of data indicator only for tables or columns. We modify Apache Spark, which is the core engine behind the Databricks Lakehouse, to build an implementation of the value of data indicator using the QueryExecutionListener API that enables tracking the value of data indicator for Apache Parquet row groups as well.

We evaluate the efficacy of the value of data indicator by illustrating how it helps data managers across three different scenarios to more efficiently make data management decisions by improving query performance and the overall utility gained from a workload. We evaluate our implementation and show that the relative cost of implementing the indicator is compensated by the better decisions it enables. In summary, the contributions of this paper

are:

- A relational data model through which data managers can access the value of data indicator to guide data management decisions.
- A method for computing the value of data indicator based on agent-defined task values and the relevance of data to those tasks.
- An implementation for the value of data indicator on the Databricks Unity Catalog and Apache Spark for relational data.

Next, Section 4.2 describes our proposed interface for the value of data indicator, illustrating how data managers can use this interface to guide data management decisions regardless of the underlying implementation of the indicator. In Section 4.3, we describe the model we use to compute the value of data and discuss other approaches to making data management decisions and the limitations of these approaches. Section 4.4 presents how to compute the value of data, discusses eliciting task values from data users, and studies how the choice of granularity—or how data is subdivided into discrete data elements—impacts data value. Section 4.5 describes our implementation on Apache Spark, and in Section 4.6 we evaluate the benefits of the value of data indicator and evaluate the query performance overhead of our implementation. Finally, we present related work in Section 4.7 and conclusions in Section 4.8.

4.2 How the Value of Data Indicator Can Benefit Data Managers

In this section, we illustrate how a value of data indicator can benefit data managers. We assume that a value of data indicator exists, and we illustrate how data managers can leverage this indicator to better solve tasks across four categories of data management tasks. We then describe the concrete relational interface through which data managers access the value of data indicator.

4.2.1 Categories of Data Management Problems

We first enumerate four categories of data management problems and cite discussion in industry about each. We discuss how data managers solve these problems today and describe how the value of data indicator can better serve managers across these categories.

Category 1: What Data is Most Valuable to an Organization? Data managers want to understand which data elements are the most valuable to products and tasks across their organization to survey the landscape of their data [Cuffari et al., 2023, Octopai, 2024] and understand what data is most critical for different teams or products [Libarikian et al., 2022, Tietoevry, 2024, Desai et al., 2021, Bergman, 2022]. For example, a CTO or division leader may want to understand if a newly procured dataset is being used or if an unexpected dataset is driving critical value.

Category 2: Resource Allocation Under a Budget. Data managers must decide how to allocate a limited resource across a subset of data elements [Krantz and Jonker, 2023, McKinsey & Company, 2023], as in Scenarios 1–3 where a data manager had to decide what data to clean with limited human time, what columns to cluster a table on, or what data to cache.

Category 3: Data Lifecycle Management. Cloud vendors offer various storage tiers, with cheaper tiers offering slower access times. Data managers want to decide what data to archive to reduce storage costs while obeying governance constraints such as regulatory requirements to store financial data for a certain period of time. They also may want to understand if important data is stale and should be refreshed to improve the performance of tasks that rely on it [IBM, 2026, News, 2023, Ghorpode, 2025].

Category 4: Data Setup Evolution. Making modifications to an organization's data setups, such as migrating data from blob storage to an analytical database, building an index on a database, or partitioning or clustering a table in a data lake can improve performance or streamline data maintenance issues [Slingerland, Darnell]. Making these changes is often

expensive, so data managers must carefully decide what changes to make, such as migrating data or building an index, and what data to apply these changes to.

Data managers use a mixture of tools and other signals like heuristics to help in these varied decisions, but each requires its own expertise to use, may not be compatible with other tools or signals, and may only apply to a subset of tasks. As a result, data managers often do not have access to the value of data across their organization, leading to less effective data management decisions.

For instance, in Category 1 data managers can manually retrieve data access patterns from system tables, such as BigQuery's information schema [BigQuery Information Schema] or Snowflake's account usage [Snowflake Account Usage] tables, to understand what data each product or team accesses, but this approach does not answer how valuable these different products are or how valuable the data is to these products. For Category 2, data managers can set budgets for projects or pre-buy capacity pools [Williams and Joshi, 2023], but they must still decide what budgets to set manually, and do not have access to a general signal for the value of their data to guide that decision. For Category 3, data managers can use tools like AWS S3 Intelligent Tiering [Amazon S3 Intelligent-Tiering] to automatically move data to cheaper storage tiers based on access patterns, but this approach may identify an inefficient solution for infrequently accessed data and requires data managers to manually identify what data to refresh. For Category 4, data managers can modify setups by creating partitioned or clustered tables [BigQuery Partitioned Tables, Cloud, 2026a], or migrate data between clouds [AWS DataSync], but they still must manually choose what columns to partition or cluster on or what data to migrate.

Data managers can make better decisions with less manual effort across these categories with access to an indicator for the value of a data element to each task that accesses it. Aggregating the value of a data element across all tasks yields the total value of a data element. Aggregating the value of data across a subset of tasks, such as those from a specific

team or product, yields the value of a data element to that team or product.

For Category 1, data managers can identify the most important data elements by sorting each data element by its data value. Data managers can determine the value of data to a specific set of tasks, such as those from relevant teams or products, by aggregating data value for only these relevant tasks. This allows data managers to understand how a data element contributes to different areas of an organization while avoiding the pitfalls of relying on access patterns alone to determine data value.

For Category 2, a data manager can use the value of data indicator and filter by tasks associated with the setup or operation under consideration, and then sort data elements by value. The data manager can then prioritize the highest value data, for instance choosing to store this data on faster storage or spending time to clean it or using this information to guide their decision with other external inputs or constraints. For instance, a data manager may need to identify tasks with data quality issues to choose what data elements could be maximally impactful if cleaned.

For Category 3, data managers can select data elements with no value or very low value as candidates for storing in cheaper and slower storage because these data elements are either not accessed or are only accessed by unimportant tasks. They can also use the value of data indicator to decide if infrequently used data is valuable enough to keep in faster storage. Data managers can also selectively monitor high-value data for staleness to cut down on the manual effort of identifying data worth refreshing.

For Category 4, data managers can sort relevant data elements by their value to a set of relevant tasks to identify what data to prioritize for setup changes. For instance, data managers can identify the value of two relational columns within a table to analytical workloads, filtering by analytical queries, to decide what column a table should be clustered on or what data should migrate to an analytical database.

Required External Inputs. We observe that the value of data indicator alone cannot solve

Table Name	Column Name	Row Group ID	Task ID	Timestamp	Value
TableA	ColumnA	0	Task0	26-01-01 12:00	50
TableA	ColumnA	1	Task0	26-01-01 12:00	50
TableB	ColumnB	0	Task1	26-01-01 12:20	25

Figure 4.1: Sample content for the Data Value table, which contains information to access the value of data indicator.

all of these data management problems if external context is needed. For instance, understanding which tasks perform similar operations or what tasks are sensitive to data quality issues requires context from other data managers or insights stored in external documents. However, with a relevant set of tasks or data elements, data managers can use the indicator to understand what data is the most valuable to their organization. With access to the value of their data, managers have a concrete, quantitative input to guide their decision-making process.

4.2.2 How To Access the Value of Data Indicator

We now describe how data managers can access the value of data indicator from two relational tables: the Data Value table and the Task Value table. These two relational tables are stored within one of an organization's existing data catalogs and are managed by the same data managers. As these are relational tables, managers can access them via SQL queries and build views on them.

The Data Value table logs the value for each data element accessed by a task in an organization. We show example data for this table in Figure 4.1. This table supports three different granularities, or ways of subdividing data into data elements: table, column, and Apache Parquet [Apache Parquet] row group, which contains all column data for a subset of rows in a table. The indicator thus assigns a numerical value to each data element as defined by the chosen granularity. We note that, depending on the granularity used, row group ID and possibly column name may be NULL.

Data managers directly access the value of data through the Data Value table. The table's Value attribute is how much value was attributed to a specific data element–identified by its table name, column name, and row group ID–from a specific task identified by its task ID. The Timestamp attribute logs when the query finished to enable data managers to observe the value of data over arbitrary periods of time. We see in Figure 4.1 that two tasks run 20 minutes apart. Task 0 accesses two row groups from a single column in Table A and assigns each a value of 50, while Task 1 accesses one row group from a column in Table B and assigns it a value of 25.

The granularity impacts how many data elements we track the value of data for, which will impact the number of rows in the Data Value table. For instance, because there are typically far fewer columns than row groups in a database, tracking at the row group granularity will result in a much larger Data Value table, as there is one row written for each task that accesses a data element.

The Task Value table has three attributes: a task ID, the task text or code, and the task's value. This table enables data managers to understand what operations a task is performing and store the value of a task which is used to compute the value of data. This interface supports arbitrary methods of assigning values to tasks, including a simple assignment of the same value to all tasks.

We note that the value of data is a function of the task value, so the Value attribute in the Data Value table simply stores the result of this function. Thus, this interface can support any definition and implementation of an instrumental value of data.

To access the value of data for a set of data elements, such as a set of columns, a data manager can write a SQL query over the Data Value table filtering for those column names and aggregating Value for each one, potentially adding filters for specific task IDs or time windows. Data managers can use this to identify the most valuable data elements within specific timeframes, products, or teams, enabling them to make better decisions across the

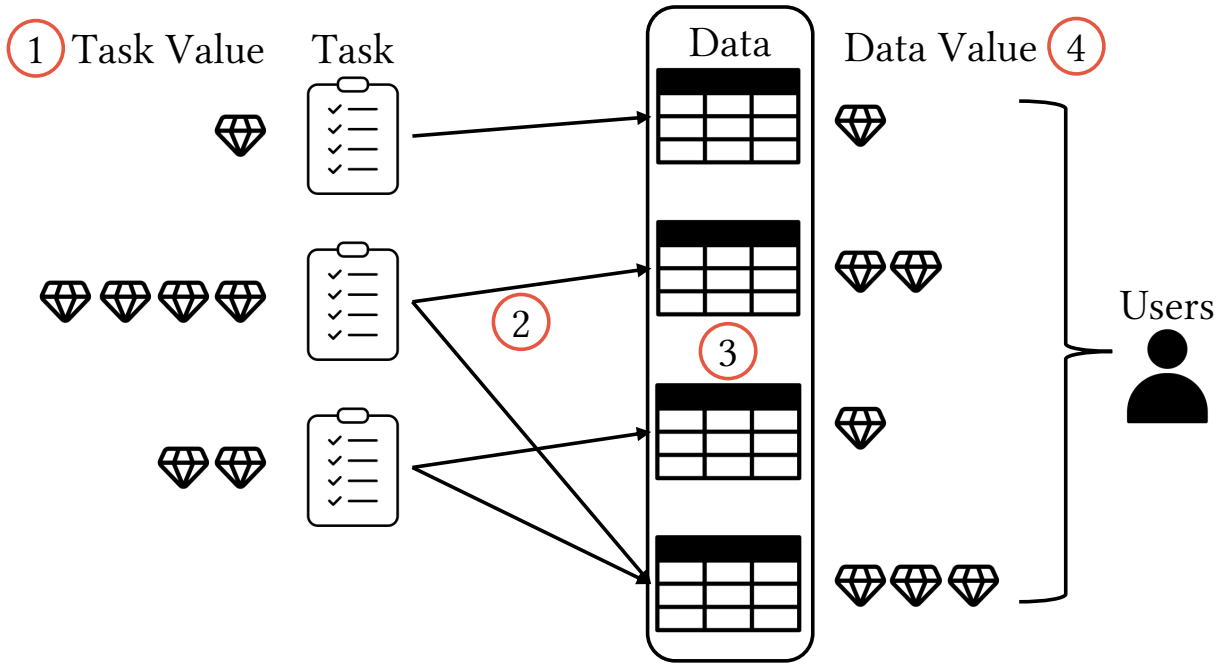


Figure 4.2: Tasks, with value (1) shown in diamonds, access (2) relevant data elements (3). Data elements accrue value (4) from each task that accesses it, and users access this value.

four categories of data management problems discussed above. Data managers can use the Task Value table to understand the operations a task performs or the task's value.

4.3 Modeling The Value of Data Indicator

Having illustrated how data managers can access and use the value of data indicator, we now describe the model we use for the instrumental value of data. We then illustrate the limitations of other heuristic approaches to solving data management problems to motivate our approach.

4.3.1 Model Overview and Inputs Needed to Compute the Value of Data Indicator

The model proposed by Castro Fernandez [Castro Fernandez, 2025] defines how **tasks** are computations to be completed over data to fill an **information need**. Data gains value by

contributing to the information need of a task, rather than having value in and of itself, so the value of data depends on the tasks that it contributes to.

Figure 4.2 shows how a set of tasks each use a set of data elements which are part of a larger dataset, with arrows drawn between tasks and relevant data elements. Each task has a value, depicted with diamonds, representing its importance to an organization. The value of data is derived from three inputs. First, the value assigned to each task represented by **1** in a red circle in Figure 4.2. Second, capturing what data elements are relevant to each task, represented by **2** in a red circle. And third, granularity, or how data is divided into discrete data elements like tables, columns, or rows, represented by **3** in a red circle.

Finally, data value, labeled with **4** and represented by diamonds, is computed from these inputs and revealed to users through the interface described in Section 4.2. We discuss how we collect the relevant data elements (input **2**) for each task in our implementation in Section 4.5.

An implicit assumption in input **1** is that all tasks in an organization have a unique identifier so a value can be assigned to them and a task's value can be accessed to compute the value of data. However, organizations may not have such a system, and instead, teams rely on the unique IDs generated by individual data systems to identify tasks. However, these IDs may not be unique across all the systems used in an organization. Without an organization-wide task ID system, data managers may need to limit the scope of the indicator to only consider tasks for which unique IDs are accessible, such as SQL queries run through a cloud data warehouse.

4.3.2 How the Value of Data Indicator Improves Over Common Approaches

We now discuss some common approaches to data management problems and identify how these approaches can lead to inefficient solutions to these problems because they fail to consider the value of a task to an organization. Consequently, the instrumental value of data

indicator, which captures this task value when computing the value of data, can guide data managers to better data management decisions.

Least Recently Used (LRU). LRU is a heuristic which prioritizes recently accessed data, sometimes used as an eviction policy for in-memory caches, such as a database buffer pool. LRU does not differentiate the quality of access, so it does not matter whether data was accessed by incredibly important tasks, like a major fraud detection service, or by less important tasks like an analyst doing exploratory work. As a result, this heuristic could evict an incredibly valuable column accessed once every day by an important fraud detection workload in favor of columns accessed often by less important exploratory work. This outcome is worse for the organization, as it reduces performance for this key task precisely because it fails to consider the value of each task to the organization.

Frequency-Based Model. Other approaches across a range of data management problems prioritize data based on the frequency of access. For instance, an instrumental model proposed by Glavic et. al. [Glavic et al., 2024] assigns each task the same value so that a task's value is its relative frequency: if Task A runs 100 times in a workload of 1000 total task executions, it has a value of 0.1. Glavic et. al. track data relevance to tasks through provenance or provenance sketches [Niu et al., 2021, Li et al., 2026].

Each data element relevant to a task adds the task's value to its data value, so every data element used by Task A adds 0.1 to its data value. Thus, a data element's value ranges between 0 (relevant to no tasks) and 1 (relevant to all tasks). Relying on relative frequency alone to dictate task value fails to capture how data managers view the importance of their tasks. Clustering a table on a frequently accessed column may benefit a majority of tasks, but this may not yield the most significant benefit to the organization if this choice disadvantages valuable tasks that run less frequently.

The five-minute rule is also a frequency-based heuristic that can guide data management decisions for some problems. Based on the frequency of access over a period of time, we can

compare the cost of storing data in memory versus the cost to fetch it from disk repeatedly. The cost of memory and time to read from disk dictates how frequently data must be accessed to be worth caching. This uses access frequency to make a caching decision, but again, it assumes that every access is of equal value. By not considering the value of tasks, these decisions may not best serve an organization. A frequency-based model for the value of data which also considers the value of tasks to an organization, could yield data management decisions that better serve the organization, such as by caching data in memory that has contributed the most value to an organization's tasks.

Organizational or Interpersonal Information. Data managers may also solve data management tasks with non-technical solutions and knowledge gained by speaking to colleagues, consulting documentation, or trial and error. For instance, a data manager may prioritize what data to clean based on a team's reputation for patience, with whom they work more regularly, precedent about what data has been prioritized in the past, or simply prioritizing the requests of the highest ranking employees first. While this is useful context, data managers still lack a systematic view of the value that data provides to an organization, and consequently may make decisions that do not best serve the organization because they lack the tools to identify better solutions.

Observability and Monitoring. Data managers may use tools to monitor specific datasets or metrics to identify potential problems and then manually investigate and address them [Post, 2023, Moses, 2025, Kirwan, 2024]. For instance, a data manager may monitor the null count or distribution of a column to detect if data quality has degraded before verifying if that data has truly degraded and prioritizing it for cleaning. These approaches use observability tools like Monte Carlo [Monte Carlo], Bigeye [Bigeye], or in-house solutions [Uber Monitoring Data Quality]. Because this approach requires significant manual effort, data managers typically choose datasets they perceive to be important or sensitive to changes to focus observability resources on. However, that data may not be critically linked

to the dropping accuracy of a model, and the most critical data quality issues present may not be linked to any important tasks at all. By not allowing the value of tasks to guide what data their attention and effort focuses on, data managers may spend vital resources monitoring data that does not best serve an organization, while more valuable data may be neglected.

The Value of Data Indicator. These approaches, by not considering the value of tasks, cannot capture scenarios where value is misaligned with some proxy metric or approach, such as access frequency or organizational precedent. In these instances, relying on these other approaches can lead to inefficient data management decisions because they fail to prioritize the data that is actually the most valuable to an organization. The instrumental value of data indicator uses task values to adapt to scenarios where these individual metrics alone identify efficient decisions and can adapt to scenarios where a more complex prioritization of data elements is more efficient.

4.4 How To Compute the Value of Data Indicator

We now present how we compute the value of data indicator given the three inputs described: task values, the mapping of relevant data elements to tasks, and data element granularity. We discuss how we elicit the task value input from data users and then study how granularity, the third input, impacts the value of data indicator.

4.4.1 Calculating the Value of Data Indicator

Given a value assigned to each task, a mapping of what data elements are relevant to each task, and a choice of data element granularity, we compute the indicator by equally dividing the task value among all relevant data elements. The total value of a data element is the sum of all values assigned to it from tasks. For example, if we value tasks in monetary units like US dollars, a task with value $50 that accesses two data elements will assign $25 to each

data element each time it executes. If it runs four times, the data element will have a total value of $100. Hence, the frequency at which a task executes impacts the observed value of data.

We make two decisions in our approach. We first decide that the total value attributed to relevant data elements from a task should sum to the value of the task. This ensures that if the task value is expressed in absolute units, such as US Dollars, the value of data can be directly interpreted in those same units.

Second, we decide to divide task value *evenly* among relevant data elements. Formally, the model from Castro Fernandez [Castro Fernandez, 2025] views both data and the information need of a task as random variables and defines the value of data as the mutual information between data and the information need. Implementing this model would require significant domain knowledge over an organization's potentially large data stores and poses a significant modeling challenge even with such knowledge. An alternate heuristic approach divides task value proportionally to the size of a data element and assumes that larger data elements contribute more information, which may not be true. Instead, given the absence of other information, we choose to assume that all relevant data elements contribute equally to a task and divide task value evenly. We now study the difference in the observed value of data indicator between our choice to evenly divide and other strategies to divide task value.

An Empirical Comparison of Dividing Strategies and the Frequency-Based Model. We compute the value of data under three different dividing strategies: **Full**, which assigns the full value of a task to each data element, **Equal**, which divides the task value evenly among relevant data elements, and **Cardinality**, which divides task value proportional to the number of rows within a data element. We note that our value of data indicator can support any of these three dividing strategies, as well as more complex strategies that may be proposed in the future. We use the TPC-DS [Nambiar and Poess, 2006] and TPC-H [TPC, 2026] workloads at SF30 for tasks and data.

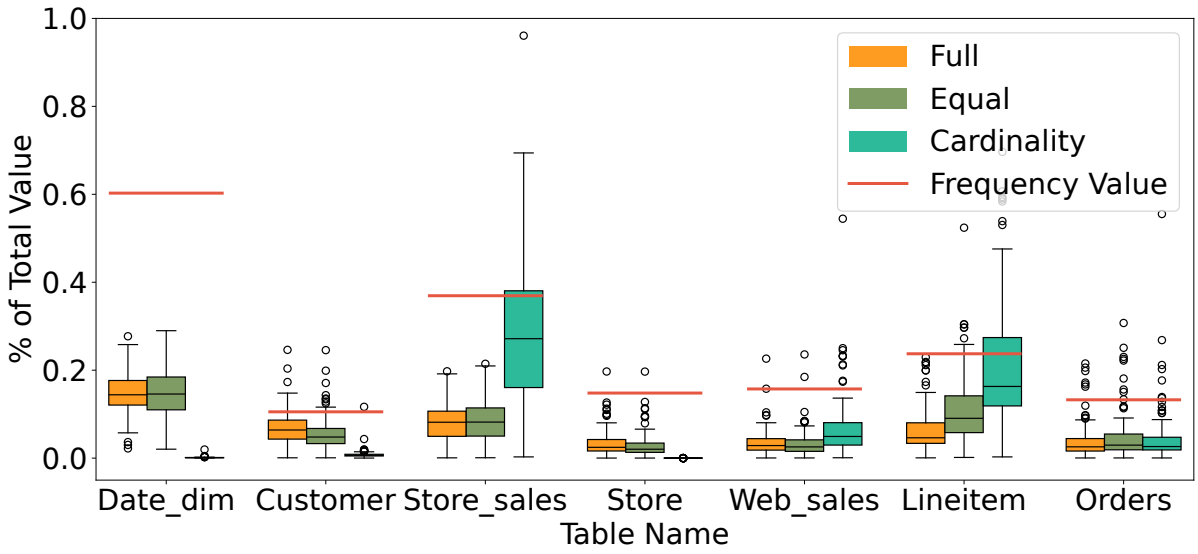


Figure 4.3: Distribution of Normalized Table Values Across 100 Draws of Query Values from Zipf Distribution at Table granularity. Compare the Full, Equal, or Cardinality division strategies. Red line is Frequency-based value assignment.

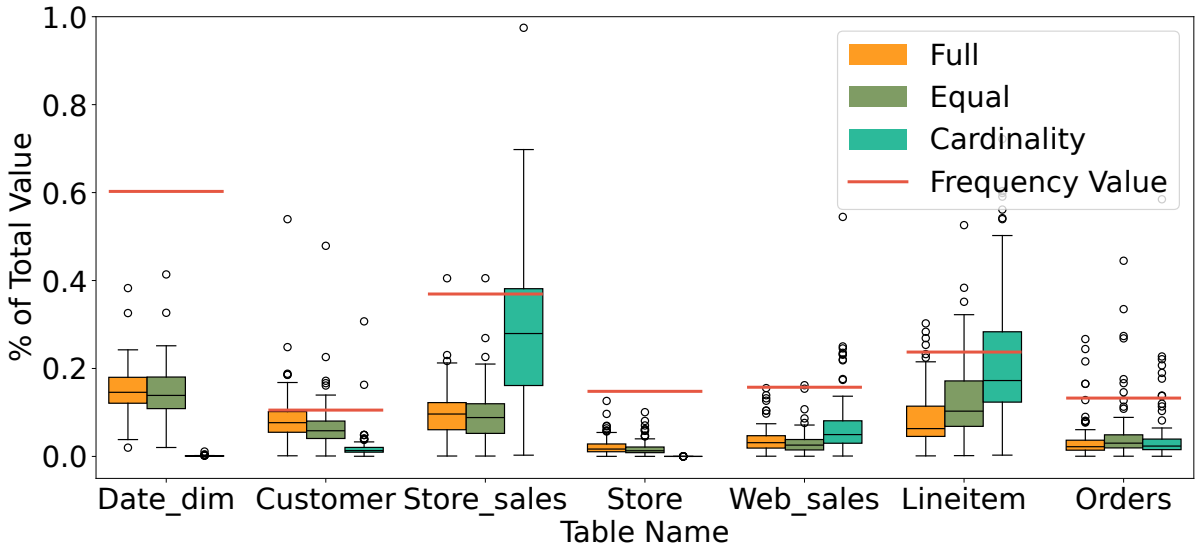


Figure 4.4: Distribution of Normalized Table Values Across 100 Draws of Query Values from Zipf Distribution at Column granularity. Compare the Full, Equal, or Cardinality division strategies. Red line is Frequency-based value assignment.

We supply the task value input to the indicator by randomly drawing task values from a Zipf distribution to simulate setups where data users assign a small number of tasks a much higher value than the rest. We study but omit drawing task values from a normal distribution, where task values have less variance, as we observe the same trends with less variance in data value. We execute 1500 queries drawn from the TPC-DS and TPC-H workloads and compute the value of each data element for each of the 100 different random draws from the Zipf distribution. We consider the **Table** and **Column** granularities, where tables or columns are data elements respectively, and compute the value of a table by summing the value for each data element in the table. We show normalized data value for each strategy and show a red line representing the frequency-based data value proposed by Glavic et. al. [Glavic et al., 2024] for the Table granularity in Figure 4.3 and the Column granularity in Figure 4.4. We show data for 8 of 32 tables for ease of visualization and because these reflect the trends observed across all tables.

We first note that the frequency-based model assigns a single data value regardless of the task values provided by data users, while our indicator takes on a full distribution of data values depending on the task values provided. We also see that frequency-based value is often significantly misaligned with our value of data indicator. In Figure 4.3 DATE_DIM is accessed by 60% of queries but attains only 15–18% of value on average under the Full or Equal dividing strategies. This is both because the tasks accessing DATE_DIM have a lower value and because some of these tasks also access many other tables, decreasing the value assigned to each table. As a result, using a frequency-based value would overvalue DATE_DIM, leading to poor data management decisions such as placing it in faster storage at the expense of more valuable data. This illustrates how a frequency-based approach could better capture the value of data if it accounted for the value of tasks to an organization.

When normalized, the Full and Equal strategies overlap significantly in both Figures 4.3 and 4.4. However, the scale of Full strategy grows as each task accesses more data elements,

while the total data value assigned under Equal (or Cardinality) will always match the total task value because the total value assigned to all relevant data elements by a task will be equal to the original task value under both the Equal and Cardinality strategies causing the differences we observe. The Cardinality strategy favors larger fact tables, like STORE_SALES, while DATE_DIM, a smaller table used by many queries, receives less value from each task. We see how, when moving to a Column granularity in Figure 4.4, these larger tables increase in value because the larger table size is amplified by the larger number of columns also accessed by tasks.

Moving from Table to Column granularity, we see some tables such as CUSTOMER increase in value across all three strategies while STORE decreased across all three, because either a larger or smaller number of their columns are actually being used. However, in DATE_DIM we see that the median value under Full shifts upwards slightly, while the median value under Equal shifts downwards. In Full, a task's value multiplies by the number of data elements it accesses. Under Column granularity, if high-value tasks predominantly access columns from one table, such as DATE_DIM, this will increase the normalized value under Full more than under Equal because a bigger share of the overall value is going to that table. For example, if Task A with value 100 accesses 5 columns and Task B with value 50 accesses 1 column, Task A accounts for 500 of 550 total value under Full, while under Equal it accounts for 100/150. If most of Task A goes to one table's columns, that will yield a higher value under Full than Equal.

4.4.2 Eliciting Task Values from Data Users

In this paper, we choose to elicit the task value input to the value of data indicator directly from data users who possess the needed context about organizational priorities and the impact of tasks.

If data users cannot provide any values, all tasks are valued equally, which reduces to

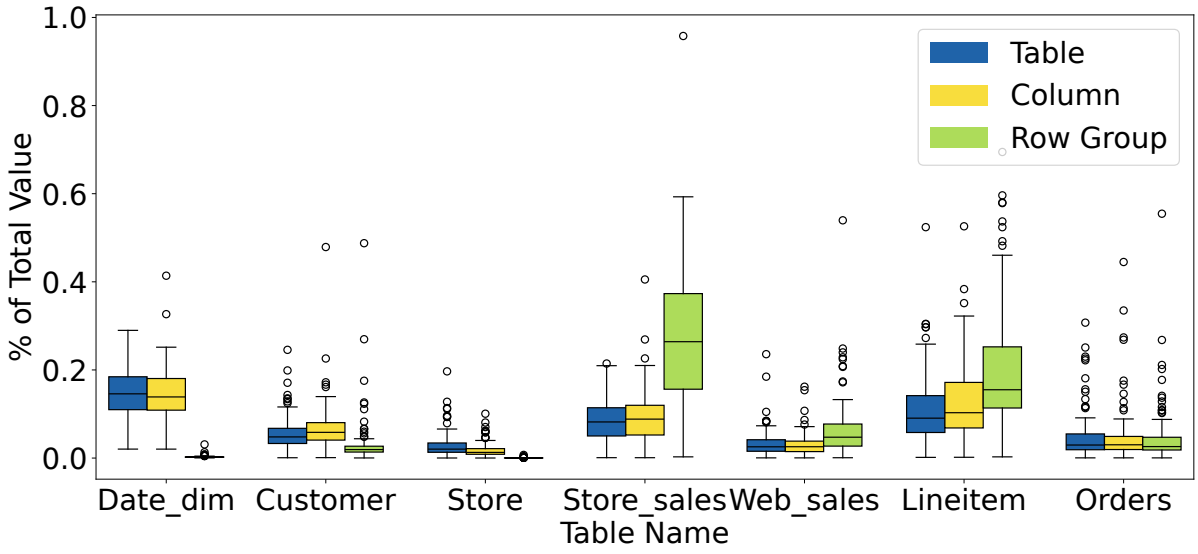


Figure 4.5: Distribution of Table Values Across 100 Draws of Query Values from Zipf Distribution shown for the Table, Column, and Row Group granularities.

a frequency-based approach, but data managers making data management decisions today often have implicit or explicit knowledge of the importance of some tasks. For instance, a data manager will know that a workflow supporting a critical product like fraud detection is more important than an internal dashboard used for exploratory analysis. This approach then systematically captures the context already possessed by data managers alone, and it also has the ability to capture the additional context of data users who have more insight on the importance of their tasks to broader organizational products and goals. These task values can be expressed in monetary units, like the revenue gained from a task, but can also be expressed in other units such as the increase in foot traffic to a store or engagement on social media.

Data users may have an initially incomplete understanding of task values but can improve value assignments over time. For instance, a data user may be able to value a well-known task like fraud detection but lack the same information for a new service. As data users gain more insight into the importance of different tasks–by studying the revenue generated by this new service, gaining insight from the developers, or speaking to leaders who understand

overarching goals–they can iteratively improve the values they assign to tasks which will better align the value of data indicator to organizational priorities.

4.4.3 How Granularity Impacts the Value of Data

In addition to task values, another input to the value of data is the choice of granularity, which defines how to logically partition data into data elements. We now empirically study the impact of this choice on the value of data.

We again use the TPC-DS and TPC-H workloads; because many queries in these workloads scan every row from a table, we author queries with additional filters to reduce selectivity on both datasets to illustrate how finer granularities can reveal what subsets of data generate the most value. The workload is 163 queries on 32 tables, and we use the Equal dividing strategy to compute data value. We define three granularities of data elements: **Table**, **Column**, and Apache Parquet **Row Group**.

We perform 100 random draws of query values from a Zipf distribution. We omit the normal distribution as we observe the same trends with less variance in data value. We execute 1500 queries drawn from the pool and show the computed value of a table by summing the value for each data element in the table. In Figure 4.5 we show normalized value for each granularity only so that outliers do not dominate the scale; however, all granularities operate on the same absolute scale because they use the same division strategy. We show data for 8 of 32 tables for ease of visualization and because these reflect the trends observed across all tables.

Data value will vary as granularity becomes finer, depending on how a table is used: if only a few columns or a small number of rows are accessed, then the column or row group granularities would show a lower value than the table granularity. On the other hand, if the majority of rows or columns accessed by tasks are from a single table, the column or row group granularity would show higher value than the table granularity because most of

a task's value is attributed to data elements within that table.

We see the first pattern in STORE and DATE_DIM, where the column value is lower than table, and row group is even lower because only a subset of columns is accessed. These tables are small, so larger fact tables like STORE_SALES dominate in the row group granularity because, if a task scans both DATE_DIM and STORE_SALES, its value is split evenly between the single row group in DATE_DIM and the 700 row groups in STORE_SALES. We see the latter pattern in the CUSTOMER table where a subset of row groups drives the table value, so the value under row group granularity is lower.

How granularity partitions data also impacts the value of data indicator. If granularity partitions data vertically, i.e., into columns, data value will not vary with table cardinality but with the number of columns, which is why DATE_DIM has a higher data value under column granularity than row group. If partitioned horizontally, i.e., by row groups, larger tables can dominate data value. Methods that divide task value unevenly among data elements could weight smaller tables more to mitigate this effect. Our indicator can support any uneven methods of dividing task value.

Figure 4.5 shows that finer granularities can help data managers prioritize the most beneficial subsets of tables in data management tasks such as caching or data cleaning, but we also see that coarser granularities still provide a useful, objective indicator of data value. We further illustrate this point in Section 4.6 by showing how the value of data indicator at a column granularity can guide data managers to better allocations for three data management tasks.

4.5 Implementing the Value of Data Indicator

Having now illustrated how data managers can access the value of data and how we compute the indicator using task values provided by data users, we now discuss how the indicator can be implemented by presenting a prototype implementation on Databricks' Unity Catalog.

This is in particular critical for capturing what data elements are relevant to each task. We illustrate that implementing the value of data indicator is feasible and does not require redesigning data systems from the ground up. We first present our implementation on Databricks, illustrating how we capture relevant data elements accessed by a task within the Unity Catalog and log the value of data indicator for table and column granularities. Then, to support the row group granularity, we modify the open source Apache Spark codebase, which is the underlying query engine for the Databricks data warehouse.

4.5.1 Unity Catalog-Based Implementation

The Unity Catalog [Unity Catalog] is a unified governance layer which manages data and enables users to run a wide range of tasks such as SQL queries and Python notebooks on that data. Databricks has open-sourced the Unity Catalog but also maintains a proprietary distribution which includes additional features. While primarily supporting analytical tasks on its Lakehouse, Databricks recently announced their Lakebase [Ghodsi et al., 2026] which are managed Postgres instances that better support transactional workloads, and any data updates or inserts on the Lakebase are automatically propagated to the tables managed by the Unity Catalog without manual ETL jobs and are accessible by analytical tasks such as SQL queries or Python notebooks. This presents a unified interface for data management and thus a promising system for implementing the value of data indicator.

Storing the Indicator. The two relational tables described in Section 4.2–the Data Value and Task Value tables–are created as tables within the Unity Catalog and are thus accessible to data managers through any interface to the Unity Catalog. This enables the management of these tables through the same interfaces, access controls, and permissions as other tables in the catalog.

Identifying Relevant Data Elements for a Task. Users can access data stored in the Unity Catalog through multiple interfaces. They can author SQL queries, run Python

notebooks using Spark's Python API, or even interact with the Databricks UI which triggers internal queries to run against the Unity Catalog, for example to show a table schema. Each of these, when run, is assigned a unique STATEMENT_ID. The QUERY_HISTORY system table stores a record for each task executed against the Unity Catalog, the time it executed, and the code for that task, among other metadata. Using either the Python code or SQL text, we can parse what tables and columns are read by a task and can use the STATEMENT_ID as the unique Task ID for the value of data indicator. Then, the Table Value table stores the value for a given task identified by its STATEMENT_ID. Because relevant data elements are parsed from the task code, this approach can support table and column granularities.

We run a periodic Databricks job which scans the QUERY_HISTORY table for new tasks, parses the relevant data elements from the task code, access the task value from the Task Value table, and then logs the relevant data elements and task value to the Data Value table. The cadence of this job can be tuned to balance the freshness of the indicator with the overhead of running the job.

Updating Task Values. Data users can access the Task Value table to identify what tasks have run and what operations they performed to assign values to these tasks. In instances where individuals have authored Python notebooks or SQL queries this may be easier to perform. However, in instances where downstream applications have run tasks on the Unity Catalog, those downstream applications will need to communicate to data users what tasks have run against the Unity Catalog based on user input and what operations those tasks performed so that users can assign task values.

4.5.2 Supporting Finer Granularities

We now present our implementation of the value of data indicator for table, column, and row group granularities on Apache Spark [Zaharia et al., 2010] which supports SQL queries over Parquet files and run through Spark's SQL interface. Apache Spark is the underlying

query engine for the Databricks Lakehouse.

Storing the Indicator. We implement the indicator's time-series data model, detailed in Section 4.2, as separate relational tables within Spark's catalog stored in Parquet files. Data managers can query the indicator directly or build views on it using Spark's SQL interface. If this implementation were integrated into the Unity Catalog, these same tables could be stored within the Unity Catalog itself.

QueryExecutionListener API. Spark exposes the QueryExecutionListener Java interface [Spark] which contains onSuccess and onFailure methods. We implement this interface and its methods and register it with the active Spark session. When a query or task completes, one of these two functions is invoked. These functions can run asynchronously from the main Spark thread and can be invoked by multiple threads. Within the QueryExecutionListener interface we determine what data elements were accessed and the timestamp of completion and append this information along with the provided task value to the interface tables within the Spark catalog. For each data element accessed by a task, the listener logs a new row in the Data Value table described in Section 4.2.

Granularity Support. Our implementation supports table and column granularity by parsing the SQL query and execution plan provided to the QueryExecutionListener to extract the relevant tables and columns accessed. However, Parquet is a columnar storage format internally organized into row groups, and the row groups accessed by a query are not accessible through the default information provided by the QueryExecutionListener.

To support the row group granularity, we modify the the open source Apache parquet-java repository[1], which Spark uses to read Parquet files and push-down filters to only scan relevant row groups. We maintain and expose which row groups were read by each file-read operation in a thread-local Map. This change modified about 40 lines of code.

1. https://github.com/apache/parquet-java

We then modify Spark's codebase[2] to access the relevant row groups and associate that information with the File Scan operator in the physical SparkPlan associated with the query's QueryExecution object. This modified about 400 lines of code. The result is that the QueryExecutionListener implementation can now determine what tables, columns, and row groups are accessed by a SQL query and log that information to the interface tables.

4.6 Evaluation

In this section, we answer the following research questions:

- **RQ1:** Can the value of data indicator be used as a decision-making signal to improve solutions to data management tasks?
- **RQ2:** How does the indicator implementation impact the performance of tasks? Does the performance impact vary based on the granularity of data element used?

To answer **RQ1**, we choose three data management tasks that illustrate how data managers can improve query performance or workload utility by prioritizing data based on the value of data indicator.

4.6.1 RQ1: Efficacy of the Value of Data Indicator

Data Placement

In cloud environments, data is often stored in blob storage, such as Amazon S3 [Amazon S3], and managed by a data lake [What is a Data Lake]. Query engines then fetch data from blob storage and place it on a smaller, local VM disk to execute queries [Kumar K M et al., 2024]. Because there is a time and monetary cost paid to fetch data from blob storage, and the local disk is often much smaller than the total data size, data managers must decide what data to place on the local disk. In this experiment, we observe how a placement strategy

2. https://github.com/apache/spark

based on the value of data can yield better outcomes for the organization compared to a more limited recency-based placement strategy.

Experimental Setup. In this experiment, we run the TPC-DS workload of 99 queries on 24 tables at 30GB stored within a single AWS S3 bucket. Each table in S3 is stored in its own Parquet file. After compression, the total size of the dataset is 11GB. We launch a t3.2xlarge EC2 instance with 8 vCPUs and 32GB of RAM with an attached disk. When queries need to run locally through DuckDB, any needed columns not present on local disk are fetched from S3. If the local disk cannot accommodate all columns to be fetched, some columns currently stored on local disk must be deleted. We compare two solutions to this problem: the Value of Data (VOD) strategy which deletes the lowest value columns when more data is required; and the least recently used (LRU) strategy, which deletes the least recently accessed columns from local disk. We choose LRU as a baseline because it is a well-known eviction strategy applicable in a range of similar data placement setups.

Each query is assigned a Task ID and a value per our model for the value of data, which is stored in the Task Value table described in Section 4.2. We assign values to each query through the following procedure. We construct three tiers of query values: low, medium, and high. We select one column as a high-value column, representing a scenario where query values are chosen based on what data is most important to an organization – for instance, queries that scan customer information are more valuable than queries that scan store location data. We then assign high value queries to be those that access this column, medium value queries to be those that access the columns we selected as medium value, and the rest as low value. There is 1 high-value query, 2 medium value queries, and 96 low-value queries. This represents a setup where a small set of queries access very important data and perform extremely business critical tasks, while the remaining queries are less important and have lower value.

We consider 5 different value assignments to study how query performance varies as task

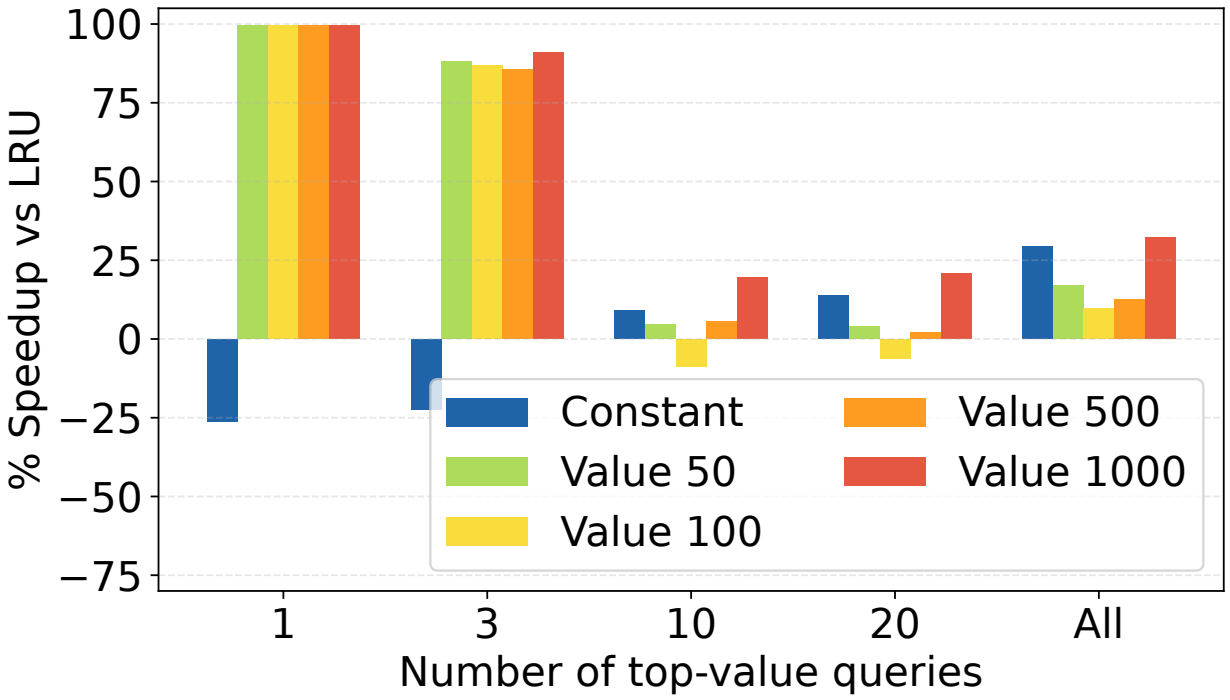
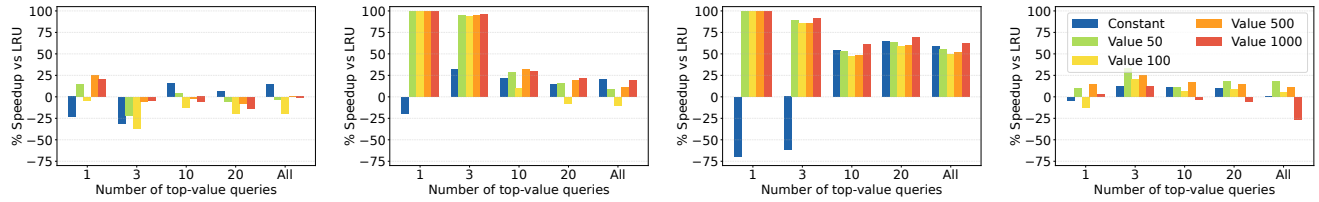


Figure 4.6: Speedup of Value of Data strategy over LRU for Top K valued queries with updating data values. Disk size 5.0GB.

values become more skewed. **Constant**, which assigns all queries value 1. **Value 50**, which assigns value 50 as the high value and 5 as medium. **Value 100**, which assigns value 100 as the high value and 10 as medium. **Value 500**, which assigns value 500 as the high value and 50 as medium. And finally **Value 1000**, which assigns value 1000 as the high value and 100 as medium. Low-value queries are assigned a value of 1 in all cases.

When a query runs, its value is accessed from the Task Value table and is divided equally among all columns it accesses. This is then logged to the Data Value table. Before the start of the experiment, we run an initialization phase which runs a set of queries from the TPC-DS pool, accesses the indicator interface to calculate the data value for each column, and populates local disk according to either the VOD or LRU strategy. To run the experiment, we run the entire TPC-DS workload once.

Local Disk Size. We consider 5 different sizes of local disk: 2.75GB, 3.5GB, 5GB, 7.5GB, and 13.75GB. The minimum disk size needed to run all queries is 2.68GB, so 2.75GB is



(a) Disk Size 2.75GB (b) Disk Size 3.5GB (c) Disk Size 7.5GB (d) Disk Size 13.75GB

Figure 4.7: Speedup of Value of Data strategy over LRU for Top K valued queries with updating data values.

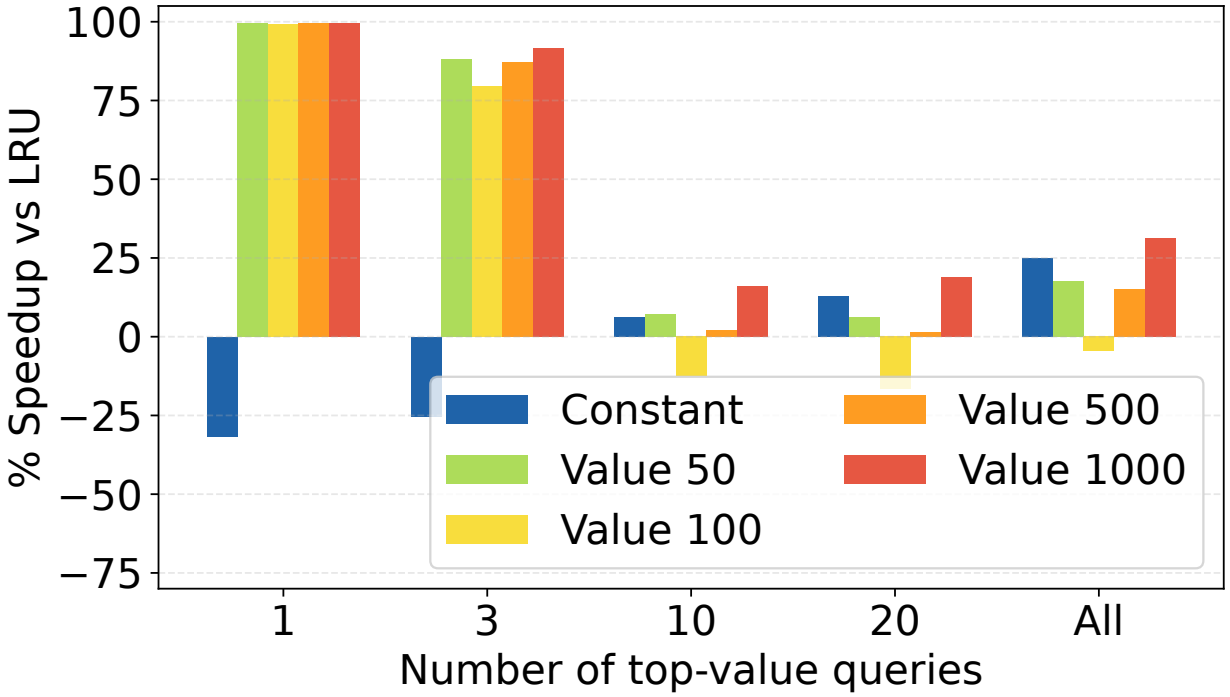
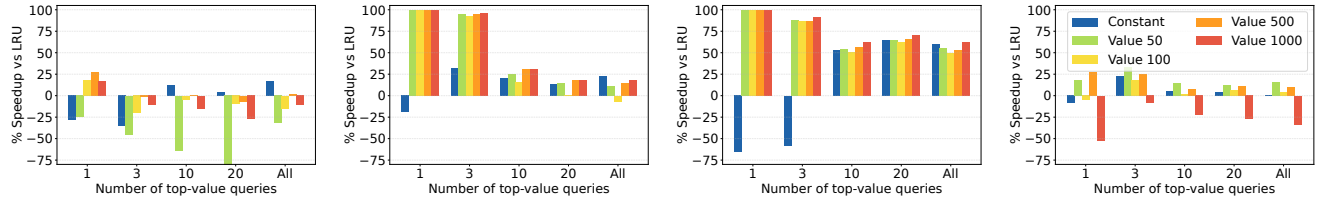


Figure 4.8: Speedup of Value of Data strategy over LRU for Top K valued queries with static data values. Disk size 5.0GB.



(a) Disk Size 2.75GB (b) Disk Size 3.5GB (c) Disk Size 7.5GB (d) Disk Size 13.75GB

Figure 4.9: Speedup of Value of Data strategy over LRU for Top K valued queries with static data values.

among the smallest feasible disk sizes possible, while 13.75GB is 125% larger than the total dataset size (11GB), so all data can fit in local disk. We vary disk size between 2.75 and 13.75 to observe how these strategies adapt in both constrained and unconstrained disk setups. We compare the query runtimes for TPC-DS under the LRU and VOD strategies for each disk size and value assignment.

Placement Results: Updating Data Values. In this setup, each task that runs during the experiment has an assigned value and updates the Data Value table in real time as it runs, so data value reflects the most up-to-date information.

We first show the percent speedup of the VOD versus LRU strategy across all value assignments when disk size is 5.0GB, or about 50% of the total dataset size, in Figure 4.6. We look at speedup for the top 1, 3, 10, and 20 queries by value as well as all 99 queries. Top 1 is the single highest value query and top 3 also includes the 2 medium value queries.

The first thing we note is that in the Constant assignment the top value query chosen is the same as the other assignments, but of course this has no particular prioritization by either strategy in this case. Hence, Constant is slower on VOD than LRU for the high and medium queries, as neither metric prioritizes those 1-3 queries over others, and data for those columns has been used more recently when the queries execute, but are less frequently used columns so are deprioritized by VOD. The Constant assignment more closely resembles the least frequently used (LFU) strategy, as data value is tied to how frequently data is accessed. Having run queries before the experiment, using the frequency of access still provides a benefit over LRU, as seen in the positive speedup for all queries, illustrating how even when task values are not skewed or data managers lack information on how valuable queries are, the value of data strategy can improve query performance.

The VOD strategy achieves significant speedups for the high and medium queries of nearly 100% for value assignments other than Constant. We note that in scenarios where a single high-value query is worth 50-1000× more than the 96 other queries in a workload,

that speedup is proportionally more significant. However, the disk size of 5.0GB is still small enough that the VOD strategy favors these high value queries over the majority of low value queries, explaining why the overall speedup is under 25%, and speedup declines significantly for larger subsets of queries.

In Figure 4.6 we note that Value 100 performs worse than all other value assignments for the top 10 and 20 queries and all queries. At Value 50, high and medium queries are prioritized enough from task value skew to realize significant speedup. While high-value data elements remain at the top of the priority list, the ranking of data elements accessed by medium and low value queries is more impacted by the frequency of access because the gap between medium and low values is 10× smaller. At Value 50, this gap is small enough that after the high value data elements, the remainder hold a similar ranking to the Constant strategy. At Value 500, the high and medium value data elements are firmly at the top of the priority list, while the rest are ranked by access frequency, as all tasks accessing them have the same value. However, Value 100 changed the ordering of the medium-accessed data elements in a way which evicted data elements for long-running low-value tasks, lowering speedup. Additionally, runtime variance from DuckDB may have contributed to the difference, as some queries that ran slower on VOD had very low absolute runtimes, making them more susceptible to that variance.

In Figures 4.7a–4.7d we show the same speedup across disk sizes ranging from 2.75GB to 13.75GB. At 2.75GB disk size in Figure 4.7a, we see a less than 25% speedup on the highest value query for all value assignments. This is because the local disk is only 70MB larger than the required data size for some queries, requiring that even very high-value columns be evicted to make room for the data needed to run the query. Given the small disk size, the VOD strategy also disadvantages lower value queries, which access more frequently used columns, explaining the slowdown of VOD for the top 10 and 20 queries. Across larger subsets of queries or all queries, we see that the speedup trends to 0, as both strategies are

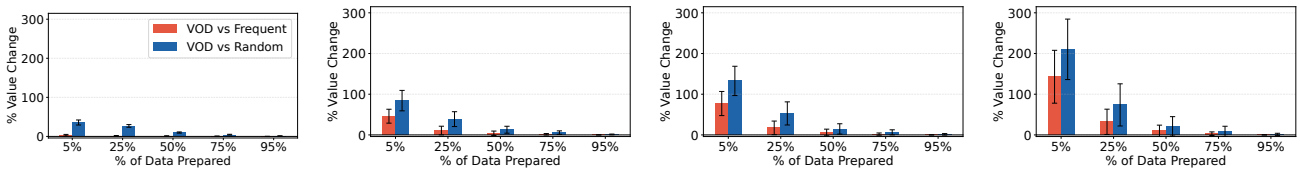
forced to evict either high-value or recently used data given the small disk size.

At 3.5GB disk size, there is enough available space to avoid evicting high-value data, leading to nearly 100% speedup on the highest value query, which we see repeated at 7.5GB disk size as well. However, the disk size is not big enough at 3.5GB to maintain frequently accessed but low-value data, so overall speedup remains under 20%. The speedup at 7.5GB grows for lower value queries while maintaining the speedup for high and medium queries. With the extra disk space, the VOD indicator now begins storing data that is frequently accessed by low-value queries. The indicator adapts to the skewed data values so that when disk space is less constrained, it does not lag behind frequency-based strategies.

Finally, at 13.75GB disk size, both strategies can store the entire dataset on local disk. Because of the initial phase all data is stored on disk at the beginning of the experiment, so there is no difference in execution between the two strategies. Some runtime variance exists from DuckDB or the hardware environment, leading to some fairly significant runtime differences between the two strategies. However, this is not due to different placement decisions, just differences in query execution times. The speedup or slowdown is much smaller in magnitude compared to smaller disk sizes and tends to 0 over the whole workload.

Placement Results: Static Data Values. Updating the Data Value interface table in real time may add extra overhead that data managers do not wish to pay while a workload executes. Instead, they may opt for data values to be updated in between workload executions so the values computed in the previous iteration are used for the next. We show the percent speedup of the VOD strategy over LRU for this scenario across the same 5 value assignments on a 5.0GB disk size in Figure 4.8 and on the other disk sizes in Figures 4.9a–4.9d.

We see overall that trends are similar to when data value is updated in real time, but speedups are slightly less significant–for instance the speedup on the high value query at 2.75GB disk size is 20% when data values are updated but is 17% when data values are static. Further, speedup for all queries for Value 1000 is near 0 when values are updated



(a) Constant (b) Value 50 (c) Value 100 (d) Value 500

Figure 4.10: Percent difference in workload value for Value of Data versus the Frequent and Random strategies versus percent of data elements prepared for each of the 5 different value assignment methods.

but are actually slower on VOD vs LRU for static values. This is because when data values are updated in real time, the indicator better adapts to the query workload running which reinforces the values assigned from the initial phase. As such, data managers can still achieve significant speedups on the most important tasks even when not using the most up-to-date data values possible.

Data Preparation

Data analysts can perform a range of preparation tasks on both structured and unstructured data elements, such as deduplicating data, converting file formats like PDF to text, replacing NULL data with defaults, or chunking large files into smaller pieces. However, analysts often have a limited budget for how many data elements they can prepare, and a task's value can improve if the data elements it accesses are prepared. For instance, if removing outliers improves a prediction task's accuracy, that would increase the task's value as it better fills the information need. In this experiment, we compare different strategies that prioritize what data to prepare to most benefit the organization, which we model as workload value gained.

Experimental Setup. In this experiment, we construct a set of data elements, a set of tasks, and a mapping of what data elements each task accesses to simulate a data management setup.

We consider a set of 99 data tasks with 425 data elements based on the TPC-DS workload

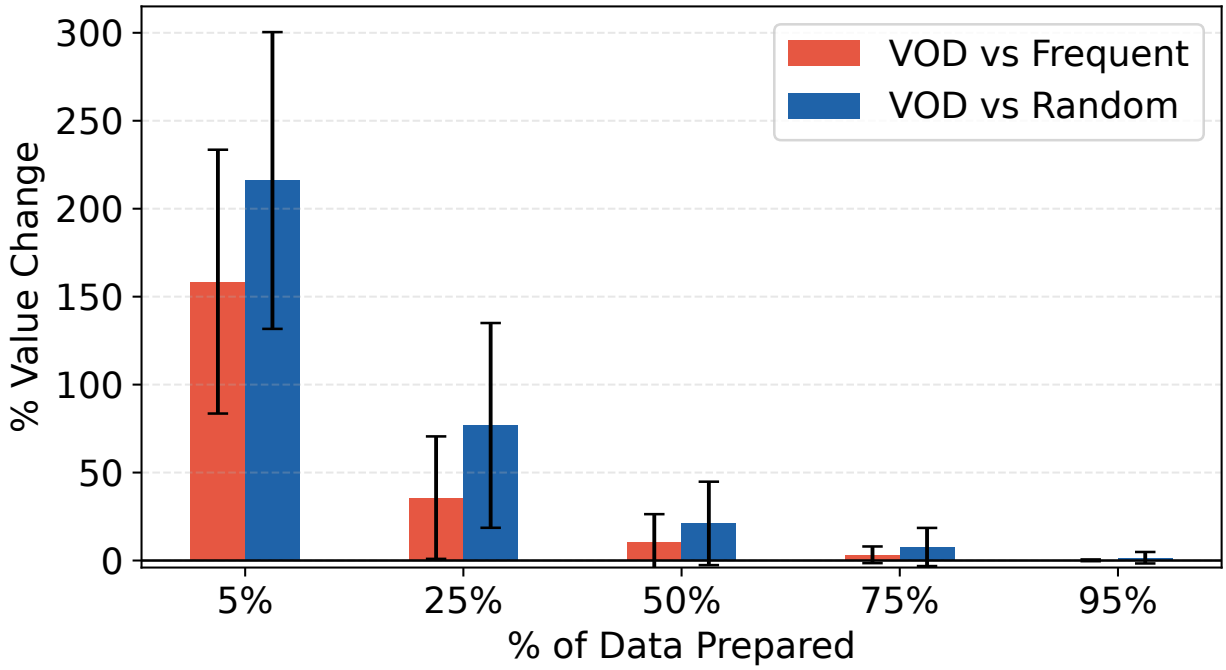


Figure 4.11: Percent difference in workload value for Value of Data over Frequent and Random strategies versus percent of data elements prepared for the most skewed value assignment, Value 1000.

which has 99 queries and 425 columns. Each task accesses some number of data elements. We construct a list of how many columns are accessed by each TPC-DS query and select uniformly at random from this list for the number of data elements a given task will access. We then select that number of data elements uniformly at random from the pool of 425 data elements. Multiple tasks can access the same data element.

Each task also has a Task ID and a value assigned to it. We use the same value assignments as in the data placement experiment: Constant, Value 50, Value 100, Value 500, and Value 1000. These are accessed through the Data Task indicator table.

Each task is also assigned uniformly at random one of three value scaling functions which define what fraction of task value is realized based on what fraction of relevant data elements are prepared. We define three classes of scaling functions–those with high sensitivity to preparation, medium sensitivity, and low sensitivity–to model how real workloads have tasks with varying levels of dependency on data quality. High-sensitivity tasks realize full value

only if all data elements are prepared, like machine learning tasks sensitive to training data quality. Medium sensitivity tasks realize full value if 50% of data elements are prepared, such as business intelligence dashboards which can still show valid trends with unprepared data. Low sensitivity tasks realize full value if 25% of data elements are prepared, such as data exploration tasks which can still provide some insights even with a small amount of prepared data. If no data elements are prepared, the scaling result is 0.2 to represent some minimum gain from executing the task. Each value scaling function increases linearly from the minimum to the full scaling factor.

Given a set of tasks, task values, and relevant data elements to these tasks as well as a budget of what percent of the total data element pool can be prepared, we evaluate three strategies for choosing what data to prepare. First, a Random strategy which prepares data elements uniformly at random. Second, the Frequent strategy which prepares the most frequently accessed data elements. This is equivalent to all queries having the same value. And third, the value of data (VOD) strategy that prepares the highest value data elements. We vary the budget from 5% to 95%. We randomly generate the tasks–which include what data elements each task accesses–100 times. We run each task 10 times prior to the experiment and divide the full value of the task equally among all data elements it accesses to compute the value for each data element.

Metric. Given some subset of data elements that are prepared, we compute workload value as the sum over all tasks of the value scaling result of a task multiplied by the task's value.

Preparation Results. In Figures 4.10a–4.10d and 4.11, we show for each value assignment the percent difference in workload value for the VOD strategy over the Random and Frequent strategies as the budget increases, showing the mean percent difference with error bars showing the standard deviation across the 100 random generations of tasks.

Unsurprisingly, the Constant value assignment in Figure 4.10a sees no difference between the VOD and Frequent strategies as these are the same strategy when all task values are

the same. There are moderate gains over Random as picking the most frequently accessed data elements will still improve value for the most tasks. However, as the budget gets larger in all experimental setups, the gains become less significant as more data elements can be prepared, so the difference in strategies becomes less significant.

As data becomes more skewed in Figures 4.10b–4.10d, we see steadily increasing gains of the VOD strategy over the Frequent and Random strategies. The VOD strategy prioritizes elements accessed by tasks with the highest potential for value gain, while prioritizing the data accessed most frequently may not yield better value for the most important tasks in a workload. As task values become more skewed, the Frequent strategy prioritizes data accessed by less valuable tasks, so the overall workload value gains of VOD become more significant. The VOD strategy also simply prioritizes data used by high and medium tasks first.

However, we note that in Figure 4.10b, where the high value is 50 and medium value is 5, some data elements are accessed by low value tasks frequently enough that they outweigh medium value data elements and still achieve 40% gain over Frequent strategy. We see that in moderately skewed value scenarios, the value of data indicator factors in both frequency and task value to determine a better way to prioritize data elements.

Data Sorting

We now consider how the value of data can guide a data manager to pick a modification to a data setup to reduce costs and improve the utility of a workload. In a cloud setup, bytes scanned directly translate to monetary costs–for example, BigQuery charges $6.25 per TB scanned by a query [Cloud, 2026b] and AWS charges $90 per TB for egress [AWS S3 Pricing]. In this experiment, we consider how a data manager can use the value of data indicator to reduce IO costs by storing the rows of a relational table in sorted order by a set of columns, like the clustering approach taken by Snowflake [Snowflake, 2026] and BigQuery [Cloud,

2026a]. We observe how a strategy using the value of data indicator can identify a more effective setup modification than a simpler frequency-based strategy.

We consider the notion of *utility*, which factors in both task value and costs, and consider a situation where a task's utility is directly proportional to the reduction in IO costs and where some tasks are far more sensitive to costs than others. This may be because an organization has created multiple BigQuery projects each with its own budget, and some teams have tighter budgets [Williams and Joshi, 2023]. This may be because some tasks access more expensive data, such as real-time market data like Bloomberg B-Pipe [Bloomberg B-PIPE]. Or this may be because some tasks access data stored in different network regions or clouds and incur network transfer fees on top of query costs [Google Cloud Network Pricing]. In these instances, reducing IO costs for some tasks is more valuable than the same reduction for other tasks. We evaluate how a value of data strategy to choose how to sort data can improve workload utility and compare this to a frequency-based approach.

Experimental Setup. In this experiment, we sort data on the TPC-DS 30GB dataset. TPC-DS contains 99 queries and 24 tables. We write each table as a set of equally sized Parquet files, where each file contains 32 row groups of 128,000 rows each. We choose 128,000 rows per row group, as this is the default setting in DuckDB, and 32 row groups per file, as this is a common setting for Parquet files.

Parquet files contain metadata for each row group which includes the minimum and maximum value for each column in that row group. We select two columns from a table and physically sort the rows in the table by those columns in lexicographic order. In this experiment, we define a budget where only 2 tables may be sorted and on only 2 columns per table. We evaluate three strategies for choosing what tables and columns to sort on. The Value of Data (VOD) strategy, where we select the highest value tables and the highest value columns in those tables using data from the interface table. The Frequent strategy, where we select tables and columns by which are most frequently accessed. And a Random

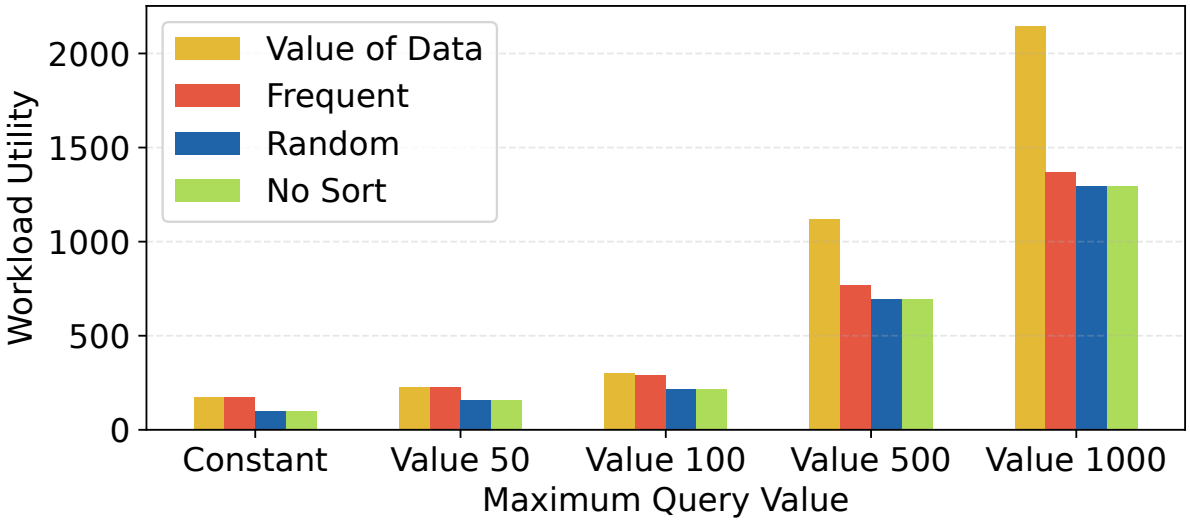


Figure 4.12: Total workload utility of VOD strategy versus Random and Frequent across 5 different task value assignments for choosing how to sort data on TPC-DS (30GB)

strategy, where we select tables and then columns uniformly at random.

We consider the same 5 value assignments as in the previous experiments: Constant, Value 50, Value 100, Value 500, and Value 1000. We run a trace of queries from TPC-DS, and we compute the value of data indicator as well as the frequency of access for each column. For each of the three strategies, we execute the TPC-DS workload once in DuckDB with profiling enabled, which measures the total bytes read during query execution.

Metric. We compare the bytes read for each query under each strategy and compare it to the bytes read by the query when run on unsorted data. For each strategy, we multiply each query's value by $\frac{\text{plain bytes read}}{\text{sorted bytes read}}$. We sum this result over all queries to yield an overall workload utility for each strategy.

This captures the scenario discussed above, where reducing bytes read increases the utility proportionally, and consequently increasing bytes read would reduce utility.

Results. We show the workload utility for each strategy across each value assignment in Figure 4.12. We also show the baseline utility without sorting data, labeled No Sort.

Across all value assignments, Random provides a negligible utility gain over the baseline,

on the order of 0.01% gain, as its choice for tables to sort on is unlikely to meaningfully reduce bytes read for any query, much less high-value queries.

In the Constant assignment, VOD and Frequent make exactly the same sorting choice because these are the cases when the strategies perfectly align. For Value 50, the next least skewed assignment, the VOD and Frequent strategies choose the same table and columns for one table, but differ on the second table. Introducing some skew in task values causes the VOD to prioritize a different table, but this table reduces bytes scanned with a similar impact to the sort order chosen by the Frequent strategy, so the workload utility achieved differs negligibly. Both strategies outperform Random and No Sort–72% increase for Constant and 46% for Value 50.

Value 100 introduces more significant skew in task values, so VOD and Frequent differ on both tables chosen. VOD increasingly prioritizes IO reduction for the high and medium value queries over the low value queries. This causes a large gain in workload utility, as an even modest reduction in bytes for a query with high sensitivity to IO costs will yield a significant increase in utility.

On Value 100 VOD utility is 4.4% higher than Frequent and 40% higher than Random and No Index, while Frequent sees a 33% gain over Random and No Sort, illustrating that using an objective indicator is still far better than using no indicator at all.

For the last two value assignments, when the gap between the high and low value queries grows to 500 or 1000, the VOD strategy chooses tables which, in addition to differing entirely from the Frequent strategy, dramatically improve IO performance for high and medium value queries, which have a much more significant impact on overall workload utility. While this yields the same or even worse IO performance for some low-value queries, this strategy favors reducing costs for the most cost-sensitive tasks.

The value of data indicator adapts to varying levels of skew in the workload, matching the gain over the No Sort baseline when it is identical to the Frequent strategy and adapting

to information about the variance in task values to achieve increasingly large utility gains over both the baseline and the Frequent strategy, which cannot react to that same variation in task values.

4.6.2 RQ2: Overhead

Overview. In this experiment, we evaluate the overhead on query runtime for our implementation of the value of data indicator on Apache Spark using the QueryExecutionListener API and study how query overhead is impacted by data element granularity.

Experimental Setup. To answer **RQ2**, we run all TPC-H queries and measure their runtime in the Spark implementation utilizing the QueryExecutionListener API as described in Section 4.5. We run TPC-H against implementations which track the value of data at the column granularity – we call this the **Column** implementation – as well as the **Row Group** granularity. We omit **Table** as it performs the same actions as **Column** with no difference in overhead. Additionally, we measure the runtime of each TPC-H query under a **Vanilla** Spark setup with no registered QueryExecutionListener and a Spark setup which registers an empty QueryExecutionListener to execute – which we label as **Listener Only**. This allows us to observe the total overhead added by our implementation over a Vanilla Spark setup and to isolate how much of that overhead is added by the QueryExecutionListener interface itself. Finally, as the row group granularity implementation requires modifying Spark and the Parquet-Java library, we compare the runtimes of TPC-H on Vanilla Spark setups running both the modified and unmodified Spark binaries.

We run all experiments on a 1-node or 4-node cluster of nodes with Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz and 64GB RAM. We run queries against a 1TB TPC-H dataset. Each query is run 3 times, and we present the mean runtimes.

Overhead vs. Vanilla Runtime. In Figure 4.13 we show for each TPC-H query the average runtime in minutes for each of the 4 implementations evaluated on a 1-node cluster.

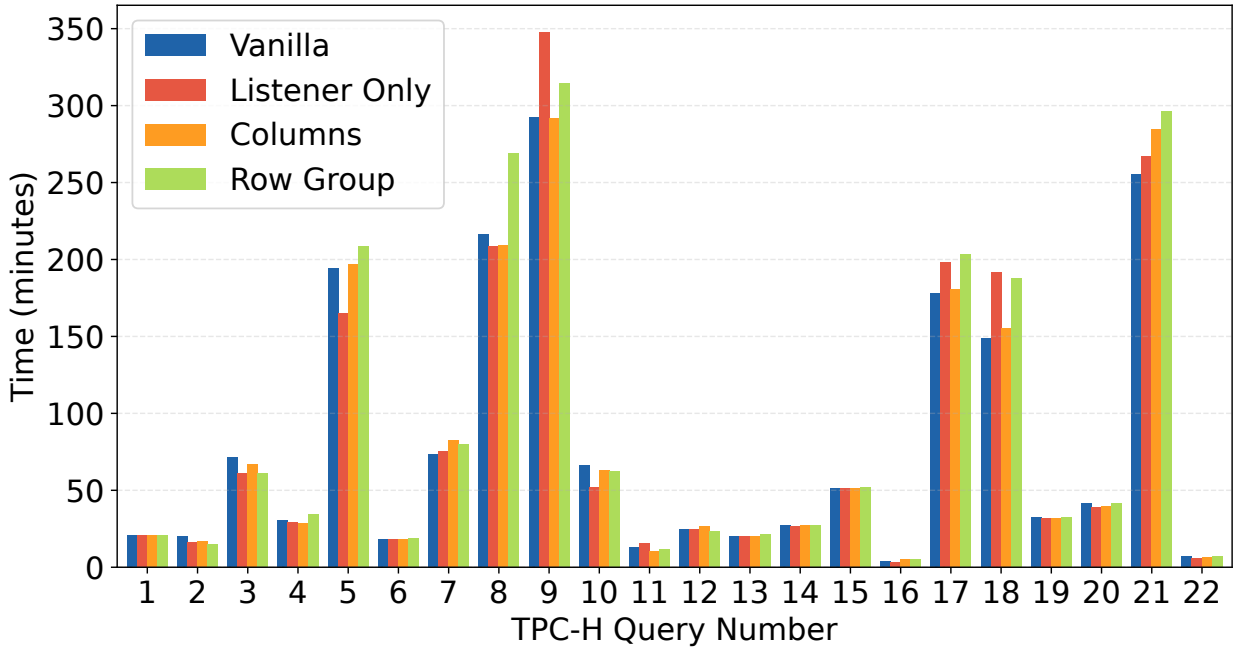


Figure 4.13: Comparing Vanilla, Register, Column, and Rowgroup implementations on TPC-H 1TB on 1-node Spark

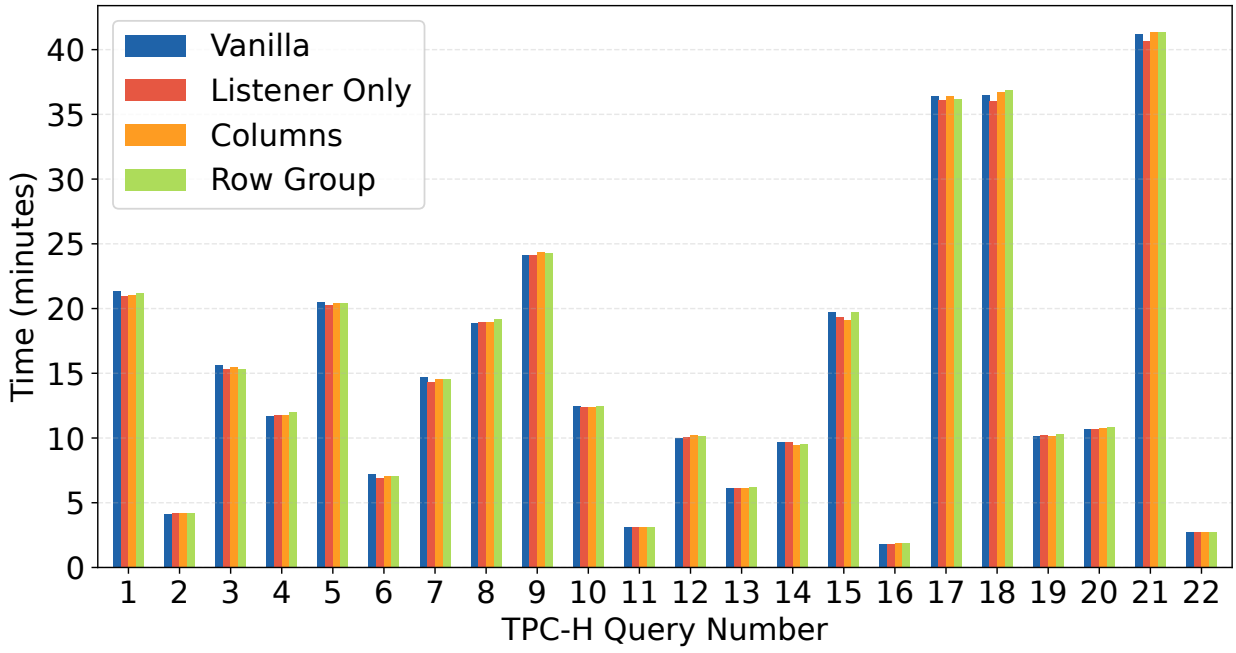


Figure 4.14: Comparing Vanilla, Register, Column, and Rowgroup implementations on TPC-H 1TB on 4-node Spark

We show the same on a 4-node cluster in Figure 4.14. For most queries, the overhead of both Column and Row Group indicator implementations are minimal, around 1% and 5% on average, respectively over the Vanilla implementation. As we see in the 1-node 1TB setup in Figure 4.13, much of the overhead of these implementations is accounted for by the Listener API, which we can see from the Listener Only column.

On 1-node, we see that there are some queries where the Row Group implementation has significant overhead over the Column implementation, with the highest at 28%. These queries access far more row groups than columns, so tracking the indicator at a finer granularity requires that value be computed for more data elements, increasing the number of rows written to the Data Value table and increasing runtime overhead. However, on a 4-node cluster this maximum overhead narrows to 3.22%, with a mean difference between Column and Row Group of 0.55%. This is because the QueryExecutionListener API is implemented as an asynchronous callback function that runs on query completion. On 4-nodes with more available CPU, the callback function executes in parallel while query results are returned, reducing the impact on the critical path of query execution. The extra resources also speed up the actual value of data logging function, further narrowing overhead.

As such, on a 4-node cluster the overall overhead narrows as well. Column achieves on average a 0.06% overhead with a maximum query overhead of 4.6% over Vanilla. Row Group achieves on average a 0.82% overhead with a maximum query overhead of 2.38% over Vanilla. Overall, we see that though finer granularities can introduce more overhead, primarily through the IO cost of writing more rows to the Data Value table, much of the overhead introduced is accounted for by the QueryExecutionListener API alone, and the remaining overhead is mitigated in larger cluster setups with more available RAM and CPUs.

Finally, we see some instances where the Vanilla implementation is unintuitively slower than other implementations. After rerunning these queries, we believe this is due to variance in Spark and JVM execution times rather than due to a systematic issue with any implemen-

tations, as well as potentially the more limited CPU and RAM resources on a 1-node cluster. When we move to a 4-node cluster in Figure 4.14, we see that though some of these unintuitive instances still exist, the margins of difference between the different implementations is far smaller.

Impact of Modified Spark Implementation. Finally, we evaluate the impact of our modifications to the Spark implementation on query performance. Spark out of the box does not expose what row groups are accessed, so we modified Spark and the Parquet-Java library it uses as a Parquet reader to expose this information. We compare the Vanilla implementation on Modified Spark and Unmodified Spark on TPC-H at 30GB and 1TB for a 1-node cluster. We observe that there is on average about a 4% overhead with most queries showing a percent difference near 0. There are some higher overheads and some negative overheads, i.e., in some cases the modified implementation is faster. This is likely simply a feature of runtime variance in Spark and JVM execution. There are two queries with longer runtimes and more significant overheads, near 20 or 35% respectively, that appear to be due to high parallelism in reads. Information about row group access is stored in a thread-local Map within the custom Parquet reader library, so each thread can read what row groups were accessed and store that somewhere accessible to the QueryExecutionListener. In the case of more parallel reads, the thread-local Map is growing larger and creating contention issues, adding to runtime overhead. However, this overhead does not grow when data size grows, indicating that this is a roughly constant factor.

4.7 Related Work

Other Methods to Value Data. Work on the instrumental value of data by Glavic et. al. [Glavic et al., 2024], which we discuss in this paper, defines the value of data by relative access frequency rather than using task values provided by data users. Related work in accounting information systems, which are systems that capture financial and accounting data

for an organization and expose this to decision makers, focuses on determining if capturing a new piece of data in an information system would better serve decision makers and does so by modeling the value of this potential new data to those decision makers [Feltham, 1968, Gallagher, 1974, Ahituv, 1980, Wand and Wang, 1996]. This work presents empirical methods to guide evolving information systems to better fit the needs of decision makers, while our work in this paper focuses on computing an indicator for the value of data to an organization under a given information system and set of tasks to better serve data managers.

Contribution of Data to a Task. Data Shapley methods [Ghorbani and Zou, 2019, Bertossi et al., 2023] focus on determining the contribution of data to a task. Work in fair division of costs in federated learning [Murhekar et al., 2025] also measures the contribution of data to a goal. These methods can complement our work and inform how our value of data indicator divides task value between relevant data elements.

Capturing Relevant Data. A key input of the value of data indicator is the relevance of data to tasks. Related work focuses on refining the notion of relevance to understand precisely what data impacts the results of a query using provenance sketches [Niu et al., 2021] which is itself related to data summarization [Glavic et al., 2021], or query resilience [Freire et al., 2015]. These works complement our own, as our value of data indicator can use these methods to determine data relevance to tasks, but our current implementation captures data access patterns after predicates are applied.

The Value of Data in Automated Database Management. We note that there are many well-known problems and solutions that use the value of data, if implicitly, to guide their decisions, which we now outline. The AutoAdmin project [Chaudhuri et al., 2011] from Microsoft Research built tools that aimed to automate database administration tasks. For example, self-tuning database systems [Chaudhuri and Narasayya, 2007] use the frequency of data access to determine the potential cost and benefit of modifying physical design and building a new index; other work concerns building tools to automatically guide and

recommend index selection [Chaudhuri and Narasayya, 1998, 1999, Das et al., 2019] and index tuning on a budget [Wang et al., 2025, 2024], which similarly aims to efficiently identify the most valuable candidate indexes.

The AutoAdmin project also published work for automated view materialization and index selection, which again aim to identify valuable data or intermediate data products to store. Other work builds hybrid columnstore and B+-tree physical designs to enable more efficient query processing for transactional and analytical workloads. This work again aims to identify scenarios where data is highly valuable to both types of workloads and builds a physical design to support that. Data managers may similarly wish to identify scenarios where a single physical design would maintain data's utility just as well.

The Value of Data in Database-as-a-service Systems. Similarly, work from Microsoft Research concerning resource allocation to multi-tenant database-as-a-service platforms [Arora et al., 2023, König et al., 2023] aims to identify efficient allocations of resources to tenants to improve performance for key tasks with minimal costs. Other work focuses on performing top-K aggregations over large datasets while reducing data movements [Siddiqui et al., 2023], which itself aims to identify what data to keep in limited cache space during query processing by identifying how to minimize costs, and thus improve utility.

Value of Data in Other Existing Problems. Other well-known algorithms proposed similarly can be viewed in the context of the value of data. The PageRank algorithm [Brin and Page, 1998] defines the value of data as the importance of a web page based on its links to other pages and uses this value to rank search results. The five-minute rule [Gray and Putzolu, 1987] defines the value of a data page as the cost to hold it in memory compared to the cost to repeatedly access it from disk over a period of time. It then uses this value to decide what data to keep in memory. Though not originally expressed in terms of data value and task value, these works can be re-evaluated in these terms and are potential applications of our value of data indicator.

4.8 Conclusion

This paper makes two key claims: first, that data managers could make better data management decisions with access to an indicator for the value of data, and second that this indicator must be computed using values on tasks provided by data managers and data users to capture the vital organizational context that more limited approaches fail to capture. We present an interface through which data managers can access and use the value of data indicator, explain how we compute the value of data indicator, and provide an implementation of the value of data indicator on Databricks' Unity Catalog and Apache Spark. We illustrate how the significant benefit this indicator can provide to data managers compensates for the overhead in query performance.

We hope this work encourages further work in implementing the value of data indicator in real data systems, and we hope this work motivates data system designers and organizations to think critically about how best to capture the value of data in their systems and expose it to data managers, as this interface will be crucial to the adoption of the indicator.

CHAPTER 5
GAINING INSIGHT INTO SPECIFIC DATA MANAGEMENT SCENARIOS

5.1 Introduction

In this chapter, we present two works that enabled us to gain deeper insights into data management problems in two scenarios. We discuss these problems in the context of data sharing and building a scheduler for a serverless data lakehouse. We first introduce *Eudoxia* [Srivastava et al., 2025], a simulator that evaluates how different scheduling algorithms would impact a function-as-a-service(FaaS) data lakehouse. Then, we discuss a paper on *Programmable Dataflows* [Xia et al., 2024], which introduces a data sharing abstraction called the *contract* that facilitates users to more easily share data.

5.2 Eudoxia: a FaaS scheduling simulator for the composable lakehouse

Bauplan [Bauplan Labs] is a composable data lakehouse that provides a function-as-a-service (FaaS) runtime for SQL queries and Python pipelines. This system decomposes user tasks into a directed, acyclic graph (DAG) of functions, assigns each function some resources, and executes the functions on cloud infrastructure. However, system designers must make a key data management decision: how should the limited computing and memory resources be allocated to each function, and how should the functions be scheduled for execution?

Unit or integration tests are case-based methods for evaluating how a given scheduling algorithm performs on workloads, while utilizing more general formal methods can be far more expensive and difficult to implement. As a result, we take a scheduling approach that relies on a deterministic model of the system like formal methods but remains cheaper and

experimentally driven like integration testing. We introduce *Eudoxia*, a simulator for the Bauplan system that allows users to implement and register arbitrary scheduling algorithms and to configure a wide range of parameters. *Eudoxia* then simulates workloads arriving to the system and how each scheduling algorithm interacts with these workloads and assigns physical resources to them, and the simulator tracks execution metrics like latency for each task, resource utilization, and throughput. The system also supports schedulers spinning up new resources to meet demand for additional cost and tracks the additional costs incurred.

Eudoxia enables system designers to evaluate candidate solutions to this key data management problem more easily and feasibly and provides key insights into some concrete challenges faced by this set of data managers.

5.3 Programmable Dataflows: Abstraction and Programming Model for Data Sharing

Data can become more valuable when combined with other data; for instance, banks may combine data to improve fraud detection or hospitals may combine data to improve patient care. Sharing data can increase the quality of a task's output or improve how well it fills an information need, thus increasing that task's value. However, data sharing is rare in practice because of the risks of leaked data, which could include the loss of competitiveness or violating privacy constraints and regulations. A key data management problem then is identifying whether sharing data with another set of agents to improve a task's outcome can be done without violating agent constraints, what the potential risks are of sharing data, and whether the benefits of sharing data outweigh these risks. Solutions exist that allow agents to run the desired task on the pooled data while ensuring some leakage constraints are met, but these solutions are generally cost-intensive and difficult to adopt in large part because agents lack the abstractions to understand the risks and value of sharing data with other agents.

In this paper, we introduce a *contract* abstraction which explicitly identifies who contributes data, what computation will take place on that data, who receives the results, and what conditions must be met. An example of conditions may be that other datasets are of sufficient size to ensure that the task's accuracy will be improved or to avoid a small sample-size bias. The contract provides this information to each participating agent before data sharing occurs, enabling agents to consider the risks and benefits of a potential sharing situation and propose changes to the contract to eventually reach a consensus with other participating agents. We believe the contract abstraction can enable more data sharing in practice by providing agents with a simple and clear way to understand the impact of a potential data sharing scenario.

5.4 Conclusion

These projects contribute concrete solutions that effectively improve the utility of data in both cloud lakehouse system design and data sharing scenarios. These projects enable us to understand the particular nuances of these two data management scenarios and enable us to identify that the utility of data is a useful and direct signal for data management problems broadly. This insight guided the approaches we presented in Chapters 3 and 4.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this dissertation, we identify that across a wide range of data management tasks, efficient solutions aim to balance costs with the value achieved from running tasks on data, thus improving utility. Currently, data managers rely on tools such as observability and monitoring to detect problems that require manual intervention or rely on signals to guide decisions. These can lead to inefficient decisions, presenting a significant opportunity for users to improve the benefits they gain from their data. We argue that data's utility, which is a function both of the value achieved from data and the costs of extracting that value, is a signal that better captures users' goals. This dissertation then explores solutions to data management problems that are designed to either directly improve data utility by reducing costs or by directly providing an indicator for the value of data to data managers as a signal that can guide better data management solutions.

Concretely, this dissertation proposes two new solutions:

- A set of algorithms and a prototype implementation which builds cheaper execution plans for SQL workloads that span across multiple cloud data warehouses, each employing a different pricing model. This approach improves data's utility by reducing cloud costs. [Srivastava and Fernandez, 2024]

- An indicator for the value of data which can be used to make key data management decisions, such as where to store data, how to prioritize limited resources on data, and what data setup modifications to make.

In two further projects [Xia et al., 2024, Srivastava et al., 2025] we explore concrete data management scenarios–building a scheduler for a cloud data lakehouse and facilitating data sharing agreements–which provide valuable insights into these scenarios and motivate the need for solutions that rely on data's utility as a signal.

Future Work in Reducing Costs and Improving Utility. Future work can aim to apply similar cost optimization techniques to other aspects of cloud costs beyond SQL analytical workloads and make the methods proposed more applicable to low-latency queries. Furthermore, work in supporting data stored across multiple clouds and propagating updates to multiple data copies, leveraging existing techniques in geo-distribution and distributed data storage, can improve the savings margins observed by ARACHNE.

Other work may aim to improve the quality of the value of data indicator by further refining the notion of relevance to leverage provenance techniques, such as provenance sketches [Niu et al., 2021], to more precisely determine what data is relevant to a task than relying solely on data access at progressively fine granularities. Similarly, developing methods to compute potentially uneven contribution of relevant data elements to tasks would further enable a more accurate value of data computation. Providing tools to enable data managers to more easily determine the values for tasks would similarly improve the quality of the indicator while reducing friction, enabling easier adoption. Additionally, the value of data indicator applies to tasks and data stored within the systems it is implemented in. Providing implementations across more data systems, and providing some orchestration across these systems, would enable the value of data to be tracked for more of an organization's data which may be spread across multiple cloud systems.

Finally, future work may identify more in-depth opportunities to apply the value of data indicator to data management problems beyond those identified here, such as automated view materialization, improving accuracy for key models, data sharing, or many others. Data's value and utility are incredibly useful concepts for data managers and organizations broadly, and building methods that leverage these concepts can vastly improve many of the roadblocks faced today. Indeed, rather than relying exclusively on bespoke solutions or signals, we illustrate in this dissertation that data management problems can be more effectively approached by relying on the utility of data, and we provide concrete solutions

which enable users to directly interact with and improve their data's utility.

BIBLIOGRAPHY

Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Sosic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic, Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, and Marko Zivanovic. Polaris: the distributed sql engine in azure synapse. *Proceedings of the VLDB Endowment*, 13(12):3204–3216, aug 2020. ISSN 2150-8097. doi:10.14778/3415478.3415545. URL https://doi.org/10.14778/3415478.3415545.

Niv Ahituv. A systematic approach toward assessing the value of an information system. *MIS Quarterly*, 4(4):61–75, 1980. ISSN 02767783, 21629730. URL http://www.jstor.org/stable/248961.

Amazon Athena. Amazon Athena - Serverless Interactive Query Service - Amazon Web Services. URL https://aws.amazon.com/athena/.

Amazon S3. Amazon s3. URL https://aws.amazon.com/s3/.

Amazon S3 Intelligent-Tiering. Amazon s3 intelligent-tiering. URL https://aws.amazon.com/s3/storage-classes/intelligent-tiering/.

Apache Arrow. Apache Arrow. URL https://arrow.apache.org.

Apache Parquet. Apache Parquet. URL https://parquet.apache.org/.

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi:10.1145/2723372.2742797. URL https://doi.org/10.1145/2723372.2742797.

Michael Armbrust, Ali Ghodsi, Reynold Xin, Matei Zaharia, et al. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, volume 8, page 28. sn, 2021.

Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis

Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. Amazon redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2205–2217, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi:10.1145/3514221.3526045. URL https://doi.org/10.1145/3514221.3526045.

Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas M. Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu, and Vasileios Zois. Flexible resource allocation for relational database-as-a-service. *Proc. VLDB Endow.*, 16 (13):4202–4215, September 2023. ISSN 2150-8097. doi:10.14778/3625054.3625058. URL https://doi.org/10.14778/3625054.3625058.

Auto-MV. Automated materialized views - Amazon Redshift. URL https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-auto-mv.html.

AWS Batch. What is Batch Processing? - Batch Processing Systems Explained - AWS. URL https://aws.amazon.com/what-is/batch-processing/.

AWS Batch Processing. Batch data processing - Data Analytics Lens. URL https://docs.aws.amazon.com/wellarchitected/latest/analytics-lens/batch-data-processing.html.

AWS DataSync. Aws datasync. https://aws.amazon.com/datasync/.

AWS S3 Pricing. S3 Pricing. URL https://aws.amazon.com/s3/pricing/.

Azure Synapse. Azure Synapse Analytics | Microsoft Azure. URL https://azure.microsoft.com/en-us/products/synapse-analytics/.

Tiemo Bang, Conor Power, Siavash Ameli, Natacha Crooks, and Joseph M. Hellerstein. Optimizing the cloud? don't train models. build oracles! In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. URL https://www.cidrdb.org/cidr2024/papers/p47-bang.pdf.

BatchExecuteStatement. Batchexecutestatement. URL https://docs.aws.amazon.com/redshift-data/latest/APIReference/API_BatchExecuteStatement.html.

Bauplan Labs. Bauplan: Where agents build safely on production data. https://www.bauplanlabs.com.

Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 221–230, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi:10.1145/3183713.3190662. URL https://doi.org/10.1145/3183713.3190662.

Johan Bergman. Ensure business value by adopting a data product process. https://www.dataedge.se/post/ensure-business-value-by-adopting-a-data-product-process, 2022. Accessed 2026-01-28.

Leopoldo Bertossi, Benny Kimelfeld, Ester Livshits, and Mikaël Monet. The shapley value in database management. *SIGMOD Rec.*, 52(2):6–17, August 2023. ISSN 0163-5808. doi:10.1145/3615952.3615954. URL https://doi.org/10.1145/3615952.3615954.

Bigeye. Bigeye. URL https://www.bigeye.com/.

BigQuery. What is BigQuery? URL https://cloud.google.com/bigquery/docs/introduction.

BigQuery analytics. Overview of bigquery analytics. URL https://cloud.google.com/bigquery/docs/query-overview.

BigQuery Information Schema. Introduction to information_schema. https://cloud.google.com/bigquery/docs/information-schema-intro.

BigQuery Optimize Query Computation. Optimize query computation. https://docs.cloud.google.com/bigquery/docs/best-practices-performance-compute.

BigQuery Optimizing. Introduction to optimizing query performance | BigQuery. URL https://cloud.google.com/bigquery/docs/best-practices-performance-overview.

BigQuery Partitioned Tables. Introduction to partitioned tables. https://docs.cloud.google.com/bigquery/docs/partitioned-tables.

BigQuery Pricing Change. Introducing new BigQuery pricing editions. URL https://cloud.google.com/blog/products/data-analytics/introducing-new-bigquery-pricing-editions.

BigQuery Reliability. Understand reliability | bigquery. URL https://cloud.google.com/bigquery/docs/reliability-intro#real-time_analytics.

Bloomberg B-PIPE. Real-time market data feed. URL https://professional.bloomberg.com/products/data/enterprise-catalog/real-time-data-feed/#global-connectivity.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30:107–117, 1998. URL https://snap.stanford.edu/class/cs224w-readings/Brin98Anatomy.pdf.

Ranjan Burman, Amit Nayak, Bosco Albuquerque, and Nita Shah. Best practices to optimize your Amazon Redshift and MicroStrategy deployment | AWS Big Data Blog, February 2022. URL https://aws.amazon.com/blogs/big-data/best-practices-to-optimize-your-amazon-redshift-and-microstrategy-deployment/.

Raul Castro Fernandez. What is the value of data? a theory and systematization. *ACM / IMS J. Data Sci.*, 2(1), June 2025. doi:10.1145/3728476. URL https://doi.org/10.1145/3728476.

Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. Semagrow: optimizing federated sparql queries. In *Proceedings of the 11th International Conference on Semantic Systems*, SEMANTICS '15, page 121–128, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334624. doi:10.1145/2814864.2814886. URL https://doi.org/10.1145/2814864.2814886.

Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E. Gonzalez, Joseph M. Hellerstein, Michael I. Jordan, Anthony D. Joseph, Michael W. Mahoney, Aditya Parameswaran, David Patterson, Raluca Ada Popa, Koushik Sen, Scott Shenker, Dawn Song, and Ion Stoica. The sky above the clouds, 2022. URL https://arxiv.org/abs/2205.07147.

Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if" index analysis utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, page 367–378, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919955. doi:10.1145/276304.276337. URL https://doi.org/10.1145/276304.276337.

Surajit Chaudhuri and Vivek Narasayya. Index merging. In *15th International Conference on Data Engineering*. IEEE Computer Society, April 1999. URL https://www.microsoft.com/en-us/research/publication/index-merging/.

Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 3–14. VLDB Endowment, 2007. ISBN 9781595936493.

Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 263–274, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130848. doi:10.1145/304182.304206. URL https://doi.org/10.1145/304182.304206.

Surajit Chaudhuri, Arnd Christian König, Arnd Christian König, Vivek Narasayya, Ravi Ramamurthy, Manoj Syamala, and Nico Bruno. Autoadmin project at microsoft research:

Lessons learned. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, December 2011. URL https://www.microsoft.com/en-us/research/publication/autoadmin-project-at-microsoft-research-lessons-learned/.

Michael Chen. What is data deduplication? methods and benefits. https://www.oracle.com/data-deduplication, February 14, 2024.

Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. *The VLDB Journal The International Journal on Very Large Data Bases*, 11(3): 216–237, November 2002. ISSN 10668888. doi:10.1007/s00778-002-0070-0. URL http://link.springer.com/10.1007/s00778-002-0070-0.

Google Cloud. Introduction to clustered tables, 2026a. URL https://docs.cloud.google.com/bigquery/docs/clustered-tables. Accessed: 2026-01-29.

Google Cloud. Bigquery pricing, 2026b. URL https://cloud.google.com/bigquery/pricing. Accessed: 2026-01-29.

CloudZero. Meet Our Customers | CloudZero. URL https://www.cloudzero.com/customers.

Alison Cuffari, Riddhi Acharya, Michael Le, Elizabeth Stein, Abed El-Husseini, and Zack Grossenbacher. Valuing data assets. https://www.deloitte.com/us/en/insights/topics/digital-transformation/valuing-data-assets.html, 2023. Accessed 2026-01-28.

Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi:10.1145/2882903.2903741. URL https://doi.org/10.1145/2882903.2903741.

Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, page 330–341, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1558603824.

Ben Darnell. 3 basic rules for choosing indexes. https://www.cockroachlabs.com/blog/how-to-choose-db-index-keys/.

Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of*

the 2019 International Conference on Management of Data, SIGMOD '19, page 666–679, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi:10.1145/3299869.3314035. URL https://doi.org/10.1145/3299869.3314035.

Dashboard. Real-World Examples of Business Intelligence (BI) Dashboards. URL https://www.tableau.com/learn/articles/business-intelligence-dashboards-examples.

Databricks. Databricks. URL https://www.databricks.com.

Databricks Data Lakes. Introduction to Data Lakes. URL https://www.databricks.com/discover/data-lakes.

Bappaditya Datta. Data warehouse and business intelligence technology consolidation using aws, July 2022. URL https://aws.amazon.com/blogs/architecture/data-warehouse-and-business-intelligence-technology-consolidation-using-aws/.

Zhamak Dehghani. Data mesh principles and logical architecture. https://martinfowler.com/articles/data-mesh-principles.html, December 03, 2020.

Veeral Desai, Tim Fountaine, and Kayvaun Rowshankish. How to unlock the full value of data: Manage it like a product. https://www.mckinsey.com/capabilities/quantumblack/our-insights/how-to-unlock-the-full-value-of-data-manage-it-like-a-product, 2021. Accessed 2026-01-28.

Yefim Dinitz. *Dinitz' Algorithm: The Original Version and Even's Version*, pages 218–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-32881-0. doi:10.1007/11685654_10. URL https://doi.org/10.1007/11685654_10.

Dominik Durner, Badrish Chandramouli, and Yinan Li. Crystal: a unified cache storage system for analytical databases. *Proceedings of the VLDB Endowment*, 14(11):2432–2444, July 2021. ISSN 2150-8097. doi:10.14778/3476249.3476292. URL https://dl.acm.org/doi/10.14778/3476249.3476292.

Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

Stephan Ewen, Holger Kache, Volker Markl, and Vijayshankar Raman. Progressive query optimization for federated queries. In Yannis Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, pages 847–864, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32961-9.

Gerald A. Feltham. The value of information. *The Accounting Review*, 43(4):684–696, 1968. ISSN 00014826. URL http://www.jstor.org/stable/243630.

Cibele Freire, Wolfgang Gatterbauer, Neil Immerman, and Alexandra Meliou. The complexity of resilience and responsibility for self-join-free conjunctive queries. *Proc. VLDB Endow.*, 9(3):180–191, November 2015. ISSN 2150-8097. doi:10.14778/2850583.2850592. URL https://doi.org/10.14778/2850583.2850592.

Charles A. Gallagher. Perceptions of the value of a management information system. *The Academy of Management Journal*, 17(1):46–55, 1974. ISSN 00014273. URL http://www.jstor.org/stable/254770.

Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603, 2009. doi:10.1109/ICDE.2009.130.

Ali Ghodsi, Stas Kelvich, Heikki Linnakangas, Nikita Shamgunov, Arsalan Tavakoli-Shiraji, Patrick Wendell, Reynold Xin, and Matei Zaharia. A new era of databases: Lakebase, February 24, 2026. URL https://www.databricks.com/blog/what-is-a-lakebase.

Amirata Ghorbani and James Zou. Data shapley: Equitable valuation of data for machine learning. In *International conference on machine learning*, pages 2242–2251. PMLR, 2019.

Soumya Ghorpode. Data lifecycle management policy. https://www.itgov-docs.com/blogs/data-governance/6-data-lifecycle-management-policy-creation-usage-archival-deletion, August 27, 2025. Accessed 2026-01-28.

Gitnux. Data management statistics, February 13, 2026. URL https://gitnux.org/data-management-statistics/.

Boris Glavic, Alexandra Meliou, and Sudeepa Roy. Trends in explanations: Understanding and debugging data-driven systems. *Found. Trends Databases*, 11(3):226–318, August 2021. ISSN 1931-7883. doi:10.1561/1900000074. URL https://doi.org/10.1561/1900000074.

Boris Glavic, Pengyuan Li, Ziyu Liu, Dieter Gawlick, Vasudha Krishnaswamy, Danica Porobic, and Zhen Hua Liu. Towards an objective metric for data value through relevance. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, page 9, New York, NY, USA, 2024. Association for Computing Machinery. URL https://www.cidrdb.org/cidr2024/.

Kevin Goff. The Baker's Dozen: 13 Tips for Better Extract/Transform/Load (ETL) Practices in Data Warehousing (Part 1 of 2). URL https://www.codemag.com/article/1709051/The-Baker%E2%80%99s-Dozen-13-Tips-for-Better-Extract-Transform-Load-ETL-Practices-in-Data-Warehousing-Part-1-of-2.

Google BigQuery Storage Pricing. Pricing | BigQuery: Cloud Data Warehouse. URL https://cloud.google.com/bigquery/pricing.

Google Cloud Network Pricing. Google cloud all networking pricing. URL https://cloud.google.com/vpc/network-pricing?hl=en.

Google ETL. What is ETL? URL https://cloud.google.com/learn/what-is-etl.

Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 395–398, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912365. doi:10.1145/38713.38755. URL https://doi.org/10.1145/38713.38755.

Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1917–1923, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi:10.1145/2723372.2742795. URL https://doi.org/10.1145/2723372.2742795.

T. Heller, S. O. Krumke, and K. H. Küfer. The reward-penalty-selection problem, 2021. URL https://arxiv.org/abs/2106.14601.

Bal Heroor. Zero etl: Benefits and limitations. https://mactores.com/blog/zero-etl-benefits-and-limitations, June 6, 2024.

Randall Hunt. S3 Select and Glacier Select – Retrieving Subsets of Objects | AWS News Blog, November 2017. URL https://aws.amazon.com/blogs/aws/s3-glacier-select/.

IBM. What is data lifecycle management (dlm)? https://www.ibm.com/think/topics/data-lifecycle-management, 2026. Accessed 2026-01-28.

IBM Serial Batch. What is a workload? URL https://www.ibm.com/topics/workload.

Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. Garlic: a new flavor of federated query processing for db2. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 524–532, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134975. doi:10.1145/564691.564751. URL https://doi.org/10.1145/564691.564751.

Akif Quddus Khan, Nikolay Nikolov, Mihhail Matskin, Radu Prodan, Christoph Bussler, Dumitru Roman, and Ahmet Soylu. Towards cloud storage tier optimization with rule-based classification. In George A. Papadopoulos, Florian Rademacher, and Jacopo Soldani, editors, *Service-Oriented and Cloud Computing*, pages 205–216, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-46235-1.

Kyle Kirwan. Complete guide to understanding data observability in 2024. https://www.bigeye.com/blog/data-observability, January 3, 2024. Accessed 2026-03-25.

Arnd Christian König, Yi Shan, Karan Newatia, Luke Marshall, and Vivek Narasayya. Solver-in-the-loop cluster resource management for database-as-a-service. *Proc. VLDB Endow.*, 16(13):4254–4267, September 2023. ISSN 2150-8097. doi:10.14778/3625054.3625062. URL https://doi.org/10.14778/3625054.3625062.

Donald Kossmann, Michael J. Franklin, Gerhard Drasch, and Wig Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Transactions on Database Systems*, 25(4):517–558, December 2000. ISSN 0362-5915, 1557-4644. doi:10.1145/377674.377677. URL https://dl.acm.org/doi/10.1145/377674.377677.

Yannis Kotidis and Nick Roussopoulos. DynaMat: a dynamic view management system for data warehouses. *ACM SIGMOD Record*, 28(2):371–382, June 1999. ISSN 0163-5808. doi:10.1145/304181.304215. URL https://dl.acm.org/doi/10.1145/304181.304215.

Tom Krantz and Alexandra Jonker. The cost of poor data quality. https://www.ibm.com/think/insights/cost-of-poor-data-quality, 2023. Accessed 2026-01-28.

Anoop Kumar K M, Shaheer Mansoor, and Sreenivas Nettem. Modernize your legacy databases with aws data lakes, part 3: Build a data lake processing layer, October 30, 2024. URL https://aws.amazon.com/blogs/big-data/modernize-your-legacy-databases-with-aws-data-lakes-part-3-build-a-data-lake-processing-layer. Accessed: 2025-04-16.

Viktor Leis and Maximilian Kuschewski. Towards cost-optimal query processing in the cloud. *Proceedings of the VLDB Endowment*, 14(9):1606–1612, May 2021. ISSN 2150-8097. doi:10.14778/3461535.3461549. URL https://dl.acm.org/doi/10.14778/3461535.3461549.

Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, November 2015. ISSN 2150-8097. doi:10.14778/2850583.2850594. URL https://dl.acm.org/doi/10.14778/2850583.2850594.

Pengyuan Li, Boris Glavic, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, Danica Porobic, and Xing Niu. In-memory incremental maintenance of provenance sketches. In Wolfgang Lehner, Vanessa Braganholo, Kostas Stefanidis, Zheying Zhang, Alexander Krause, and João Felipe Nicolaci Pimentel, editors, *Proceedings 29th International Conference on Extending Database Technology, EDBT 2026, Tampere, Finland, March 24-27, 2026*, pages 56–68. OpenProceedings.org, 2026. doi:10.48786/EDBT.2026.05. URL https://doi.org/10.48786/edbt.2026.05.

Rundong Li, Ningfang Mi, Mirek Riedewald, Yizhou Sun, and Yi Yao. Abstract cost models for distributed data-intensive computations. *Distributed and Parallel Databases*, 37(3): 411–439, September 2019. ISSN 1573-7578. doi:10.1007/s10619-018-7244-2. URL https://doi.org/10.1007/s10619-018-7244-2.

Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. Combining aggregation and sampling (nearly) optimally for approximate query processing. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1129–1141, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi:10.1145/3448016.3457277. URL https://doi.org/10.1145/3448016.3457277.

Ari Libarikian, Kayvaun Rowshankish, Markus Berger-de Leon, and Vishnu Kamalnath. From raw data to real profits: A primer for building a thriving data business. https://www.mckinsey.com/capabilities/tech-and-ai/our-insights/from-raw-data-to-real-profits-a-primer-for-building-a-thriving-data-business, 2022. Accessed 2026-01-28.

Guy Lohman. Is Query Optimization a "Solved" Problem? – ACM SIGMOD Blog, April 2014. URL http://wp.sigmod.org/?p=1075.

John Martinez. 50 Years Of ETL: Can SQL For ETL Be Replaced?, May 2021. URL https://www.datanami.com/2021/05/06/50-years-of-etl-can-sql-for-etl-be-replaced/.

Steve McDowell. The impact of storage on the ai lifecycle. https://www.weka.io/resources/analyst-report/the-impact-of-storage-on-the-ai-lifecycle/, October 21, 2025.

McKinsey. Cloud cost-optimization simulator | McKinsey. URL https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/cloud-cost-optimization-simulator.

McKinsey & Company. The missing data link: Five practical lessons to scale your data products. https://www.mckinsey.com/capabilities/tech-and-ai/our-insights/the-missing-data-link-five-practical-lessons-to-scale-your-data-products, 2023. Accessed 2026-01-28.

Monte Carlo. Monte carlo: Trust your agents in production. URL https://www.montecarlodata.com/.

Barr Moses. What is data observability? 5 key pillars to know. https://www.montecarlodata.com/blog-what-is-data-observability/, July 31, 2025. Accessed 2026-03-25.

Aniket Murhekar, Jiaxin Song, Parnian Shahkar, Bhaskar Ray Chaudhury, and Ruta Mehta. You get what you give: Reciprocally fair federated learning. In Aarti Singh, Maryam Fazel,

Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 45289–45310. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/murhekar25a.html.

Thomas P. Nadeau and Toby J. Teorey. Achieving scalability in olap materialized view selection. In *Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '02, page 28–34, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135904. doi:10.1145/583890.583895. URL https://doi.org/10.1145/583890.583895.

Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *VLDB*, volume 6 of *VLDB '06*, pages 1049–1058, 2006.

Box News. Understanding data lifecycle management. https://blog.box.com/three-main-goals-of-data-lifecycle-management, December 5, 2023. Accessed 2026-01-28.

Xing Niu, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. Provenance-based data skipping. *Proc. VLDB Endow.*, 15(3):451–464, November 2021. ISSN 2150-8097. doi:10.14778/3494124.3494130. URL https://doi.org/10.14778/3494124.3494130.

Jake Noble. A practical guide to building an online recommendation system. https://www.databricks.com/blog/guide-to-building-online-recommendation-system, October 27, 2025.

Octopai. Saving data costs with data lineage, 2024. URL https://www.octopai.com/saving-data-costs-with-data-lineage/. Accessed: 2025-04-16.

Luis L. Perez and Christopher M. Jermaine. History-aware query optimization with materialized intermediate views. In *2014 IEEE 30th International Conference on Data Engineering*, pages 520–531, Chicago, IL, USA, March 2014. IEEE. ISBN 978-1-4799-2555-1. doi:10.1109/ICDE.2014.6816678. URL http://ieeexplore.ieee.org/document/6816678/.

Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. How i learned to stop worrying and love re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1758–1761, 2019. doi:10.1109/ICDE.2019.00191.

Orestis Polychroniou, Wangda Zhang, and Kenneth A. Ross. Distributed Joins and Data Placement for Minimal Network Traffic. *ACM Transactions on Database Systems*, 43(3):1–45, November 2018. ISSN 0362-5915, 1557-4644. doi:10.1145/3241039. URL https://dl.acm.org/doi/10.1145/3241039.

Ethan Post. Data observability architecture and optimizing your coverage. https://www.montecarlodata.com/blog-data-observability-architecture-and-optimizing-your-coverage/, December 20, 2023. Accessed 2026-03-25.

Avijit Prasad. Data lakes - Azure Architecture Center, December 2022. URL https://learn.microsoft.com/en-us/azure/architecture/data-guide/scenarios/data-lake.

Presto. Presto: Free, Open-Source SQL Query Engine for any Data. URL http://prestodb.github.io/.

Redshift Spectrum. Spectrum performance caching and performance | AWS re:Post. URL https://repost.aws/questions/QUaVNX2NJ0REm95-dhZY4O5A/spectrum-performance-caching-and-performance.

Matt Scaer, Manish Vazirani, and Tarun Chaudhary. Top 10 performance tuning techniques for Amazon Redshift | AWS Big Data Blog, August 2020. URL https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-techniques-for-amazon-redshift/.

Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813, Macao, Macao, April 2019. IEEE. ISBN 978-1-5386-7474-1. doi:10.1109/ICDE.2019.00196. URL https://ieeexplore.ieee.org/document/8731547/.

Pathik Sharma and James Fu. Cost optimization best practices for BigQuery. URL https://cloud.google.com/blog/products/data-analytics/cost-optimization-best-practices-for-bigquery.

Tarique Siddiqui, Vivek Narasayya, Marius Dumitru, and Surajit Chaudhuri. Cache-efficient top-k aggregation over high cardinality large datasets. *Proc. VLDB Endow.*, 17(4):644–656, December 2023. ISSN 2150-8097. doi:10.14778/3636218.3636222. URL https://doi.org/10.14778/3636218.3636222.

Cody Slingerland. Aws redshift guide: Use cases, pros and cons, and pricing. https://www.cloudzero.com/blog/aws-redshift/.

Snappy. snappy. URL http://google.github.io/snappy/.

Snowflake. Micro-partitions & data clustering, 2026. URL https://docs.snowflake.com/en/user-guide/tables-clustering-micropartitions. Accessed: 2026-01-29.

Snowflake Account Usage. Snowflake account usage. https://docs.snowflake.com/en/sql-reference/account-usage.

Snowflake ETL. The Pitfalls of ETL Processing. URL https://www.snowflake.com/guides/pitfalls-etl-processing.

Apache Spark. Interface queryexecutionlistener. URL https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/util/QueryExecutionListener.html.

Tapan Srivastava and Raul Castro Fernandez. Saving money for analytical workloads in the cloud. *Proc. VLDB Endow.*, 17(11):3524–3537, July 2024. ISSN 2150-8097. doi:10.14778/3681954.3682018. URL https://doi.org/10.14778/3681954.3682018.

Tapan Srivastava, Jacopo Tagliabue, and Ciro Greco. Eudoxia: a faas scheduling simulator for the composable lakehouse. *Proceedings of Workshops at the 51th International Conference on Very Large Data Bases*, 2150:8097, 2025.

Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. The ldbc social network benchmark: Business intelligence workload. *Proc. VLDB Endow.*, 16(4):877–890, dec 2022. ISSN 2150-8097. doi:10.14778/3574245.3574270. URL https://doi.org/10.14778/3574245.3574270.

Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. Choosing a cloud dbms: architectures and tradeoffs. *Proc. VLDB Endow.*, 12(12):2170–2182, aug 2019. ISSN 2150-8097. doi:10.14778/3352063.3352133. URL https://doi.org/10.14778/3352063.3352133.

Zoiner Tejada. Online analytical processing (OLAP) - Azure Architecture Center. URL https://learn.microsoft.com/en-us/azure/architecture/data-guide/relational-data/online-analytical-processing.

The Duckbill Group. Duckbill. URL https://www.duckbillgroup.com/.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005, Long Beach, CA, USA, 2010a. IEEE. ISBN 978-1-4244-5445-7. doi:10.1109/ICDE.2010.5447738. URL http://ieeexplore.ieee.org/document/5447738/.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005, Long Beach, CA, USA, 2010b. IEEE. ISBN 978-1-4244-5445-7. doi:10.1109/ICDE.2010.5447738. URL http://ieeexplore.ieee.org/document/5447738/.

Tietoevry. Measuring the value of your data, 2024. URL https://www.tietoevry.com/en/blog/2024/07/measuring-the-value-of-your-data/. Accessed: 2025-04-16.

TPC. Tpc-h version 2 and version 3, 2026. URL https://www.tpc.org/tpch/. Accessed: 2026-02-08.

TPC-DS. Tpc benchmark ds. URL https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf.

Uber Monitoring Data Quality. Monitoring data quality at scale with statistical modeling. https://www.uber.com/blog/monitoring-data-quality-at-scale/, May 7, 2020. Accessed 2026-03-25.

Unity Catalog. Unity catalog: Unified and open governance for data and ai. URL https://www.databricks.com/product/unity-catalog.

Yair Wand and Richard Y. Wang. Anchoring data quality dimensions in ontological foundations. *Commun. ACM*, 39(11):86–95, November 1996. ISSN 0001-0782. doi:10.1145/240455.240479. URL https://doi.org/10.1145/240455.240479.

Xiaoying Wang, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. Wii: Dynamic budget reallocation in index tuning. *Proc. ACM Manag. Data*, 2(3), May 2024. doi:10.1145/3654985. URL https://doi.org/10.1145/3654985.

Xiaoying Wang, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. Esc: An early-stopping checker for budget-aware index tuning. *Proc. VLDB Endow.*, 18(5):1278–1290, January 2025. ISSN 2150-8097. doi:10.14778/3718057.3718059. URL https://doi.org/10.14778/3718057.3718059.

What is a Data Lake. What is a data lake. URL https://aws.amazon.com/what-is/data-lake/.

What is OLAP? What is Online Analytical Processing? - Online Analytical Processing Explained - AWS. URL https://aws.amazon.com/what-is/olap/.

Alicia Williams and Yogendra Joshi. Cost management in bigquery: how to control spending with budgets and custom quotas, November 28, 2023. URL https://cloud.google.com/blog/products/data-analytics/manage-bigquery-costs-with-custom-quotas.

Sarah Wooders, Shu Liu, Paras Jain, Xiangxi Mo, Joseph E. Gonzalez, Vincent Liu, and Ion Stoica. Cloudcast: High-Throughput, Cost-Aware overlay multicast in the cloud. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 281–296, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/wooders.

Fuhui Wu, Qingbo Wu, and Yusong Tan. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, 71(9):3373–3418, September 2015. ISSN 0920-8542, 1573-0484. doi:10.1007/s11227-015-1438-4. URL http://link.springer.com/10.1007/s11227-015-1438-4.

Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1081–1092, 2013. doi:10.1109/ICDE.2013.6544899.

Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be late: Optimizing spot instance savings under deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 185–203, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/wu-zhanghao.

Siyuan Xia, Chris Zhu, Tapan Srivastava, Bridget Fahey, and Raul Castro Fernandez. Programmable dataflows: Abstraction and programming model for data sharing. *arXiv preprint arXiv:2408.04092*, 2024.

Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, nov 2019. ISSN 2150-8097. doi:10.14778/3368289.3368294. URL https://doi.org/10.14778/3368289.3368294.

Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1802–1805, April 2020. doi:10.1109/ICDE48307.2020.00174.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.