THE UNIVERSITY OF CHICAGO

SAVING MONEY FOR ANALYTICAL WORKLOADS IN THE CLOUD

AN M.S. PAPER SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
TAPAN SRIVASTAVA

CHICAGO, ILLINOIS
MAY 24, 2023

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF TABLES vi

ACKNOWLEDGMENTS vii

ABSTRACT viii

1 INTRODUCTION 1

2 BACKGROUND AND OPPORTUNITIES 5
2.1 Characterization of Analytics Workload 5
2.2 Cloud Data Warehouses and Pricing Models 5
2.3 Breakdown of Cloud Costs 6
2.4 Cost Saving Opportunity and Challenges 7
2.5 Problem Statement and Goals 8

3 INTER QUERY EXECUTION PLAN 10
3.1 Algorithmic Setup and Goal 10
3.2 Designing the Algorithm 12
3.2.1 Optimal Solution 12
3.2.2 Intuition for the Greedy Strategy 14
3.3 Inter-Query Algorithm 16

4 INTRA QUERY EXECUTION PLAN 17
4.1 Algorithmic Setup and Goal 17
4.2 Identifying Profitable Cuts 17
4.3 Intra-Query Algorithm 20

5 ARACHNE OVERVIEW 21
5.1 Overview 21
5.2 Profiler Module 22
5.3 O3: The Impact of SQL Syntax 24
5.4 O4: Cost Savings with IaaS 24
5.5 Cost-Relevant Implementation Details 25

6 EVALUATION 26
6.1 Experimental Setup 26
6.2 RQ1. Inter-Query Processing 28
6.2.1 Cost-Saving Inter-Query plans 28
6.2.2 Studying the Impact of IaaS in Cost Saving 32
6.2.3 Profiling Cost 32

6.3 RQ2. Hybrid Query Processing 34
6.4 RQ3. SQL Pre-Optimization 35
6.5 RQ4. What-If Analysis on Cloud Costs 37

7 RELATED WORK 39

8 CONCLUSION 41

REFERENCES 42

LIST OF FIGURES

1.1 Size scanned (TB) vs. runtime (hours) for 2 queries. Decision boundary for size vs. runtime using $5/TB (pay-per-byte) and $1.086/hour (pay-per-compute). . . 2

3.1 Bipartite model for the inter-query algorithm. The top number on a node shows its query savings σ_q or migration cost μ_t. The bottom value shows v_t or v_q for that table or query. We show how this value is calculated for t_3 and q_2. 12

5.1 Arachne overview, with system components and execution backends 21

6.1 Breakdown of Arachne costs into migration costs, cost of queries in Redshift, and cost of queries in BigQuery, compared to the BigQuery baseline. Three Multi datasets (1TB GA1)–points labelled in Figure 6.2a. 29
6.2 Cost (USD) vs runtime (hours) for datasets with Multi plans over Amazon Redshift and Google BigQuery. 29
6.3 Query costs normalized to most expensive plan for Arachne's hybrid plans vs plans only in BigQuery, DuckDB, and DuckDB with Arachne-produced syntax. . 34
6.4 DuckDB and Redshift Percent Speedup of Machine-Generated SQL over Standard Syntax 36
6.5 (GA1 1–2TB) Simulated inter-query results varying either BigQuery (pay-per-byte) cost–labelled as BQ–or egress cost from source cloud–labelled as Egress. . 37

LIST OF TABLES

2.1 Prices for databases in pay-per-compute (PC) and pay-per-byte (PB), blob storage, read/write from blob storage, and data egress (GCP us-east1, S3 us-east, Azure hot storage). Systems used in evaluation are bolded. 7

6.1 Inter-query plan-type by setup at 1TB and 2TB. 29
6.2 Profiling costs, iterations needed to compensate for profiling costs, and estimation error for 15, 25, 50, and 100% samples (GA1 1TB). 31
6.3 Absolute baseline, Arachne hybrid plan costs. Alt Syntax is DuckDB with Arachne-produced SQL 34

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Raul Castro Fernandez, for his guidance throughout this project. I continue to learn a tremendous amount from the way he asks questions and analyzes problems, and I look forward to continuing to learn from him. I am also grateful to Prof. Michael Franklin and Prof. Aaron Elmore for devoting their time and energy to be on my committee. I would like to thank Hartrich Zack for all their support and assistance making the color schemes and figures present in this paper as well as all the fellow Ph.D. students who helped me work through the various phases of this project. Finally, I would like to thank my parents, Amitabh and Richa Srivastava, for teaching me the value of hard work and for all their love, support, and advice.

ABSTRACT

As users migrate their analytical workloads to cloud databases, it is becoming just as important to reduce monetary cost as it is to improve query performance. In the cloud, users choose to pay based on either the compute time or the amount of data a query reads. We observe that analytical queries are compute- or IO-intensive and each query type executes cheaper in a different pricing model. We exploit this opportunity and propose methods to build cheaper execution plans across pricing models. We implement these methods, observe how SQL syntax and query optimizers impact cost, and consider a transparent deployment of DuckDB on infrastructure-as-a-service to avoid the cost premiums of platform-as-a-service without placing the burden of deployment or maintenance of software on users. We reduce workload costs by as much as 80% and produces execution plans spanning multiple pricing models that save as much as 47%. We also reduce individual query costs by as much as 90%. We simulate the effect of different cloud prices on real query performance and observe that there are still significant multi-cloud opportunities if cloud vendors change their prices. These results indicate the massive opportunity to save money by building execution plans across multiple pricing models.

CHAPTER 1

INTRODUCTION

Saving money is as important as improving query performance for cloud analytics. This is particularly true for *periodic* workloads because any achieved savings on a query accumulate over time. For example, saving $5 on a single ETL query that runs three times a day will save $5400 a year, and organizations may have hundreds of such periodic queries to power different applications such as filling dashboards to track store information or managing ETL pipelines [24, 85, 37, 31, 61, 88, 19, 15]. As the Duckbill group describes [67], these periodic workloads take up a significant portion of data analytics budgets [48, 54, 17], so reducing costs for periodic workloads is a high priority [33, 75, 14, 3, 20, 76, 5]. Unfortunately, despite increasing costs, cloud databases are designed to maximize query performance and do not offer many mechanisms to directly save money. To save money, users are increasingly working with consulting firms such as McKinsey, the DuckBill group, or CloudZero [16, 53, 23] that configure and optimize deployments for individual users. We also configure cloud databases per documentation and best practices [38, 41, 9, 8], but our aim is to explore more general cost saving opportunities that go beyond tuning databases.

The simple, key observation we make is that there is a variety of pricing models available and each prices IO and CPU differently. This creates an opportunity to save money by scheduling queries across clouds and placing CPU- and IO-bound queries in the cloud service with the lower cost. The two most prominent pricing models 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 [45] while in *pay-per-byte* the user pays for the amount of data scanned irrespective of compute time, e.g., in Google BigQuery [7]. A CPU-bound query will execute cheapest in a pay-per-byte system while an IO-bound query runs cheapest in pay-per-compute. This is illustrated in Figure 1.1, which shows scanned bytes versus runtime for queries considering a database that charges $5/TB (pay-per-byte) and one that charges $1.086/hour (pay-per-

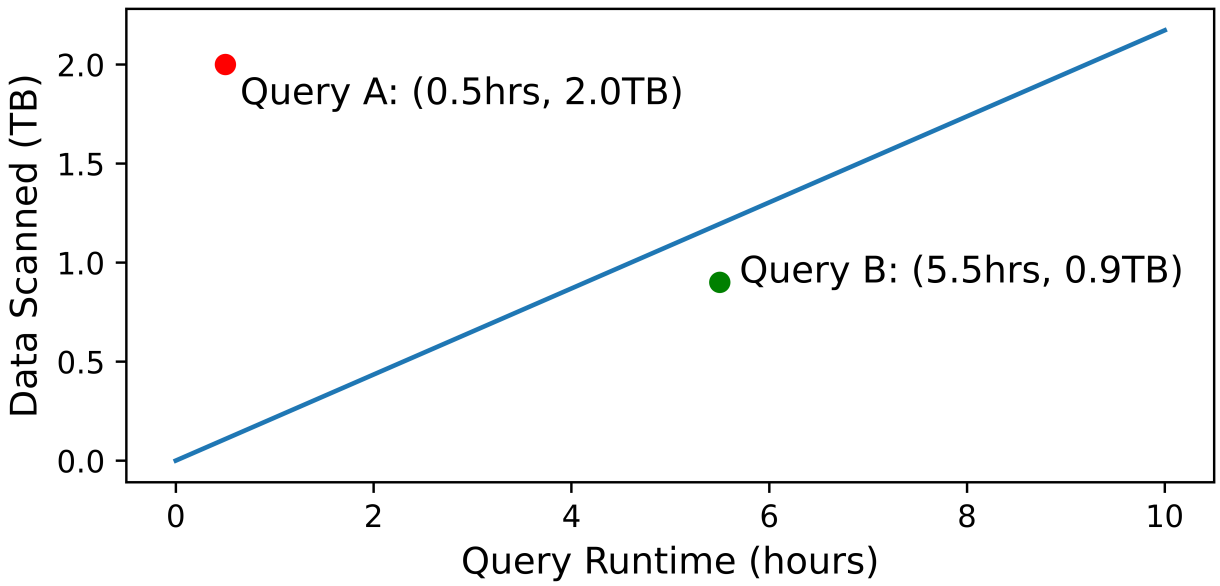


Figure 1.1: Size scanned (TB) vs. runtime (hours) for 2 queries. Decision boundary for size vs. runtime using \$5/TB (pay-per-byte) and \$1.086/hour (pay-per-compute).

compute). The figure includes a decision boundary: queries above the line are cheaper in pay-per-compute while queries below the line are cheaper in pay-per-byte. IO-intensive query A reads 2TB and runs for 30 minutes; compute-intensive query B reads 0.9TB and runs for 5.5 hours. Running query A in *pay-per-compute* and query B in *pay-per-byte* would be cheaper than running both queries in a single cloud.

We study and characterize opportunities to save money when scheduling queries across clouds with different pricing models. Specifically, we exploit the following opportunities:

- **O1: Inter-Query Algorithm.** Identify opportunities to save costs by scheduling queries on cloud databases with different pricing models.
- **O2: Intra-Query Algorithm.** Identify *hybrid query plans* that schedule query subplans across cloud databases with different pricing models.

We propose two algorithms to exploit opportunities **O1** and **O2**. While algorithmic contributions are important, we wanted to ensure our algorithms incorporate all relevant details of a deployment, so we built a prototype called Arachne to pre-process and schedule query workloads across cloud databases according to the inter- and intra-query algorithms,

account for all relevant costs—including data migration, loading, storage, and more—and ensure that our algorithms did not make simplifying assumptions that would diminish their impact. In the process of implementing Arachne we identified two additional unexpected saving opportunities:

- **O3: SQL Syntax.** Subtle changes to SQL syntax cause query optimizers to choose different physical plans for the same logical query. These different physical plans may execute significantly faster (or slower), impacting cost in *pay-per-compute* models. We measure the impact of varying SQL syntax on query cost and use the SQL syntax which yields the lowest cost.
- **O4: IaaS.** To avoid deploying and maintaining software, users pay a premium to use platform-as-a-service (PaaS) cloud databases. Since Arachne hides the underlying backend behind a SQL interface, it allows us to swap PaaS for the cheaper infrastructure-as-a-service (IaaS) and study the effect on costs. We deploy DuckDB [82] on top of IaaS to avoid the PaaS premium while handling deployment and maintenance automatically for users.

We incorporate **O1-O4** into Arachne and study the impact of these opportunities on workload costs. We choose Amazon Redshift [45] and Google BigQuery [7] which employ pay-per-compute and pay-per-byte pricing models to conduct our study, and we carefully configure these systems to the best of our ability to minimize query cost. We leave an exhaustive comparison of the impact of these opportunities on specific systems for future work. Overall, our results show that there are massive opportunities for saving money. We achieved up to 47% savings (we run workloads for $99 while the original cost $193) with an inter-query plan across pricing models and achieved up to 90% cost savings on a query via an intra-query plan. We also provide an in-depth study of the impact of SQL syntax on query cost, showing how small syntax changes can reduce cost up to 97% or increase it up to 4×.

Our results depend, to certain extent, on the prices that cloud vendors set; however, these prices are subject to change. We investigate if multi-cloud savings achieved by exploiting **O1**–**O4** still exist as cloud prices change. We conduct a *what-if* analysis where we plug in the real runtime and size scanned by queries into an analytical model, vary cloud prices, and evaluate if multi-cloud plans still yield savings. This analysis spotlights that these results are robust to changing cloud prices, how cloud egress costs–the cost cloud vendors charge for moving data out of their cloud–are a financial barrier to data migration, and how changes to these encourage data movement or lock-in users to a single cloud.

The rest of the paper is organized as follows. Section 2 presents background and the cost saving opportunity. Sections 3 and 4 present the inter- and intra-query algorithms to exploit **O1** and **O2**. Section 5 presents Arachne and the strategies used to exploit **O3**–**O4**. Section 6 presents the evaluation, followed by related work (Section 7) and conclusions in Section 8.

CHAPTER 2

BACKGROUND AND OPPORTUNITIES

We now characterize periodic workloads (Section 2.1), discuss cloud pricing models (Section 2.2), breakdown cloud vendor costs for analytics workloads (Section 2.3), and present cost saving opportunities and the goals of this work (Sections 2.4–2.5).

2.1 Characterization of Analytics Workload

We aim to save money for *periodic analytical* workloads in the cloud, which execute at distinct intervals. They are typical for ETL, filling in reports, or running business intelligence tools [4, 6, 40, 39, 35]. We divide a typical analytics pipeline deployment into 4 stages:

Analytical Workload Cloud Setup. A collection of ETL pipelines periodically **(1)** collects data from databases (and other sources) and **(2)** loads it into the target system, a cloud OLAP database optimized for analytical workloads. The periodic workload **(3)** runs against the OLAP system, and the results are **(4)** materialized in reports, dashboards, intermediate tables and views, and other formats. Note these 4 stages represent a wide variety of workloads.

2.2 Cloud Data Warehouses and Pricing Models

To execute these workloads in the cloud, users can choose between two OLAP systems: *infrastructure-as-a-service* (IaaS) and *platform-as-a-service* (PaaS). These systems are priced according to different pricing models, and we study saving opportunities across them.

IaaS vs PaaS. In IaaS, users allocate virtual machines on which they deploy an OLAP database. The VM charges for compute time and users manually deploy, maintain, and operate the OLAP system (e.g., Trino [83, 36], Apache Pinot [64, 27], Apache Hive [87, 21]) on top of the provisioned cloud resources.

In contrast, PaaS offerings (e.g., Google BigQuery [7], AWS Redshift [45], Microsoft Azure Synapse [34], or Snowflake [55]) directly provide OLAP databases to users and charge a premium over IaaS in exchange for handling deployment and maintenance of software. In many cases they also offer higher performance.

Pay-per-compute vs Pay-per-byte. PaaS offers two pricing models: *pay-per-compute* and *pay-per-byte*. The first, like IaaS, charges for the amount and duration of allocated computing resources. The second charges for the bytes read by a query regardless of runtime. Table 2.1 shows prices for popular databases across both pricing models. We note that all major current pay-per-byte databases charge $5/TB, so we use only one such database in our evaluation.

2.3 Breakdown of Cloud Costs

Deploying the analytical pipeline (the 4 stages above) and executing analytic workloads in the cloud incurs a number of cloud costs. These costs are distilled into the following categories:

- **Blob Storage cost**: storing data in blob storage (such as Amazon S3, Google Cloud Storage) or PaaS managed storage.
- **Read/Write cost**: API calls to read or write data in blob storage.
- **Loading cost**: compute resources consumed by the machine while loading data into the OLAP database.
- **Egress cost**: cloud vendors charge per-byte for data moved outside their cloud or between regions.
- **Query Processing cost**: cost of query execution in an OLAP database; can be billed per-byte or per-compute

We show the prices for each category set by cloud vendors as of April 2023 in Table 2.1.

Pricing Model	Database	Cost
PC	**Amazon Redshift–ra3.xlplus**	**$1.086/hour**
PC	Amazon Redshift–ra3.4xlarge	$3.26/hour
PC	Azure Synapse 100 DWU	$1.20/hour
PC	Azure Synapse 500 DWU	$6/hour
PC	Snowflake Small Warehouse (AWS US-East)	$4/hour
PC	Snowflake Large Warehouse (AWS US-East)	$16/hour
PC	n2-standard-32 VM (GCP)	$1.55/hour
PC/PB	Amazon Redshift Spectrum	RS Cost + $5/TB
PB	**Google BigQuery**	**$5/TB Scanned**
PB	Amazon Athena	$5/TB Scanned
PB	Azure Synapse Serverless	$5/TB Scanned

Cloud Vendor	GCP	S3	Azure
Storage ($/GB/mo)	$0.023	$0.023	$0.018
Writes ($/10k ops)	$0.05	$0.05	$0.065
Reads ($/10k ops)	$0.004	$0.004	$0.005
Egress ($/GB)	$0.120	$0.09	$0.087

Table 2.1: Prices for databases in pay-per-compute (PC) and pay-per-byte (PB), blob storage, read/write from blob storage, and data egress (GCP us-east1, S3 us-east, Azure hot storage). Systems used in evaluation are bolded.

During stage **1**, users pay the cost to transfer data to the cloud and to store data in that cloud. This transfer may incur egress costs. During stage **2**, users pay the cost of API calls to blob storage and the cost of loading data into the OLAP database. During stage **3**, users pay the cost of query execution. Finally, during stage **4**, users pay the cost of any potential egress needed to materialize the results. We focus on stages 2 and 3 because they directly involve OLAP cloud databases.

2.4 Cost Saving Opportunity and Challenges

The insight we exploit is that it is possible to save on execution costs by migrating compute- or IO-bound queries to an OLAP system with a *beneficial* pricing model. We extend this reasoning to subqueries as well, achieving cost savings by executing subqueries in their *beneficial* pricing models. These insights present an opportunity for cost savings, which we

now make more precise.

Given the size scanned by a query S, query runtime R and per-byte cost α_S and per-compute cost α_R, we observe that $\alpha_S \times S = \alpha_R \times R \implies S = \frac{\alpha_R}{\alpha_S} \times R$. This shows that a query which ran for R seconds would cost the same in a pay-per-compute pricing model as one that read $\frac{\alpha_R}{\alpha_S} \times R$ bytes in a pay-per-byte model. This corresponds to the line in Figure 1.1 which separates queries cheaper in pay-per-compute from those cheaper in pay-per-byte.

Estimating R for queries is difficult because query optimizer estimates are noisy and the semantic meaning of plan cost is relative to other costs and does not directly correlate to runtime [51, 70, 91]. Instead, we leverage the *periodic* nature of the workload to collect this information for queries via a *profiling* stage (described in Section 5.2). This information is then used by the inter- and intra-query algorithms to achieve cost savings.

2.5 Problem Statement and Goals

Our goal is that users will submit their usual, periodic workload, receive the same results, and pay less or at most the same cost in the end. Our analysis will exploit **O1**–**O4** to achieve this lower cost.

Formally, given a workload of periodic queries Q and tables T that executes on a source execution backend X_s, we take the workload and a set of execution backends (each of which may correspond to different pricing models) and find inter- and intra-query plans (opportunities **O1** and **O2**) that save costs. Users do not need to change any existing query or loading (ETL) script because our implementation takes care of all necessary data movement. Our implementation uses a *profiling* stage to gather all necessary information and systematically identifies changes to SQL syntax (opportunity **O3**) that further save costs. During this process, we also find opportunities to save on the premium of PaaS by using DuckDB deployed on top of IaaS (opportunity **O4**).

Scope. Our goal is to provide a proof-of-concept that demonstrates empirically that we can achieve significant savings by scheduling queries and subqueries across clouds. As such there are some extensions to our work that are nevertheless out of the scope of this paper.

- This analysis could be extended to balance workload cost and runtime, but this is orthogonal to our goal of understanding cost saving opportunities, so we focus exclusively on saving money.
- There were many other database options for our evaluation: for example, Redshift Spectrum charges users per-compute for the cluster and per-byte scanned from S3; BigQuery Omni [25] enables users to run BigQuery on data stored in other clouds; and, Amazon Athena provides a per-byte database in AWS. We choose and carefully configure specific cloud databases as representatives of pricing models to conduct our study, but we do not aim to exhaustively evaluate all combinations of cloud OLAP systems or all possible configurations of these enterprise-grade systems as this does not aid our evaluation of **O1**–**O4**.
- Our implementation pays repeated *migration costs* to copy, load, and store data temporarily in a second execution backend while queries run before deleting this copy of data. Our analysis could be extended to adjust to users changing the source backend to save costs, but that lies outside the scope of this work.

In Sections 3 and 4 we present the algorithms to exploit inter- and intra-query execution (**O1** and **O2**). Section 5 explains the profiling phase, how we exploit **O3** and **O4**, and how our implementation is engineered to work transparently to users.

CHAPTER 3

INTER QUERY EXECUTION PLAN

In this section, we present the inter-query algorithm. We provide the setup (Section 3.1), present an optimal inter-query algorithm and the intuition for our proposed greedy strategy (Section 3.2), before presenting our proposed algorithm (Section 3.3).

3.1 Algorithmic Setup and Goal

We consider a periodic analytical workload with tables T and queries Q. Initially, T is entirely stored in a *source execution backend* X_s. Customers pay the cost of storing T in X_s, executing Q in X_s and the egress cost of retrieving the results, when that is necessary.

Given a second execution backend, X_d, the algorithm's goal is to identify a subset of queries, $W \subset Q$ to execute in X_d so that the overall cost of executing the workload is lower than when executed only on X_s. To run a query in X_d the algorithm must migrate all tables from X_s that the query depends on and then execute all queries in X_d, so these migration costs must be taken into consideration. The algorithm requires the following input data:

- The set of tables T and queries Q in the workload.
- The source X_s and destination X_d execution backends e.g., AWS Redshift or Google BigQuery.
- A set of *cloud prices* P. These are prices for blob storage (p_{blob} per-byte), data retrieval (p_{read} read cost, p_{write} write cost), and execution backend costs (p_{sec} for per-compute pricing models, p_{byte} for per-byte pricing models) e.g., p_{byte} = \$5/TB in Google BigQuery, p_{blob} = \$0.023/GB/month in S3, p_{write} = \$0.05/10,000 operations and p_{read} = \$0.004/10,000 operations in GCP.
- The egress cost e per-byte to move from X_s to X_d e.g., \$0.09/GB out of AWS.
- A function s which returns size of a given table. This is easily measured and is defined for

sake of notation.

- A function C_{X_i} which takes an execution backend, X_i and a query q and returns the cost of q in X_i.

All the above inputs are easy to obtain except for C_{X_i}, which is obtained during a profiling stage. This is explained in Section 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 T representing tables and Q representing queries. We draw an edge $(t \in T, q \in Q) \in E$ if query q needs access to base table t to execute.

We next assign weights σ_q to each vertex $q \in Q$ 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* associated with that table. This cost represents 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:

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

In Figure 3.1 we show an example of this model with three tables $T = \{t_1, t_2, t_3\}$ and four queries $Q = \{q_1, q_2, q_3, q_4\}$. 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 which maximizes *query savings*. Concretely, for $S \subset T$ let $N(S) = \{q \in Q | \forall (t, q) \in E, t \in S\}$. Further, let $N^{-1}(q) = \{t \in T | (t, q) \in E\}$. Our goal then is to solve the following optimization problem:

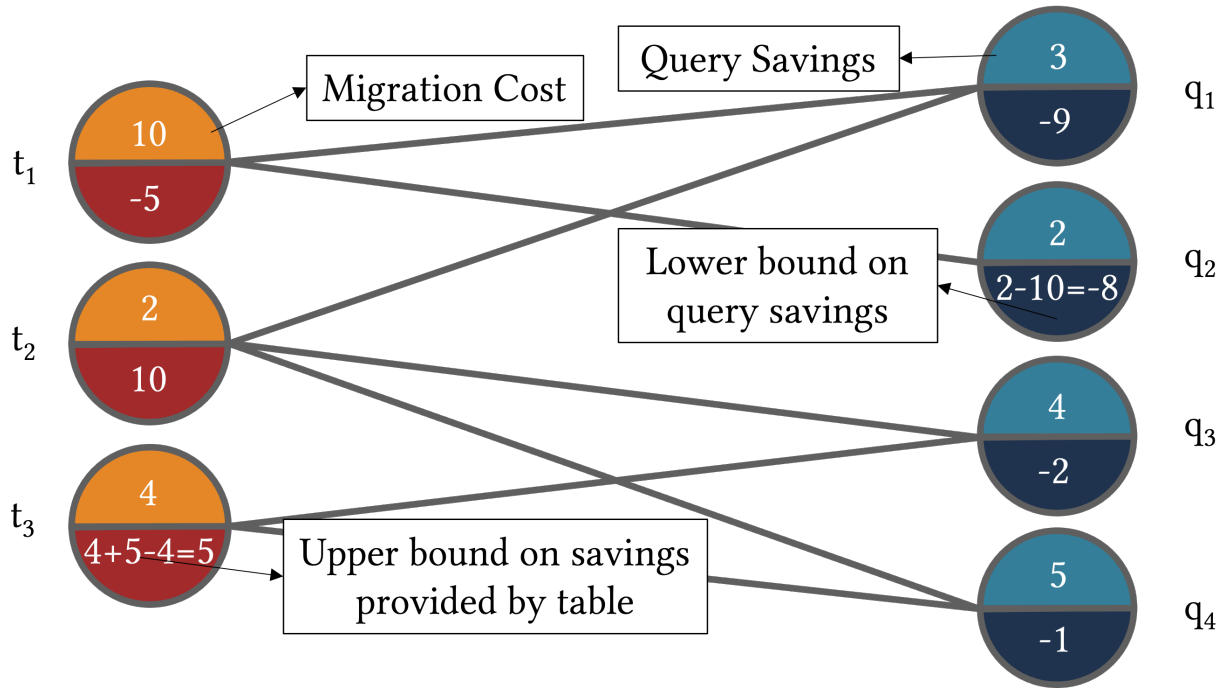


Figure 3.1: Bipartite model for the inter-query algorithm. The top number on a node shows its query savings σ_q or migration cost μ_t. The bottom value shows v_t or v_q for that table or query. We show how this value is calculated for t_3 and q_2.

$$\max_{S \subset T} \sum_{q \in N(S)} \sigma_q - \sum_{s \in S} \mu_s$$

For example, in Figure 3.1 we optimally move tables t_1, t_2 and queries q_3, q_4 to X_d, achieving overall savings of (4+5)-(2+4)=\$3.

3.2 Designing the Algorithm

We first present an optimal, min-cut based algorithm. Then we present the intuition for our greedy strategy.

3.2.1 Optimal Solution

We use a similar approach as Heller et. al. [62]. We construct a capacity function $c : E \to \mathbb{R}$ and a bipartite graph G as above; nodes on the left are tables and on the right are queries.

We draw *directed* edges with infinite capacity from tables to queries corresponding to 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 create a sink node b and draw edges from every query q_j to b; $c((q_j, b)) = \sigma_{q_j}$ The algorithm computes the min-cut $C = (A, B)$ and designates the set of queries and tables in B to migrate to X_d. We define $c(C)$ to be the capacity of edges spanning a cut C.

We note that every potential solution to the inter-query optimization problem is defined uniquely by the set of queries W that move to X_d. We make two claims showing that the solution corresponding to the min-cut of this graph achieves the largest savings.

Claim 1. For every potential solution $W \subset Q$ there is a cut $C_W = (A, B)$ of finite capacity and vice versa.

Claim 2. A solution with queries W saves $\sum_{q \in Q} \sigma_q - c(C_W)$.

Proof of Claim 1. A cut has infinite capacity only if a query and a table it depends on are on opposite sides of the cut. Given a solution $W \subset Q$, let $B = W \cup N^{-1}(W) \cup \{b\}$ and A be the remaining nodes. All edges spanning the cut are outbound from a or inbound from b, so $C_W = (A, B)$ will have finite capacity.

Equivalently, for a given finite cut $C = (A, B)$, let $W = B \cap Q$, i.e. W is all queries in B. Then $N^{-1}(W) = B \cap T$, since otherwise an infinite capacity edge would span A and B as there would either be an edge from A to W or from B to some query in A. If $\overline{W} = Q \setminus W$, we can write $B = \{b\} \cup W \cup N^{-1}(W)$ and $A = \{a\} \cup \overline{W} \cup Q \setminus N^{-1}(W)$.

Proof of Claim 2. Using the form for a cut given in Claim 1, for a set of queries W and a resulting cut $C_W = (A, B)$ we can rewrite the exprssion for $c(C_W)$ to show that the solution maximizing savings equivalently produces a min-cut of the graph.

$$c(C_W) = \sum_{q \in \overline{W}} \sigma_q + \sum_{t \in N^{-1}(W)} \mu_t = \sum_{q \in Q} \sigma_q - \sum_{q \in W} \sigma_q + \sum_{t \in N^{-1}(W)} \mu_t$$

$$\implies \sum_{q \in W} \sigma_q - \sum_{t \in T_W} \mu_t = \sum_{q \in Q} \sigma_q - c(C_W)$$

Complexity of Optimal Algorithm. Using the Ford-Fulkerson algorithm [59], the complexity is $O(|V||E|^2)$. Dinitz's algorithm brings this complexity down to $O(|V|^2|E|)$ [57].

3.2.2 Intuition for the Greedy Strategy

We now present the intuition for a greedy algorithm, which finds the optimal solution well in practice and is more efficient than the optimal algorithm. The greedy algorithm iteratively removes the table with the lowest potential to save costs, defined for a table t as the upper bound on savings achievable by moving queries dependent on t to X_d.

At each step, the algorithm computes the upper bound for each table, removing from consideration the table with the lowest potential. The algorithm then records the cost of the resulting inter-query plan. Finally, after there are no more tables to remove, the algorithm checks which inter-query plan resulted in the cheapest plan.

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

Analogously, we define the values v_q for each query q as the *query savings* of that query 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 the achievable 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 representation, and remove all outbound edges from $N^{-1}(q)$. In Figure 3.1 we compute v_t and v_q and present them in the lower half of each node, e.g., v_{t_1} is the savings of q_1 and q_2 minus the cost of t_1, or $3 + 2 - 10 = -5$.

input: $X = \{X_s, X_d\}$, cloud prices P, e egress cost, T', Q', s, C
output: T_f, Q_f

Function `InterQueryProcessing`(X, P, e, T, Q, s, C)**:**
  Let σ_q, μ_t be defined as in Equations 3.1 and 3.2 using s and P;
  Let T', Q', T_f, Q_f = ReducePlan(X, P, e, T, Q, s, C) ;
  Remove all outbound edges from T_f ;
  Let planCosts = {} ;
  while $T' \neq \emptyset$ **do**
    For $t \in T'$, let $v_t = (\sum_{q \in N(t)} \sigma_q) - \mu_t$;
    Let $t' \in T'$ be the table with minimium $v_{t'}$;
    $T' = T' - \{t'\}$;
    $Q' = \{q | N^{-1}(q) \subset T'\}$;
    $T', Q', T'_f, Q'_f = ReducePlan(X, e, T', Q', s, f)$;
    Let $T' = T' \cup T'_f$; Let $Q' = Q' \cup Q'_f$;
    planCosts$[T' \cup T_f, Q' \cup Q_f]$ = cost$(T' \cup T_f, Q' \cup Q_f)$;
  Let T_θ, Q_θ yield the minimial cost plan in planCosts ;
  return T_θ, Q_θ;
Function `ReducePlan`(X, P, e, T', Q', s, C)**:**
  Let σ_q, μ_t be defined as in Equations 3.1 and 3.2 using s and P;
  Let $Q' = \{q | \sigma_q > 0\}$;
  Let $T_f, Q_f = \{\}, \{\}$;
  For $t \in T'$ let $v_t = (\sum_{q \in N(t)} \sigma_q) - \mu_t$;
  For $q \in Q'$ let $v_q = \sigma_q - \sum_{t \in N^{-1}(q)} \mu_t$;
  while $T' \neq \emptyset \wedge \exists v_t < 0 \wedge \exists v_q > 0$ **do**
    Let $U = \{t | v_t < 0\}$; Let $V = \{q | v_q > 0\}$;
    Let $T' = T' - U$;
    Let $Q' = Q' - \{q | q \in N(t) \forall t \in U\}$;
    Let $T' = T' - \{t | t \in N^{-1}(q) \forall q \in V\}$;
    Let $Q' = Q' - V$;
    Let $Q_f = Q_f \cup V$;
    Let $T_f = T_f \cup \{t | t \in N^{-1}(q) \forall q \in V\}$;
    For $t \in T'$ let $v_t = (\sum_{q \in N(t)} \sigma_q) - \mu_t$;
    For $q \in Q'$ let $v_q = \sigma_q - \sum_{t \in N^{-1}(q)} \mu_t$;
  return T', Q', T_f, Q_f;

Algorithm 1: Inter-query greedy algorithm

3.3 Inter-Query Algorithm

The greedy algorithm, shown in Algorithm 1, first invokes ReducePlan. This procedure computes v_t and v_q (lines 20–21) and then removes tables with $v_t < 0$ from consideration and automatically moves queries with $v_q > 0$ to X_d (lines 23–29). The algorithm repeats the process until either no tables or queries meet this criteria or there are no tables left (line 22). ReducePlan returns those tables and queries that are added to the final set in T_f and Q_f and returns the remaining set of tables and queries to consider in T', Q' (line 32). The algorithm then removes all outbound edges from the set of tables assigned to move to X_d (line 4).

The algorithm then proceeds greedily. While there are still tables to be considered (line 6), the algorithm computes v_t (line 7) and removes the t with minimal v_t, assigning it to remain in X_s (lines 8–9) before removing nodes corresponding to $N(t)$ (line 10). We then call ReducePlan again to prune away those tables with negative upper bounds and identify queries with positive lower bounds (line 11). The algorithm records the overall workload cost the current plan (line 13), associating this cost with the set of tables and queries used. This process is repeated until all tables have been removed from the set. Finally, the subset of tables and queries which produced the cheapest plan are returned (lines 14-15).

Complexity Analysis. Let $n = |T|$ and $m = |Q|$. Computing v_t or v_q is linear in $n + m$ and at worst we remove only one table per iteration, yielding a worst-case complexity of $O(n(n + m))$. The optimal algorithm, with complexity $O(|V|^2|E|)$, is both an order of magnitude less efficient and is dependent on the number of relationships between queries and tables. The greedy algorithm's complexity is independent of query complexity.

The greedy solution is not optimal in all cases; we evaluate the algorithm across 48 workloads on 2 setups at 1TB and 2TB, both with and without IaaS, producing 192 instances. Our greedy strategy finds the optimal solution in 188 of these 192 instances.

CHAPTER 4

INTRA QUERY EXECUTION PLAN

We now present the intra-query algorithm. Intra-query plans achieve significant savings on the query level, and when a few queries dominate workload costs intra-query plans significantly reduce workload costs. We discuss the setup and then present the algorithm.

4.1 Algorithmic Setup and Goal

The setup is similar to the inter-query setup (see Section 3). However, instead of receiving a query workload this algorithm only receives a single query q as input. It aims to identify a subquery that saves money when migrated from X_s to X_d after accounting for any incurred migration costs.

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 cut of q so that one subquery executes in X_s and the other in X_d such that the total query cost, including migration cost, is lower than running the entire query in X_s.

4.2 Identifying Profitable Cuts

In this section, we introduce the insight we use to identify profitable cuts. Let $T = (V, E)$ be a query plan. We use the same pricing notation as for the inter-query setup: p_{byte}, p_{sec}, and e for per-byte, per-compute, and egress prices. We also define the migration cost μ_t for a table as in Equation 3.2. For notation we define s as the size of each table and $rs(v)$ as the maximum row size defined by table schema for a node v. Finally, when we make a cut

at $v \in V$, we refer to the subgraph **b**elow as $S_{\mathbf{b}}(v)$ and the subgraph **a**bove as $S_{\mathbf{a}}(v)$. We now highlight some important definitions.

- $f_w(v)$ returns the output cardinality of $v \in V$
- $f_r(v)$ returns the runtime of $S_b(v)$
- $\mathcal{L}(v) = \{u \in S_a(v) | u \text{ is a leaf}\}$; $C_s(v) = \alpha_s \sum_{u \in \mathcal{L}(v)} f_w(u) rs(u)$
- $C_r(v) = p_{sec} f_r(v)$; $C_m(v) = \sum_{t \in \mathcal{L}(v)} \mu_t$

C_s represents the cost of the upper subquery in pay-per-byte. C_r represents the cost of the lower subquery in pay-per-compute. C_m represents the migration cost incurred to move from X_s to X_d.

These values require C_{X_i} which provides the cost for this query in all execution backends. We assume that the algorithm has access to C_{X_i} and f_w and explain in Section 5 how these are obtained. The algorithm's goal then is to find $v \in V$ such that:

$$C_r(v) + C_m(v) + C_s(v) < C_{X_s}(q) \tag{4.1}$$

The algorithm aims to find the cheapest hybrid plan, represented on the left, that costs less than naively executing the query in X_s.

Insight. Naively, we could determine the optimal hybrid query plan if one exists by making a cut at every operator and executing each resulting hybrid query plan. This approach requires we pay query processing costs and any potential migration costs for each hybrid plan, and the number of possible plans grows with the size of the query. Our goal is to find cheaper hybrid query plans while minimizing the overhead costs needed to find this plan.

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 restructure Equation 4.1 and compute the maximum savings achievable by a hybrid plan. We consider those operators with positive savings opportunity and use this value to guide

what candidate operators we evaluate.

Calculating Savings Opportunity. We compute *savings opportunity* o_v as $p_{sec}f_r(v) < C_{X_s}(q) - (C_m(v) + C_s(v))$ by reordering Equation 4.1. C_{X_i} and f_w are provided from the profiling phase (Section 5.2), so we can compute the right hand side directly.

We draw two conclusions. First, if $o_v < 0$, the hybrid plan produced from a cut at v will not be cheaper than the baseline cost, $C_{X_s}(q)$. Second, the only way to determine if a hybrid plan will produce savings is to compute f_r, which requires in particular paying profiling costs to run the hybrid plan. Our algorithm thus also wants to reduce the number of times it computes f_r.

input: $T = (V, E)$, $X = \{X_s, X_d\}$, p_{byte}, p_{sec}, e, f_w, $C_{X_s}(q)$
output: Cheapest Execution Plan

Function `IntraQueryAlg`(T, X, p_{byte}, p_{sec}, e, f_w, $C_{X_s}(q)$)**:**
  for $u \in V$ **do**
    Let $C_m(u)$, $C_s(u)$ be as defined earlier ;
    Let $o_u = C_{X_s}(q) - (C_m(A) + C_s(B))$;
  Let *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 $o_v < a_u$ **then**
        Remove v from *candidates* ;
    for $v \in$ *candidates* $\wedge$ v *upstream of* u **do**
      $o_v = o_v - p_{sec}f_r(u)$;
      if $o_v < 0$ **then**
        Remove v from *candidates* ;
  Return u with maximal $a_v > 0$ if one exists or $C_{X_s}(q)$;

Algorithm 2: Intra-query algorithm

Updating Opportunity Values. Measuring $f_r(v)$ allows us to infer information about o_u for other nodes. For example, consider a node u_1 which sits downstream of another node n_w. Then $f_r(u_1) \geq f_r(u_2)$, so if we compute $f_r(u_2)$, the opportunity of u_1 must decrease by $p_{sec}f_r(u_2)$ since it must pay at least that much in runtime cost. Additionally, if we calculate the real savings for a hybrid plan produced from v, we remove all candidates with o_u less

than this real savings because cuts at those nodes cannot produce cheaper hybrid plans. These insights allow us to remove multiple candidates per iteration, reducing the number of invocations to f_r needed to find the optimal solution.

Consideration of More Complex Setups. It is sufficient to identify intra-query opportunities to consider hybrid plans from one cut across two backends. Using multiple cuts or backends may yield greater savings but are outside the scope of of this work.

4.3 Intra-Query Algorithm

Using these insights and definitions, we present the intra-query algorithm in Algorithm 2. The algorithm computes the opportunity o_u for every node u in the query plan (lines 2-4). It removes those nodes which have negative opportunity (line 5), leaving a set of candidates. It iterates over the list of candidates from largest to smallest opportunity (line 6–7). Computing f_r for all candidate may be cost-prohibitive, so iterating from largest to smallest opportunity minimizes the potential savings not achieved.

For each candidate u, the algorithm computes $f_r(u)$, the real savings, a_u, (lines 8–9) and updates the opportunity for other candidates (lines 10–16). It re-orders the candidates by opportunity and repeats, until all candidates have been checked or after K iterations (line 6). At the end, it returns the cheapest hybrid query plan executed or the baseline plan (line 17).

Complexity Analysis and Discussion of Optimality. The algorithm removes at least one candidate per iteration, so the worst-case complexity is $O(|V|)$. The algorithm parses a single query plan, which may not be optimal. However, the algorithm will never choose a hybrid plan that costs more than the baseline to execute.

CHAPTER 5

ARACHNE OVERVIEW

An isolated evaluation of the inter- and intra-query algorithms would presume that all inputs including baseline query costs were *a priori* collected and would ignore any complications from interfacing with multiple cloud OLAP systems. To make this setup more realistic, we implement a prototype, Arachne, that implements these two algorithms, collects all input information, and interfaces with cloud OLAP systems. Building this artifact helped us uncover and exploit two other substantial opportunities for savings, **O3**–**O4**, which only manifest from particular interactions with cloud systems. In this section we discuss how Arachne collects input information, interfaces with OLAP databases, and exploits **O3**–**O4**.

5.1 Overview

Figure 5.1 shows Arachne's architecture. Arachne takes as input an unmmodified periodic SQL workload. Arachne implements and executes the inter- and intra-query algorithms of Sections 3 and 4 in the **savings module**. The **profiling module** gathers query costs needed by the savings module and observes the impact of SQL syntax on query cost (**O3**). The **execution engine** takes a SQL query and source and destination backends, optionally rewrites the query using Apache Calcite [46], executes the query in the source backend, materializes the results in a compressed Parquet file, and moves it to the destination backend.

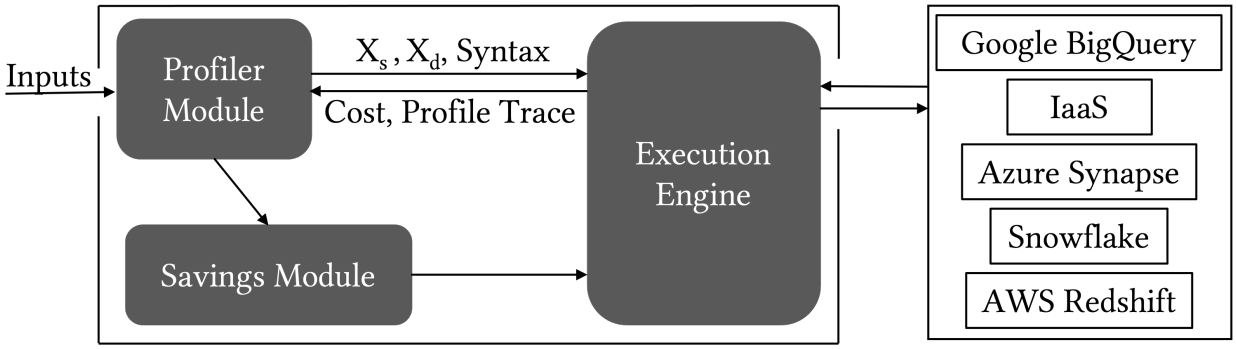


Figure 5.1: Arachne overview, with system components and execution backends

Extending the execution backends. Arachne supports three execution backends: Google BigQuery, AWS Redshift, and DuckDB. These backends are sufficient to study pay-per-compute, pay-per-byte, and IaaS pricing models. However, Arachne can be easily extended to new execution backends.

Implementation. Arachne is built in Java and uses Apache Calcite [46] to convert SQL into logical query plans, perform query optimization, and convert logical query plans back into SQL.

Next, we motivate and present the profiler, which collects necessary information for **O1**–**O2** and present our observations on the performance and cost impact of SQL syntax and DuckDB (**O3**–**O4**).

5.2 Profiler Module

Goal of the Profiler Module. The profiler module gathers the baseline cost for each query in each execution backend and the output cardinality for each operator of a query plan.

Query optimizers must estimate these two parameters when finding a query plan, so one approach could be to use query optimizers to obtain these values. Unfortunately, query optimizers' estimates are well-known to be noisy, and the semantic meaning of a plan *cost* is relative to other plans and not reflective of runtime, so we cannot directly compare the monetary cost of plans. Estimating query runtime and output cardinality remains a difficult task [71, 91, 78].

Profiling Approach. Instead, Arachne *profiles* by executing the workload on each execution backend and *measuring* the actual query runtime R and the cardinality at the output of each operator S. These profiles provide accurate data for future workload executions, but do incur additional migration and query processing costs. Profiles can provide accurate information for many iterations because query runtime, size scanned, and thus query cost often do not

vary significantly as data incrementally changes, but when data does change significantly re-profiling will be necessary, just as data statistics must be kept up-to-date for database query optimizers. Luckily, it is possible to measure R and S on a small workload sample and extrapolate to the original dataset size without introducing significant error. This is because despite the difficulties of extrapolating query runtime from samples (we make the connection to the problem of sampling joins [51]) the costs of profiling are dominated by *pay-per-byte* pricing models, which depend only on the input data and not the query runtime. We show empirically in Section 6 that the extra profiling cost is quickly compensated by the savings in workload execution. Furthermore, we can collect these profiles during iterations of the periodic workload to reduce the extra profiling costs paid by users.

Additionally, many periodic workloads are latency insensitive. For example, if a midnight workload that produces a report will be consumed the next morning at 8am, it does not matter to the analyst if the report is ready at 2am or 3am; however, the organization would prefer to take an extra hour if that means they save on costs. Even though profiling and cheaper workload execution may add latency to workload execution, latency-inensitive periodic queries will not be significantly affected.

Output Cardinalities and Graph Matching. Operator cardinality is independent of execution backend, so Arachne obtains cardinalities from a DuckDB profile. However, DuckDB's physical plan may differ from those of other execution backends. Thus, we must match operators across query plans to label them with the correct cardinality. DuckDB does not re-order operators between subqueries defined in SQL, so Arachne converts a query plan back into SQL, writing all operator trees as nested subqueries. DuckDB's physical plan thus precisely matches the original query plan, enabling Arachne to assign cardinalities to operators.

5.3 O3: The Impact of SQL Syntax

While implementing Arachne, we observed that different SQL syntax, such as Arachne-produced SQL, changes the physical plan chosen by query optimizers, impacting performance and cost (**O3**). We study these differences for cost savings. Arachne-produced SQL writes join trees and common table expressions as nested subqueries rather than using a single SELECT statement or the WITH statement. The optimizer then reorders joins, unions, and other operator trees and treats treats subquery scans differently.

Arachne detects opportunities for cost saving in the profiling module by executing each query in both the original and Arachne-produced SQL syntax. It then uses the syntax that produces the cheaper plan in the final workload execution plans. We present an evaluation of the impact of SQL syntax in Section 6.

5.4 O4: Cost Savings with IaaS

Arachne hides the underlying execution backend behind its SQL interface, so we use Arachne to explore saving opportunities from IaaS by deploying DuckDB on IaaS. Arachne copies data–stored as Parquet files–to local VM disk. DuckDB can directly query Parquet files and materialize intermediates as compressed Parquet files. Arachne treats this as another execution backend and collects profiles for queries in it as such. This information is then considered along with other PaaS offerings in the same cloud, so Arachne can identify the cheapest manner in which a query can be executed in a cloud environment. In this way, Arachne can determine if it can save money by utilizing IaaS over PaaS and then can transparently deploy and execute workloads on IaaS while removing the burden of deployment and maintenance from users.

5.5 Cost-Relevant Implementation Details

Data Transfer Between Cloud Vendors. Arachne avoids the expensive cloud vendor-provided tools for data transfer between clouds, e.g., AWS DataSync, by implementing a simple, parallel cloud transfer tool which uses blob storage APIs to execute parallel multipart downloads and uploads on IaaS. Our transfer tool on a GCP n2-standard-32 VM transferred a 615.3GB dataset for $0.58; that transfer in AWS DataSync costs $7.69.

Data Compression and Cost Savings. Migration costs largely charge per-byte. Arachne compresses data in Parquet files to migrate fewer bytes and thus reduce costs.

SQL Syntax Compatibility. Arachne-produced SQL may not be at first compatible with target backends due to differences in SQL dialects, e.g., BigQuery requires column names to contain only alphanumerics or underscores requiring specific fixes, or query plans have internally conflicting column names, such as an asterisk. Arachne builds and applies syntax rules to ensure compatibility for Arachne-produced SQL queries.

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

CHAPTER 6

EVALUATION

In this section, we answer the following research questions:

- **RQ1:** Does Arachne save money via the inter-query algorithm? (**O1**) and what is the effect of IaaS vs PaaS? (**O4**).
- **RQ2:** Does Arachne save money via intra-query analysis? (**O2**).
- **RQ3:** What is the effect of SQL syntax on query savings? (**O3**).
- **RQ4:** To verify that the results are not brittle to specific prices chosen by cloud vendors, we explore the effect of these prices via a *what-if* analysis.

6.1 Experimental Setup

Execution Backends and Data Format. We use Google BigQuery to represent the *pay-per-byte* pricing model and Amazon Redshift to represent *pay-per-compute*. In Redshift, cluster size impacts cost and performance, so we explore the GA1 and GA4 setups which consist of 1- and 4-node ra3.xlplus Redshift clusters respectively. We deploy DuckDB on a GCP VM with 16 vCPU, 190GB RAM, and a 1.4TB disk costing $1.48/hour. We configure Redshift, BigQuery, and DuckDB optimized according to docs and best practices [38, 41, 9, 8], i.e. we enable Redshift's automatic view materialization to reduce query runtime and cost [2].

Because many data lakes support open data formats [13, 81] we consider data stored in Parquet files [26] and compressed with the Snappy algorithm [30]. When using Parquet files, we create external tables in BigQuery pointing to these files in blob storage. Redshift loads Parquet files directly from S3, and files are directly copied to IaaS local disk from blob storage. 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. Finally, we

consider data loaded into BigQuery, which is free but significantly increases runtime; loading 1TB took 12 minutes whereas creating external tables took only 20 seconds. Additionally, data is now stored in a closed format and only accessible either via BigQuery's SQL interface, which will add query processing costs to access data, or BigQuery's Storage Read API [11, 12] that is much more expensive than blob storage API costs.

Workload. We use 48 different workloads derived from TPC-DS [74], an analytical workload that consists of 24 tables and 99 queries. We generate a workload by removing a single table from the TPC-DS dataset, creating a 23-table dataset with a subset of the original 99 queries, on average about 60 queries. Each workload will have a unique query composition: some will be dominated by compute-bound queries, others by IO-bound queries, so we can explore the impact of workload composition on saving opportunities. This generation process produces 24 workloads. We use a 1TB and a 2TB version of the original TPC-DS, creating 48 workloads.

To evaluate the intra-query algorithm, we use TPC-DS queries and we author queries on the TPC-DS and LDBC-SNB dataset [22] at 100GB. We explain these in detail in the respective section.

Setting X_s and X_d. With current prices, Redshift is significantly cheaper than BigQuery. As a result, if we set X_s to be Redshift, there are few opportunities to identify cost saving opportunities. Instead, we set X_s to BigQuery, where all data resides initially, and consider opportunities to migrate to Redshift or IaaS (DuckDB).

Metrics. We measure the monetary cost in US dollars and runtime in time units. Our cost measurements accounts for all applicable cloud costs (see Section 2.3) and validate it by checking the breakdown of costs that cloud vendors used to charge our account. We use VMs collocated in the same cloud region as blob storage buckets to prevent regional transfer costs. Because tables are stored in compressed Parquet files, we account for the benefits of compression on migration costs such as data egress or data retrieval. Unless we indicate

otherwise, we use cloud costs as of April'23 in Table 2.1.

6.2 RQ1. Inter-Query Processing

We now explore opportunities **O1** and **O4**. First, we evaluate the cost opportunities of the inter-query algorithm in PaaS (Section 6.2.1). Then, we study the impact of IaaS execution backends (Section 6.2.2) before reporting incurred profiling costs (Section 6.2.3).

6.2.1 Cost-Saving Inter-Query plans

We run the inter-query algorithm as part of Arachne for each of the 48 workloads. We first provide an overview of the results, then zoom in on a few interesting workloads to understand the cost and runtime. Finally, we present an in-depth cost breakdown for a few workloads.

Overview. Table 6.1 shows an overview of the outcomes in setups GA1 and GA4. The Arachne column indicates workloads for which Arachne found an alternative plan that saved money compared to running the full workload in X_s. Arachne saves money in 43/48 cases; only 5/48 workloads remain in Google BigQuery in which case the Arachne plan costs the same as in BigQuery. Of those workloads where Arachne finds alternative cost saving plans, Multi refers to cases where Arachne found a plan with a mix of queries running in X_s and others running on X_d. Cases where all queries moved to Redshift are labeled as AWS. About 50% of workloads move completely to Redshift, indicating that queries achieve enough savings in that pricing model to compensate for egress costs. Arachne achieved savings of up to 55.4% in a single plan across all possible plan-types, and the multi-cloud plans chosen, in all but one case, achieve cost savings of 15%–35%. These plans utilize the pay-per-byte and pay-per-compute pricing models along with query characteristics to achieve these savings. We did not find significant differences between GA1 and GA4.

In-depth Cost and Runtime Analysis. We now focus on Multi plans, where queries run

Setup	Arachne	Multi	GCP	AWS	Total
GA1–1TB	**21**	6	3	15	24
GA4–1TB	**21**	6	3	15	24
GA1–2TB	**22**	5	2	17	24
GA4–2TB	**22**	5	2	17	24

Table 6.1: Inter-query plan-type by setup at 1TB and 2TB.

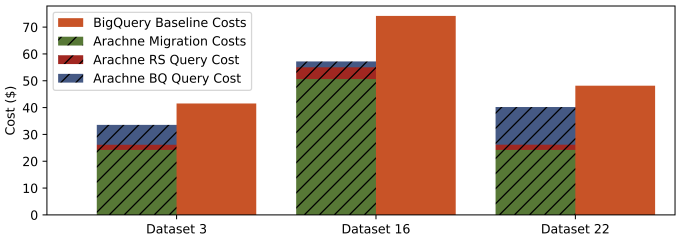
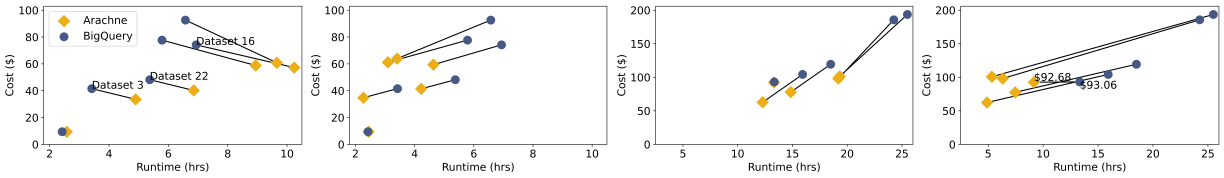


Figure 6.1: Breakdown of Arachne costs into migration costs, cost of queries in Redshift, and cost of queries in BigQuery, compared to the BigQuery baseline. Three Multi datasets (1TB GA1)–points labelled in Figure 6.2a.



(a) 1TB GA1 (b) 1TB GA4 (c) 2TB GA1 (d) 2TB GA4

Figure 6.2: Cost (USD) vs runtime (hours) for datasets with Multi plans over Amazon Redshift and Google BigQuery.

on a mix of BigQuery and Redshift, to understand the opportunities of combining *price-per-byte* and *price-per-compute* pricing models.

Figure 6.2 shows query runtime in hours on the x-axis and cost in USD in the y-axis. Each blue dot corresponds to the corresponding runtime and cost of one of the 48 workloads running in X_s, BigQuery in our experiments. For example, there are 6 of 48 Multi workloads in the GA1–1TB setup presented in Table 6.1, so there are 6 blue dots in Figure 6.2a. Along with each blue dot, we show in a yellow dot the corresponding runtime and cost of that workload when run on Arachne. Dots corresponding to the same workload are connected with a line to facilitate interpreting the results.

Multi workloads execute cheaper in Arachne than in BigQuery. If not, Arachne would have kept the workload in BigQuery. Consequently, all lines decrease from the blue dot, and the degree of its reduction corresponds to monetary savings. Interestingly, these plans achieve savings as high as 34% and over 15% for most workloads. In 2/48 workloads the savings are

marginal because Arachne migrates very few queries and tables which yield small savings.

Figure 6.2 shows that BigQuery's runtime is lower than Arachne when using GA1 because GA1 does not exploit all the parallelism available in the workload. When we change to GA4 we observe that Arachne's plan is both cheaper and faster than a BigQuery-only plan. In the larger cluster (GA4), loading times decrease, and Redshift fully exploits the parallelism available to complete the workloads much faster (and cheaper) than in BigQuery. Figure 6.2 shows the results for the 2TB workloads. The trends are similar to 1TB except that in this case Arachne is both cheaper and faster than BigQuery even in the GA1 case. Even though loading times are longer in GA1 2TB than 1TB, query processing is sped up enough to yield overall lower workload runtime.

Understanding cost breakdown in detail. We breakdown the Multi plans into migration costs and query costs for 3 datasets in 1TB GA1 along with their corresponding BigQuery baseline costs in Figure 6.1. We further group query costs by those which moved to Redshift and those which remained in BigQuery. There are diagonal lines through bars depicting Arachne chosen plans.

These three multi-cloud plans achieved 19, 22, and 16.6% savings over the BigQuery baseline respectively. The massive decrease in query processing cost as IO-heavy queries move to Redshift drive the workload savings, while migration accounts for the majority of multi-cloud plan costs. Egress comprises 90% of all migration costs, while 5–8% of migration costs are spent loading data into Redshift. The remainder is the cost of blob storage and data retrieval.

BigQuery Internal Tables. For internal tables, BigQuery charges users once for each table scanned, even if the table is scanned multiple times in the query. For external tables, BigQuery charges users for *each* table scan operator, even if multiple operators scan the same table. When data is stored internally, this causes the bytes scanned and thus the cost of a query to decrease despite the query being equivalent. While multi-cloud plans still

Sample %	15			25			50			100		
Dataset	Cost	Iter	Error	Cost	Iter	Error	Cost	Iter	Error	Cost	Iter	Error
0	28.62	1	0.03	44.47	2	0.03	84.64	3	0.03	158.39	5	0.0
1	27.84	2	0.03	43.13	2	0.03	81.97	4	0.03	153.32	6	0.0
2	24.71	2	0.03	39.7	3	0.04	75.04	4	0.04	139.03	8	0.0
3	16.52	3	0.05	24.78	4	0.05	46.17	6	0.05	83.94	11	0.0
4	25.43	2	0.04	39.11	3	0.04	73.55	5	0.04	135.59	9	0.0
5	25.85	2	0.03	39.89	3	0.03	75.29	5	0.04	140.2	9	0.0
6	26.85	2	0.03	41.52	2	0.03	78.58	4	0.04	146.48	7	0.0
7	15.64	N/A	0.08	22.56	N/A	0.08	39.96	N/A	0.09	67.6	N/A	0.0
8	27.13	2	0.03	41.89	2	0.03	79.45	4	0.04	148.16	7	0.0
9	28.41	1	0.03	44.08	2	0.03	83.85	3	0.03	157.01	6	0.0
10	28.45	1	0.03	44.12	2	0.03	84.01	3	0.04	156.87	5	0.0
11	19.59	N/A	0.06	29.2	N/A	0.06	53.63	N/A	0.06	95.3	N/A	0.0
12	27.32	2	0.03	42.27	2	0.03	80.21	4	0.04	149.56	7	0.0
13	28.33	1	0.03	43.96	2	0.03	83.58	3	0.03	156.33	6	0.0
14	28.74	1	0.03	44.58	2	0.03	84.86	3	0.03	158.98	5	0.0
15	22.36	N/A	0.04	33.9	N/A	0.04	63.25	N/A	0.04	116.61	N/A	0.0
16	25.01	2	0.04	38.42	3	0.04	72.42	5	0.04	134.75	8	0.0
17	9.87	145	0.08	13.68	200	0.06	24.76	362	0.07	43.5	635	0.0
18	28.24	1	0.03	43.69	2	0.03	83.06	3	0.03	155.71	6	0.0
19	28.46	1	0.03	44.1	2	0.03	83.92	3	0.04	156.62	5	0.0
20	28.43	1	0.03	44.13	2	0.03	83.94	3	0.03	157.22	6	0.0
21	26.46	2	0.03	40.88	2	0.03	77.32	4	0.04	143.85	7	0.0
22	19.54	3	0.05	29.66	4	0.05	55.24	7	0.05	99.73	13	0.0
23	27.74	2	0.03	42.98	2	0.03	81.61	4	0.04	152.39	6	0.0

Table 6.2: Profiling costs, iterations needed to compensate for profiling costs, and estimation error for 15, 25, 50, and 100% samples (GA1 1TB).

exist, the savings are negligible. However, we consider only 1 and 2TB datasets in these experiments, and in pay-per-byte models query cost increases linearly with dataset size, so this simply scales costs down as though the BigQuery price were lower. Our what-if analysis in Section 6.5 explores the impact of lower BigQuery prices further.

Summary. The results show that Arachne successfully exploits **O1**, the inter-query algorithm, to produce multi-cloud plans that saves 15%–35% in all but one case. These plans analyze query characteristics and the per-compute and per-byte pricing models to achieve these savings. In all cases, the greedy algorithm takes one to two seconds to run, so it is not a significant cost.

6.2.2 Studying the Impact of IaaS in Cost Saving

We next include DuckDB on IaaS in our inter-query analysis to evaluate the savings opportunity of IaaS over PaaS (**O4**). We deploy DuckDB on a VM with 16 vCPU, 190GB RAM, and a 1.4TB disk costing $1.48/hour (pay-per-compute) in order to execute more memory intensive queries. We deploy DuckDB in GCP, where data is stored originally, to avoid egress costs. This lets us compare PaaS with the cheaper but less convenient IaaS.

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

Using this setup, Arachne chose a GCP-only plan for all workloads in the GA1 and GA4 setups at 1TB and 2TB, i.e. queries never migrate to Redshift. Most queries were cheaper in DuckDB than BigQuery, and most queries were also cheaper in Redshift than in DuckDB because Redshift is faster. However, the absolute savings attainable by moving queries from X_s to X_d were reduced because of the DuckDB deployment that migration costs were no longer offset. However, Arachne still achieves up to 70% savings over the BigQuery-only (PaaS) baseline because running queries on IaaS was much cheaper and because these plans still exploit the compute- or IO-bound nature of queries across multiple pricing models. Hence, Arachne can identify opportunities and achieve significant savings with a transparent deployment of DuckDB on IaaS, all without separate user intervention, setup, deployment, or maintenance.

6.2.3 Profiling Cost

Gathering inter-query inputs (Section 5.2) incurs *profiling* costs which we now analyze. The profiling stage measures the cost of running each query in each execution backend. This

requires the user to pay to move and load tables to each new execution backend and to pay blob storage and read/write costs for the tables. This cost will in general be much larger than the cost of simply running the queries in one execution backend; however, as discussed in Section 5.2, a single profiling session will provide accurate data for many future executions, and we can cheapen profiling with sampling to reduce the cost burden of re-profiling as data changes. For example, Table 6.2 shows in column 100 the cost of this naive strategy and the number of iterations of the cheaper periodic workload needed to compensate for it.

We present the total profiling cost; if profiling occurs during repeated executions of periodic workloads, the *additional* costs users pay on top of their usual workload costs would be less than the naive cost presented in Table 6.2, reducing the net cost of profiling.

Additionally, we observe that costs may be reduced by profiling over a sample. Although estimating runtime based on a sample is difficult for some queries [51, 70, 91], most profiling cost comes from *pay-per-byte* pricing models, where runtime has no bearing in the final cost. We show the costs and estimation error for samples; estimation error is calculated as the difference between the target sampling size (i.e. 15%) and the observed sampling profiling cost divided by the naive profiling cost. The experiments show that even small samples estimate the necessary input parameters with low error for a fraction of the cost. We do not claim that sampling is the best approach, only that it sufficiently demonstrates how to cheapen profiling. More sophisticated approaches, such as those using parameterized cost models to more accurately sample for non-linear-complexity operators like joins [69], can further reduce error, which we leave for future work.

In most cases the cost is compensated after just 2 iterations. The Arachne plan achieves marginal savings for dataset 17, and achieves no savings for the GCP-only plans in datasets 7, 11, and 15. Many iterations on dataset 17 were needed to compensate profiling costs, but this number is significantly reduced by sampling.

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

Table 6.3: Absolute baseline, Arachne hybrid plan costs. Alt Syntax is DuckDB with Arachne-produced SQL

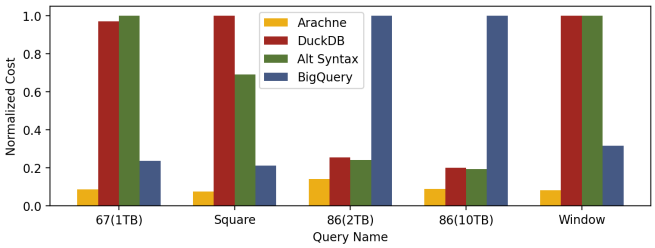


Figure 6.3: Query costs normalized to most expensive plan for Arachne's hybrid plans vs plans only in BigQuery, DuckDB, and DuckDB with Arachne-produced syntax.

6.3 RQ2. Hybrid Query Processing

In this section, we explore opportunity **O2** via the intra-query algorithm. The opportunities for intra-query savings are significant but occur less often than inter-query plans, 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 executed on a 1TB TPC-DS dataset and query 86 is evaluted on a 2TB and 10TB TPC-DS dataset. The "Window" query joins several tables, executes many group-by operations, and executes a complex window operation on the result. We also write the "Square" query, executed on the 100GB LDBC-SNB dataset, which searches for squares in social media graphs, e.g., finding a path from person A to B to C to D and back to A.

Experimental Setup. Data lives in GCP and we consider intra-query plans between BigQuery (pay-per-byte) and DuckDB (pay-per-compute) on a GCP VM to avoid migration costs. We use the same VM for DuckDB as in Section 6.2.2. Profiling costs are incurred by collecting baseline costs for the query in BigQuery and DuckDB, copying data to the VM, and measuring f_r as discussed in Section 4.

Results. Figure 6.3 shows the costs of the hybrid plan found by Arachne compared to the query executed in other backends, normalized to the most expensive baseline for each query. We isolate **O3** and present the cost of Arachne-produced SQL syntax as Alt Syntax.

Arachne finds cheaper intra-query plans: at least 2x and at most 5x compared to the next cheapest baseline and orders of magnitude compared to the most expensive baseline. This demonstrates the potential of hybrid plans to save costs. In this experiment, we report normalized values to emphasize the relative savings; the absolute savings in this case are small because out-of-memory errors on larger datasets prevented us from running these queries with longer runtimes. We show the absolute numbers in Table 6.3 and note that relative costs are a better indicator of savings for periodic workloads, as the total budget savings corresponds to the cost of a query multiplied by the number of times such a query executes.

Common Characteristics. Typical of analytical queries, these queries perform operations like graph traversal or window operation which join several tables (IO-intensive stage) followed by a window or self-join (CPU-intensive stage). Queries with this structure are good candidates for the intra-query algorithm.

Summary. Compared to **O1**, there are fewer situations where **O2** identifies a cost saving opportunity, but when opportunities exist, the relative savings are significant. The profiling costs to find these plans are earned back in under 25 iterations for 3 of the 5 queries. Query 67 and Window are earned back in 28 and 46 iterations and cost $85.68 and $40.18 respectively. The savings achieved are significant and compensate for incurred profiling costs.

6.4 RQ3. SQL Pre-Optimization

We explained in Section 5.3 that the SQL syntax affects query cost, so Arachne observes these effects and chooses the SQL syntax that yields the cheapest query execution. We measure that effect (**O3**) here. We compare the physical plans, performance, and cost in pay-per-compute systems of TPC-DS queries written in standard and Arachne-produced SQL. Both syntaxes are semantically equivalent and return the same results. We execute queries in Redshift and DuckDB. We found query plan differences due to syntax in 28/60 of

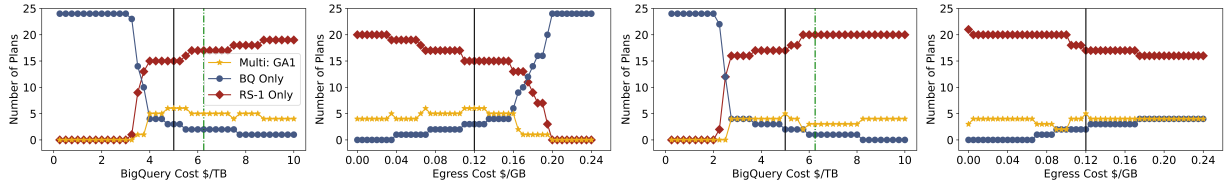
Figure 6.4: DuckDB and Redshift Percent Speedup of Machine-Generated SQL over Standard Syntax

the queries tested in DuckDB and 18/50 of those tested in Redshift.

Figure 6.4 shows the runtime speedup per query on the y-axis of the Arachne-produced query over the original syntax in either DuckDB or Redshift. Some instances have massive speedups–query 25 in Redshift experiences a 96% speedup and reduction in cost–and query 37 in DuckDB experienced 97% speedup. However, the impact is not always positive: query 95 in Redshift and query 47 in DuckDB experience 3 or 4× slowdowns or increases in cost.

To investigate this massive performance impact, we compare physical plans between standard and Arachne-produced syntax, which writes out all multi-input operators as nested subqueries. Arachne-produced syntax induces many changes, including: i) insertion of more projection operators, i.e., projection pushdown; ii) changes in join ordering; iii) predicate pushdown. These changes affect the size of intermediates and thus query performance.

To understand why syntax differences impacted query optimization, we studied the open-source optimizer for DuckDB. Different join orderings between physical plans account for all performance differences in DuckDB. DuckDB's optimizer estimates join node costs using the maximum cardinality of left and right child, so all orders will have the same join cost and the first one iterated over will be chosen, explaining why changes in syntax can yield a much more performant join ordering i.e. in query 37. The fix we proposed for this risks causing performance regression in other benchmarks, so it could not be merged [18], illustrating once



(a) 1TB BQ (b) 1TB Egress (c) 2TB BQ (d) 2TB Egress

Figure 6.5: (GA1 1–2TB) Simulated inter-query results varying either BigQuery (pay-per-byte) cost–labelled as BQ–or egress cost from source cloud–labelled as Egress.

again the complexity of modifying query optimizers. While we could not perform the same analysis on Redshift as it is closed source, we observe that the most significant impacts are correlated with different join orderings as these dramatically change intermediate size.

Query optimizer rules are sensitive to SQL syntax, so SQL syntax leads to different plans with different performance, impacting cost in pay-per-compute platforms, sometimes by several factors. Arachne utilizes its profiling stage to observe this effect and exploit them for cost-savings.

6.5 RQ4. What-If Analysis on Cloud Costs

So far, the results shown depend on the specific prices chosen by cloud vendors as of April'23. In this section, we include all data collected during our experiments into a simulator, where we vary the price-per-byte (BigQuery price) and egress price from GCP (our source execution backend), run the inter-query algorithm for the 48 datasets using GA1, and observe the impact on cost saving opportunities. We observe that trends are similar in GA4.

Figure 6.5 shows the breakdown of inter-query plans using the same categories as in Section 6.2–GCP, AWS, or Multi–as BigQuery and egress cost vary for 1TB and 2TB. The vertical line indicates the true price as of April'23. The dashed, green vertical line indicates BigQuery's new price announced for July'23 [10]. This illustrates how cloud prices are subject to change at any time. However, there will still be significant multi-cloud opportunities once BigQuery increases this price. The main take-aways are:

- There are multi-cloud saving opportunities in both cases as prices vary, so **O1** savings are robust to changes in prices.
- A reduction of 25% of BigQuery price to \$3.75/TB would keep most plans in BigQuery. At 2TB the necessary price reduction increases as the savings of Redshift over BigQuery increase as dataset size increases.
- Even small changes to egress prices induce changes to query placement. Increasing egress prices even by \$0.015/GB changes a Multi plan into a GCP plan.
- When egress prices are low, multi-cloud plans are still the cheapest option for 4/24 workloads at 1TB and 2TB. Even if cloud vendors lower financial barriers to data movement there are still inter-query opportunities (**O1**) to save money.
- High egress prices lock-in plans at 1TB, as migration costs outweigh savings, but do not at 2TB, meaning that the savings achievable by utilizing multiple pricing models grow faster than egress costs. Multi-cloud savings still exist if egress prices increase.

Overall, this result shows that the results we presented are not brittle to changes in price: some queries have a fundamental affinity to different pricing models. More importantly, they show the power platforms have to lock in workloads by adjusting prices lightly; this anti-competitive restriction should be concerning to all of us.

CHAPTER 7

RELATED WORK

Other Cloud Databases. Many other cloud and third-party databases such as Snowflake, Azure Synapse, Trino, Apache Hive, Amazon Athena, and SparkSQL [55, 34, 42, 28, 86, 1, 43] scan cloud storage or open formats like Delta Lakes [44]. These databases use per-second billing (Presto, Hive, SparkSQL), per-byte billing (Athena), or some combination of the two. While we could have used these other offerings in our evaluation, Google BigQuery and Amazon Redshift effectively represent both pricing models across clouds, enabling us to evaluate opportunities **O1**–**O4**.

Cloud Cost Savings. Early work on cloud money savings focuses on scheduling algorithms [89] or exploring cost sources in different execution backends [84]. More recent work has aimed to use and exploit the more cost-efficient serverless, or function-as-a-service, cloud offerings [79, 72, 60, 93] as well as pushdown computation to other cloud services to speed up queries and lower costs [92, 94, 90], for example using S3 Select [63]. Some prior work aims to recommend [47] or automatically choose cloud configurations [50] for users to help them parse through the large search space, but these methods use only a single pricing model in their cloud cost modelling. Leis and Kuschewski formally model per-second costs for cloud workloads [68]. To the best of our knowledge, Arachne is the first effort to systematically explore opportunities for saving costs for analytical queries by combining multiple execution backends with different pricing models.

Other Complementary Optimizations. The rich history of work in semantic caching and distributed optimization techniques [56, 58, 66, 77] can guide optimal data placements to minimize workload costs [65, 80]. Other prior work saves money through view selection and materialization in data warehouses such as Redshift or Amazon Athena [73, 52, 29, 32]. These efforts save costs within a single pricing model and can be applied to database configurations *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. Recently, Berkeley's Sky Computing group vision outlines opportunities for multi-cloud workload execution [49]. Our work on Arachne emphasizes cost savings across pricing models.

CHAPTER 8

CONCLUSION

This paper presents, exploits, and evaluates four money saving opportunities for cloud analytical workloads. The key is to exploit the different resources consumed by queries along with the different pricing models offered by cloud vendors. We use the periodic nature of our target workloads to measure hard-to-estimate query information. We implement algorithms which exploit inter-query and intra-query opportunities, observe and exploit the impact of SQL syntax on query cost, and exploit IaaS to save money.

We hope this work will encourage further investigation into multi-cloud saving opportunities. Ideally, this line of work fosters competition between cloud vendors, driving down prices and benefiting users. Cloud vendors may, however, simply modify prices or pricing models to prevent data movement, working to lock-in users. We show that even in extreme situations multi-cloud opportunities exist, and we hope that cloud vendors take the opportunity to reduce costs for users and to pay for the loss in revenue by becoming more energy-efficient to drive down their internal costs.

REFERENCES

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

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

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

[4] Batch data processing - Data Analytics Lens, . URL https://docs.aws.amazon.com/wellarchitected/latest/analytics-lens/batch-data-processing.html.

[5] Data Warehousing on AWS. .

[6] Batch processing - Azure Architecture Center. URL https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/batch-processing.

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

[8] Cost optimization best practices for BigQuery, . URL https://cloud.google.com/blog/products/data-analytics/cost-optimization-best-practices-for-bigquery.

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

[10] Introducing new BigQuery pricing editions, . URL https://cloud.google.com/blog/products/data-analytics/introducing-new-bigquery-pricing-editions.

[11] Use the BigQuery Storage Read API to read table data, . URL https://cloud.google.com/bigquery/docs/reference/storage.

[12] Pricing | BigQuery: Cloud Data Warehouse, . URL https://cloud.google.com/bigquery/pricing.

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

[14] Making dashboards faster, . URL https://www.metabase.com/learn/administration/making-dashboards-faster.

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

[16] Duckbill, . URL https://www.duckbillgroup.com/.

[17] Fanatics, . URL https://www.duckbillgroup.com/clients/fanatics/.

[18] Incorrect Join Ordering Due To Incorrect Join Node Cost Calculation · Issue #3525 · duckdb/duckdb, . URL https://github.com/duckdb/duckdb/issues/3525.

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

[20] Increase the Scale of BI with the Help of Hadoop OLAP. URL https://www.kyvosinsights.com/blog/how-olap-on-hadoop-helps-you-increase-scale-of-bi-phenomenally/.

[21] Apache Hive. URL https://hive.apache.org/.

[22] LDBC Social Network Benchmark (LDBC SNB). URL https://ldbcouncil.org/benchmarks/snb/.

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

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

[25] Introduction to BigQuery Omni. URL https://cloud.google.com/bigquery/docs/omni-introduction.

[26] Apache Parquet. URL https://parquet.apache.org/.

[27] Apache Pinot™: Realtime distributed OLAP datastore | Apache Pinot™. URL https://pinot.apache.org/.

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

[29] Performance - Amazon Redshift. URL https://docs.aws.amazon.com/redshift/latest/dg/c_challenges_achieving_high_performance_queries.html.

[30] snappy. URL http://google.github.io/snappy/.

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

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

[33] How Data Warehouses Can Save Your Company Money. URL https://www.sumoheavy.com/post/how-data-warehouses-can-save-your-company-money.

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

[35] Batch Processing - A Beginner's Guide. URL https://www.talend.com/resources/batch-processing/.

[36] Distributed SQL query engine for big data. URL https://trino.io/.

[37] OLAP 101: What it is and How to Apply it to Your Marketing, June 2020. URL https://smartboost.com/blog/how-to-use-online-analytical-processing-olap-in-marketing/.

[38] 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/. Section: Amazon Redshift.

[39] Running Batch Jobs in IBM Cloud Code Engine, April 2021. URL https://www.ibm.com/cloud/blog/running-batch-jobs-in-ibm-cloud-code-engine.

[40] Introducing AWS Glue Flex jobs: Cost savings on ETL workloads | AWS Big Data Blog, August 2022. URL https://aws.amazon.com/blogs/big-data/introducing-aws-glue-flex-jobs-cost-savings-on-etl-workloads/. Section: Analytics.

[41] 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/. Section: Amazon Redshift.

[42] 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, August 2020. ISSN 2150-8097. doi:10.14778/3415478.3415545. URL https://dl.acm.org/doi/10.14778/3415478.3415545.

[43] 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*, pages 1383–1394, Melbourne Victoria Australia, May 2015. ACM. ISBN 978-1-4503-2758-9. doi:10.1145/2723372.2742797. URL https://dl.acm.org/doi/10.1145/2723372.2742797.

[44] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, August 2020. ISSN 2150-8097. doi:10.14778/3415478.3415560. URL https://dl.acm.org/doi/10.14778/3415478.3415560.

[45] 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*, pages 2205–2217, Philadelphia PA USA, June 2022. ACM. ISBN 978-1-4503-9249-5. doi:10.1145/3514221.3526045. URL https://dl.acm.org/doi/10.1145/3514221.3526045.

[46] 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*, pages 221–230, Houston TX USA, May 2018. ACM. ISBN 978-1-4503-4703-7. doi:10.1145/3183713.3190662. URL https://dl.acm.org/doi/10.1145/3183713.3190662.

[47] Joyce Cahoon, Wenjing Wang, Yiwen Zhu, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Chengcheng Wan, Alexandra Ciortea, Sreraman Narasimhan, and Subru Krishnan. Doppler: automated SKU recommendation in migrating SQL workloads to the cloud. *Proceedings of the VLDB Endowment*, 15(12):3509–3521, August 2022. ISSN 2150-8097. doi:10.14778/3554821.3554840. URL https://dl.acm.org/doi/10.14778/3554821.3554840.

[48] Helena Caminal, Yannis Chronis, Tianshu Wu, Jignesh M. Patel, and José F. Martínez. Accelerating database analytic query workloads using an associative processor. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 623–637, New York New York, June 2022. ACM. ISBN 978-1-4503-8610-4. doi:10.1145/3470496.3527435. URL https://dl.acm.org/doi/10.1145/3470496.3527435.

[49] 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, May 2022. URL http://arxiv.org/abs/2205.07147. arXiv:2205.07147 [cs].

[50] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment*, 15(1):112–126, September 2021. ISSN 2150-8097. doi:10.14778/3485450.3485461. URL https://dl.acm.org/doi/10.14778/3485450.3485461.

[51] 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.

[52] 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.

[53] CloudZero. Meet Our Customers | CloudZero, . URL https://www.cloudzero.com/customers.

[54] CloudZero. 55 Cloud Computing Statistics That Will Blow Your Mind (2023), . URL https://www.cloudzero.com/blog/cloud-computing-statistics.

[55] 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*, pages 215–226, San Francisco California USA, June 2016. ACM. ISBN 978-1-4503-3531-7. doi:10.1145/2882903.2903741. URL https://dl.acm.org/doi/10.1145/2882903.2903741.

[56] Shaul Dar and Michael J Franklin. Semantic Data Caching and Replacement. page 12.

[57] 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.

[58] 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.

[59] 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.

[60] Samuel Ginzburg and Michael J. Freedman. Serverless Isn't Server-Less: Measuring and Exploiting Resource Variability on Cloud FaaS Platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 43–48, Delft Netherlands, December 2020. ACM. ISBN 978-1-4503-8204-5. doi:10.1145/3429880.3430099. URL https://dl.acm.org/doi/10.1145/3429880.3430099.

[61] 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.

[62] T. Heller, S. O. Krumke, and K.-H. Küfer. The Reward-Penalty-Selection Problem, June 2021. URL http://arxiv.org/abs/2106.14601. arXiv:2106.14601 [cs].

[63] 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/. Section: Amazon S3 Glacier.

[64] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, and Ravi Aringunram. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 International Conference on Management of Data*, pages 583–594, Houston TX USA, May 2018. ACM. ISBN 978-1-4503-4703-7. doi:10.1145/3183713.3190661. URL https://dl.acm.org/doi/10.1145/3183713.3190661.

[65] 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.

[66] 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.

[67] Kevin Kuchta. A Duck Tale, February 2021. URL https://www.duckbillgroup.com/blog/a-duck-tale/.

[68] 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.

[69] 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.

[70] 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*, pages 1129–1141, Virtual Event China, June 2021. ACM. ISBN 978-1-4503-8343-1. doi:10.1145/3448016.3457277. URL https://dl.acm.org/doi/10.1145/3448016.3457277.

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

[72] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 115–130, Portland OR USA, June 2020. ACM. ISBN 978-1-4503-6735-6. doi:10.1145/3318464.3389758. URL https://dl.acm.org/doi/10.1145/3318464.3389758.

[73] 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*, pages 28–34, McLean Virginia USA, November 2002. ACM. ISBN 978-1-58113-590-9. doi:10.1145/583890.583895. URL https://dl.acm.org/doi/10.1145/583890.583895.

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

[75] Aili McConnon Ohayon, Daniel. Enabling static analysis of SQL queries at Meta, November 2022. URL https://engineering.fb.com/2022/11/30/data-infrastructure/static-analysis-sql-queries/.

[76] Padmalathas. Get cost analysis and set budgets for Azure Batch - Azure Batch, December 2021. URL https://learn.microsoft.com/en-us/azure/batch/budget.

[77] 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/.

[78] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. How I Learned to Stop Worrying and Love Re-optimization, March 2019. URL http://arxiv.org/abs/1902.08291. arXiv:1902.08291 [cs].

[79] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD*

International Conference on Management of Data, pages 131–141, Portland OR USA, June 2020. ACM. ISBN 978-1-4503-6735-6. doi:10.1145/3318464.3380609. URL https://dl.acm.org/doi/10.1145/3318464.3380609.

[80] 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.

[81] PRASADA1207. Data lakes - Azure Architecture Center, December 2022. URL https://learn.microsoft.com/en-us/azure/architecture/data-guide/scenarios/data-lake.

[82] Mark Raasveldt and Hannes Mühleisen. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, Amsterdam Netherlands, June 2019. ACM. ISBN 978-1-4503-5643-5. doi:10.1145/3299869.3320212. URL https://dl.acm.org/doi/10.1145/3299869.3320212.

[83] 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/.

[84] 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. *Proceedings of the VLDB Endowment*, 12(12):2170–2182, August 2019. ISSN 2150-8097. doi:10.14778/3352063.3352133. URL https://dl.acm.org/doi/10.14778/3352063.3352133.

[85] 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.

[86] 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, 2010. IEEE. ISBN 978-1-4244-5445-7. doi:10.1109/ICDE.2010.5447738. URL http://ieeexplore.ieee.org/document/5447738/.

[87] 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, 2010. IEEE. ISBN 978-1-4244-5445-7. doi:10.1109/ICDE.2010.5447738. URL http://ieeexplore.ieee.org/document/5447738/.

[88] Alex Woodie. 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/.

[89] 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.

[90] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS. *Proceedings of the VLDB Endowment*, 14(11):2101–2113, July 2021. ISSN 2150-8097. doi:10.14778/3476249.3476265. URL https://dl.acm.org/doi/10.14778/3476249.3476265.

[91] 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. *Proceedings of the VLDB Endowment*, 13(3): 279–292, November 2019. ISSN 2150-8097. doi:10.14778/3368289.3368294. URL http://arxiv.org/abs/1905.04278. arXiv:1905.04278 [cs].

[92] 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. ISSN: 2375-026X.

[93] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/zhang-hong.

[94] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1345–1359, Philadelphia PA USA, June 2022. ACM. ISBN 978-1-4503-9249-5. doi:10.1145/3514221.3517856. URL https://dl.acm.org/doi/10.1145/3514221.3517856.