Managing Premium Data

Prasang Upadhyaya

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Magdalena Balazinska, Chair

Dan Suciu, Chair

Arvind Krishnamurthy

Program Authorized to Offer Degree:
University of Washington, Department of Computer Science and Engineering

University of Washington

Abstract

Managing Premium Data

Prasang Upadhyaya

Co-Chairs of the Supervisory Committee:
Associate Professor Magdalena Balazinska
Computer Science and Engineering

Professor Dan Suciu
Computer Science and Engineering

Data is transforming science, business, and governance by making decisions increasingly data-driven and by enabling data-driven applications. The data used in these contexts usually has significant economic or social value. Frequently, data is purchased from a provider where the price is linked to how the data will be used and the allowed usage is typically detailed in a license agreement. Data processing, too, is moving to public clouds where users must pay for access to cloud resources, which are frequently shared by multiple users, especially when users analyze a common dataset. Current solutions to manage the economic value of data (prices and licenses) rely on expensive support from economists, auditors and lawyers, thus, reducing the net value of data. Similarly, how to price *shared* cloud resources is poorly understood and when pricing ignores the shared nature of use, the cloud resources are significantly underutilized and users cannot realize the full value of their data.

In this thesis, we develop novel, principled and usable tools to manage data licenses and the pricing issues for data and cloud-based data processing.

We first present DataLawyer, a system to specify and enforce data use policies on relational databases. It includes an SQL-based formalism to precisely define policies, and novel algorithms, to automatically and efficiently evaluate the policies. Experiments on a

real dataset from the health-care domain demonstrate overhead reductions of up to 330× compared to a direct implementation of such a system on existing databases.

Next, we present a new approach for selecting and pricing shared optimizations on the cloud by using Mechanism Design. We develop new mechanisms, where users bid for optimizations, to select and price additive and substitutive optimizations, and for the general setting where the users and their bids can change over time. We show analytically that our mechanisms incentivize truthful bidding and ensure that the cloud never loses money. We show experimentally that our mechanisms yield higher utility than the state-of-the-art approach based on regret accumulation.

Lastly, we present improvements to data APIs. APIs are a common way to buy data. But users can significantly overpay when they makes multiple API calls and end up purchasing the same data item more than once. We provide a novel, lightweight and fast method to support pricing where a buyer is only charged once for each purchased tuple, even with multiple API calls. To enable this, we present a pricing framework where buyers can refund repeat purchases of data. We provide the protocols for refunds and develop optimizations to reduce the overhead of exercising refunds. Experiments show that data costs are significantly reduced (10× to 99×) for comparatively modest increases (2× to 5×) in query runtimes.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables iv

Chapter 1: Introduction 1
1.1 Motivation 2
1.2 Contributions 6

Chapter 2: Managing Data Use Agreements 10
2.1 Motivation 14
2.2 Policies 15
2.3 The DataLawyer System 19
2.4 Evaluation 32
2.5 Discussion 45
2.6 Conclusion 46

Chapter 3: Pricing Computation 47
3.1 Motivating Use-Case 51
3.2 A Mechanism Design Problem 52
3.3 A Mechanism for Static Collaborations 55
3.4 A Mechanism for Dynamic Collaborations 58
3.5 Mechanisms for Substitutable Optimizations 66
3.6 Evaluation 74
3.7 Conclusions 83

Chapter 4: Pricing Data 85
4.1 Problem Description 90

4.2 Naïve Approaches . . . 91
4.3 Refunds . . . 92
4.4 Extensions and Optimizations . . . 99
4.5 Evaluation . . . 106
4.6 Conclusion . . . 115

Chapter 5: Related Work . . . 116
5.1 Managing Data Use Agreements . . . 116
5.2 Pricing Shared Optimizations . . . 117
5.3 Optimal History-Aware Pricing with Data APIs . . . 119

Chapter 6: Conclusion . . . 120

Bibliography . . . 124

LIST OF FIGURES

Figure Number Page

2.1 Policy and query evaluation time for DataLawyer and NoOpt on policy P_6 and query W_1. . . . 34
2.2 Policy and query evaluation time (in ms) for DataLawyer and NoOpt for all policies. . . . 35
2.3 Overheads of log compaction. . . . 39
2.4 Policy and query evaluation time (in msecs) for each policy and query W_4. . . . 42
2.5 Advanced Interleaved Optimization. . . . 43
2.6 Policy Unification. . . . 44

3.1 Performance of Add$^{\text{On}}$ and Regret for the astronomy workload. . . . 77
3.2 Total utility as a function of optimization cost for different collaboration sizes. 78
3.3 Add$^{\text{On}}$ vs Regret performance with varying degree of collaboration. . . . 81
3.4 Add$^{\text{On}}$ improves while Regret worsens with temporal skew. . . . 82
3.5 Effect of change in selectivities of substitutable optimization on total utility. 84

4.1 An illustration of the limitations of `REST` data APIs for history-aware pricing. 86
4.2 Tree-structured group refunds. . . . 103
4.3 Amount paid for data with and without history-aware pricing. . . . 108
4.4 Time per query for three different pricing approaches: history-agnostic, history managed by the seller, and refunds-based. . . . 109
4.5 Overhead of coupon generation and refund requests. . . . 111
4.6 Time to evaluate the query and run the various pricing techniques for a workload of fast queries. . . . 113
4.7 Time to evaluate the query and run the various pricing techniques for a workload of slow queries. . . . 114

LIST OF TABLES

Table Number Page

2.1 Example policies from commercial data sellers and their informal classification. . . 11
2.2 The policies used in the experiments. 33
2.3 Queries used in experiments. Queries selected to cover a wide range of runtimes. 34
2.4 Policy and query evaluation time (in ms) for DataLawyer. 40
3.1 Symbol Table. For symbols with the argument time t, we drop t for offline mechanisms. 56

ACKNOWLEDGMENTS

To begin with, I am immensely indebted to my advisors, Magda Balazinska and Dan Suciu. I continue to be amazed by their exquisite taste in research, their brilliance, and their limitless enthusiasm; and I hope that I leave with some measure of those qualities. Throughout my stay at graduate school, they have exhorted me to take a long view of my work. I am very thankful for their patience in letting me make mistakes and arrive at my own conclusions, even if they, probably, anticipated much in advance what those conclusions might be.

I would like to thank Lindsay Michimoto, the first person I met at UW-CSE, and who was singularly responsible in ensuring that all my graduate school problems and moments of crises were strictly confined to the technical aspects of my research.

I thank YongChul Kwon, Paraschos Koutris, Bill Howe, and Martina Unutzer for the very rewarding collaborations I have had with them.

Figuring out the next step after graduate school took some deep thinking (and tons of interviewing). Apart from my advisors, who encouraged me the most to broaden my search, I owe my thanks to Raghav Kaushik, Christopher Re, Alexandra Meliou, Vibhor Rastogi, Chinmay Jain, and Ankit Singla for their advice and for sharing stories about their own careers and job search.

The UWDB group is one of the most friendly and smart group of people I have met: Alvin Cheung, Dan Halperin, Sudeepa Roy, Shumo Chu, Brandon Haynes, Jeremy Hyrkas, Dominik Moritz, Kristi Morton, Ryan Maas, Laurel Orr, Jennifer Ortiz, Jingjing Wang, Nodira Khoussainova, Emad Soroush, and Abhay Jha. They

have patiently listened to and helped improve many of my first practice talks, and then some more.

I will miss my generous Seattle friends who provided great company, amazing food, and fun conversations! I owe many thanks to Dvijotham Krishnamurthy, Ankit Gupta, Rahul Garg, Bharathwaj Palvannan, Shrey Khanna, Atul Kumar, Akshay Bhagwatwar, Vikash Kumar, Bijetri Bose, Shilpa Naidu, Shrainik Jain, Ravi Karkar, and Rishabh Shukla.

Graduate school would not have been possible without the love of my family: my sister Parul, and my parents, Amita and Pradeep. They have encouraged me to do what I feel passionate about while reminding me to enjoy life beyond work. Their sense of humor, wise advice, and unflinching support is what has allowed me to take those bets in my life that have paid off for me to arrive where I am today.

DEDICATION

To my parents.

Chapter 1

INTRODUCTION

Data has value. In industry, predictive models and data analytics are increasingly being used to improve products and services. For example, in finance, models serve to improve financial products through finer customer segmentation [53, 69]. In health care, data is enabling personalized and evidence-based treatment [68, 67, 88]. Data in education enables real-time feedback and personalized instruction [116]. Academic research in economics [34], astronomy [99, 66, 85], geology [55, 70, 112, 83] and health [72] too have become data driven. Beyond analytics, consumer facing applications are also becoming data rich. For example, public transit applications [44, 80] are commonly augmented with real-time traffic information. Social data such as user recommendations [121] and user highlights [58] is an important part of many mobile applications.

In many of the examples listed above, the data used for analysis or in data-rich applications is acquired from other companies [45, 31, 117, 41, 100, 44, 36] (Schomm et al. [97] survey data sellers and list 46 commercial data suppliers as of 2013). The reliance on external sources of data has led to the emergence of *data markets*, which comprise the technical, legal, and financial support needed for the exchange of data that has significant economic or social value.

Typically, when data is exchanged in data markets, it comes with a price and restrictions regarding how the data may be used. For example, Microsoft's Azure Marketplace [117] provides APIs to access over 100 paid data sources. For an instance of legal restrictions, consider Yelp [121]: Yelp requires that their ratings for businesses should not be combined with ratings from other providers to compute a meta-score, but Yelp allows their ratings to be displayed alongside the other ratings. Moreover, data analysis is increasingly being

conducted on public clouds where resources in the cloud come at a cost and are frequently shared by multiple users, especially when users analyze a common dataset. For example, telescope data from the Sloan Digital Sky Survey [99] is hosted on a public SQL Server, which is shared by different researchers; and Amazon Web Services hosts a large set of community databases for free [13], but it charges users for the Amazon EC2 compute instances needed to analyze that data.

Although the computational challenges of data collection, storage, dissemination and analysis is an active area of research, the emerging data markets calls for a wider perspective on how we work with data: we must also look at the economic and legal setting in which data is obtained, queried and analyzed. This leads to new challenges that require a fundamental change in our models, abstractions, and designs of data management technology for data in its broader context.

1.1 Motivation

We now explore, in more detail, the challenges of managing the licensing, the computation, and the data aspects.

1.1.1 Data Use Agreements

The use of data from data markets is, almost always, governed by data use agreements. These terms restrict, in various ways, how the data can be used. For example, Navteq [76], a map making company, prohibits users from joining their map data with other datasets: if the user wants to join, she needs to purchase the data at a higher price; Amazon Kindle [58] limits who can access a given book depending on the user context and whether the book has been lent to another user; and, Microsoft Translator [75] limits the amount of data that can be retrieved in a time window. These are much simplified descriptions. We surveyed 13 commercial and government data providers, and found that the average length of the terms of use document was 4489 words or about 8.3 pages; all were written in natural language, most often using legal terms, were often difficult to read, understand, and remember, and

were sometimes ambiguous.

It is impractical to expect that every user can understand and adhere to such agreements. While individual end-users happily ignore terms of use (commonly clicking the *I-agree* box without even opening the accompanying document), when a major firm acquires a valuable database, it cannot ignore its terms of use; they risk significant losses if their employees fail to adhere to the terms of use. This means that the company is responsible for, and interested in monitoring how its employees access, view, use, and manipulate data from its valuable databases. This often applies also to data produced internally by a company. For example, all large Internet companies put significant value on their user data (and their user privacy) and restrict both the employees who can access the data and what they can do with that data: *e.g.*, end-users may have agreed that their data be used to improve a certain product but for no other purpose. The company must respect this restriction.

To date, firms have few ways to enforce such terms of use and typically rely on access restrictions [4, 40] that obfuscate parts of the data, database triggers [82, 89], data auditing [57, 4, 52, 73, 35] and extensive employee training. Unfortunately, obfuscation severely reduces the utility of data and neither access control nor triggers can express many common requirements found in data use agreements. Techniques based on auditing can detect data misuses, but only after the fact.

Thus, the current status quo reduces the utility of data and relies on the users to comply with the license terms. Since these agreements are lengthy and complex, and may apply to sophisticated workloads over large datasets, non-expert users might inadvertently violate the terms or if they are overly cautious, they might not fully utilize the data in the mistaken belief that their proposed use can cause a violation. Thus, there is a need for systems that ease the understanding of license agreements and improves compliance.

1.1.2 Pricing Computation

Data analysis is increasingly being conducted on public clouds. There are many data-management-in-the-cloud options ranging from highly-scalable systems with simplified query

interfaces (*e.g.*, Windows Azure Storage [15], Amazon SimpleDB [12], Google App Engine Datastore [48]), to smaller-scale but fully relational systems (SQL Azure [71], Amazon RDS [9]), to data intensive scalable computing systems (Amazon Elastic MapReduce [8], to highly-scalable unstructured data stores (Amazon S3 [11]), and to systems that focus on small-scale data integration (Google Fusion Tables [46]).

Public clouds offer multiple options for users to trade-off price and performance such as views [3], indexes (*e.g.*, users can create indexes in SQL Azure and Amazon RDS), the choice of physical location of data (affecting latency and price as with Amazon S3), how data is partitioned (*e.g.*, Amazon SimpleDB data "domains"), and the degree of data replication (*e.g.*, Amazon S3 standard and reduced-redundancy storage). Cloud systems have an incentive to configure optimally, because this increases their customer's satisfaction and can also optimize the cloud's overall performance.

Providers of cloud based services know how to price the resources for individual use. But, resources in the cloud are frequently shared by multiple users, especially when users analyze a common dataset. For example, telescope data from the Sloan Digital Sky Survey [99] is hosted on a public SQL Server, which is shared by different researchers. Amazon's S3 storage service also allows users to share their data with others, with each user paying his or her own data access charges [10]. With shared use, optimizing the database for one user can benefit others too. How to price such shared resources is poorly understood and when pricing ignores the shared nature of use, the cloud resources are significantly underutilized and users can not realize the full value of their data.

Prior research [26, 56] on optimizing databases for shared use predict user demand for different configuration options. The cost of implemented the options is amortized to future queries that use them. These systems do not account for two important issues that we see in markets: (a) the cloud providers are not altruistic and will not implement an option without guarantees that its cost will be recouped, and (b) they assume that users in the cloud will act truthfully. In practice, users will try to game the system if doing so improves their utility. Any usable solution must ensure that clouds do not lose money. Moreover such solutions

must also adapt to the reality that users may start and terminate their use of the cloud arbitrarily and unpredictably over time.

1.1.3 Pricing Data

Many applications use data purchased online through REST APIs [45, 31, 100, 37, 117, 120, 92, 21] provided by data sellers. Existing APIs enable buyers to submit requests for data in the form of parameterized queries. For example, to purchase data from Twitter, one can specify keywords of interest, say a user name, in the API call and Twitter returns all activity, up to an API defined limit, from the user. Typically, sellers charge buyers based on how much data they purchase. In many scenarios, buyers need to make repeated calls to the seller's API and it is a challenge to keep track of the data purchased by applications to avoid charging applications twice in the case they request the same data twice.

The seller may put the burden on the buyer to never purchase the same data twice. Apart from increased storage and computation overhead for the seller, and the increased application complexity for the buyer this solution is unable to handle the following:

- *Updates*: For datasets such as weather and traffic, the underlying data is updated over time. In such cases, it may not be possible to predict when the updates are made to the subset of data that a buyer is interested in and the only way to know of an update is to redo the call to the data API.

- *Caching restrictions*: Some APIs such as Yelp [121] prohibit all forms of caching of the their data, while others, such as Twitter [107], prohibit caching of certain forms of data while permitting caching for the other parts of the data. Thus, even if the buyer knew that they would require a newly purchased data item in the future, they are prohibited from caching it and reusing it when the need arises.

Thus, in both circumstances above, the buyer can not avoid making multiple API calls and must incur the cost of repeat purchases of the same data.

An alternative is to store enough information about a buyer's purchase history so that the seller can figure out if a data item has already been paid for by the buyer. It may be beneficial for sellers to provide a service that only charges for data once so as to enable *price discrimination*. Although there are customers who may pay the full price of the data and not worry about paying extra, there are price-conscious customers who may not buy the dataset unless the data is available within their budget. Providing an avenue for such customers to optimize and reduce their data costs can increase revenues. But, storing such additional information to enable pricing that accounts for prior API calls can impose space and time overheads that are in the order of the data size and the number of previous API calls.

Thus, neither of the approaches that exclusively rely on buyer-side techniques or seller-side techniques provide a satisfactory solution that optimizes for both data costs and performance at the same time.

1.2 Contributions

We propose new models, abstractions and systems to automate the management of data use agreements that are associated with premium data. We also propose new mechanisms for sellers to price cloud computing resources shared across many users and propose new methods to optimize the purchase of data from data sellers.

Data Use Agreements In Chapter 2, we present DataLawyer, a system for the declarative specification and automated enforcement of data usage policies. We introduce a formal language, based on SQL, for specifying usage policies, which is rich enough to express all policies encountered in our survey (such as those in Table 2.1). The DataLawyer system, which is used as a middle-ware over a relational DBMS, allows users to run normal SQL queries, but before executing a query, it checks all policies. If any policy is violated, the query is rejected and the user is informed about the violation; otherwise, the query is evaluated in normal fashion.

Automated policy enforcement can be expensive because it requires recording a signif-

icant amount of information about each query: *e.g.*, the user identifier, the provenance expression, etc. To improve performance, we propose two novel algorithms: *log compaction*, addresses the problem of *high and growing* overhead of policy evaluation as more data is collected about a user's use of the data, by discarding information that is no longer necessary for future policy checking; and, *interleaved evaluation*, represents an advanced method to efficiently evaluate expensive policies and leverages the observation that, when a policy is satisfied, it is usually because there is a fragment of the policy that already proves it is satisfied.

We show experimental results from evaluating variants of policies in our survey on a real database from the health-care domain, MIMIC II [72]. Our results demonstrate that DataLawyer, with the optimizations, has a constant overhead that in many cases is to a few percent of the total query runtime, which is from 10× up to 330× less than an unoptimized implementation.

The work presented in this chapter originally appeared in the SIGMOD 2015 paper [110] entitled "Automatic enforcement of data use policies with datalawyer." Chapter 2 further adds the proof of NP-hardness of computing the absolute witnesses in §2.3 and an additional experiment for the advanced optimizations in §2.4.4.

Pricing Computation In Chapter 3, we present a new approach for selecting and pricing shared optimizations by using Mechanism Design.

We first show how the optimization pricing problem maps onto a cost-recovery mechanism design problem and how the Shapley Value Mechanism [74], which is known to be both cost-recovering and truthful, solves the problem of pricing a single optimization. We propose a direct extension of the mechanism to the case of additive optimizations in an *offline* scenario, where all users access the system for the same time-period.

Second, we present a novel mechanism for the online scenario, where users come and go. It turns out to be much more difficult to design mechanisms for the online setting:

algorithms that are truthful or cost recovering in the static setting cease to be so in the dynamic setting, see [79, pp 412]. We prove our new mechanism to be both cost-recovering and truthful in the dynamic setting.

Third, we extend the mechanisms for the offline and the online cases to the case where optimizations are inter-dependent, called substitutive optimizations, where the user derives a single value for any optimization in a set, however, implementing multiple optimizations from the set does not improve the user value. We prove the new mechanisms truthful and cost-recovering.

We experimentally compare our mechanisms against the state-of-the art approach based on regret accumulation [26] using a real dataset from the astronomy domain. We show that our mechanisms produce up to a 3× higher utility and provide the same utility for ranges of optimization costs up to 12.5× higher than the state-of-the-art approach in addition to handling selfish users and while ensuring that the cloud recovers all costs.

The work presented in this chapter originally appeared in the VLDB 2012 paper [109] entitled "How to price shared optimizations in the cloud." Chapter 3 further adds proofs related to substitutable mechanisms in §3.5.

Pricing Data APIs In Chapter 4, we propose lightweight modifications to data APIs to achieve *optimal history-aware pricing* so that buyers are only charged once for data that they have purchased and that has not been updated.

The key idea behind our approach is the notion of refunds: buyers buy data as needed but have the ability to ask for refunds of data that they had already purchased before. Thus, the payment for data is conducted in two steps: the usual payment when data is received and another round where the buyer asks for refunds. While asking for refunds, the buyer *proves* to the seller that she has been charged multiple times for the same data. The proofs are constructed so as to protect against tampering by the buyer even

when the buyer is not truthful or can collude with other buyers.

We also propose a generic and extensible framework to support refunds. We then show how it can be extended to accommodate multiple buyers, updates, and optimizations to reduce the computational and communication overheads of using refunds.

Refund-based pricing has other benefits too. First, it only requires constant overheads for the seller. Second, it also provide anonymity to the buyers about what data they purchase and when the purchases are made. That is, the seller need not retain any identifying information about the user that can recreate a user's query history.

We evaluate empirically and compare the refund based approaches to approaches that store user history at the server as well as approaches that do not provide optimal pricing. We show that significant cost savings of 10× to 99× can be obtained through the use of refunds while sustaining performance overheads that are no larger than 2× in the best case and 5× in the worst case.

In summary, we provide new models to represent license agreements in a form that enables a DBMS to automatically and efficiently verify that user queries comply with the agreements. We provide new mechanisms for pricing shared cloud resources. Lastly, we provide a light-weight framework that augments existing data APIs so that they can provide optimal pricing of workloads consisting of queries that may return the same data.

Chapter 2

MANAGING DATA USE AGREEMENTS

Whether sold online or offline, data typically comes with terms of use, which limit how the buyer can use the data. These are usually written by lawyers, span multiple pages, are difficult to read and are sometimes ambiguous. We performed an informal survey of 13 data providers[1] and found that terms of use accompanied all of the datasets or the APIs to acquire those datasets. Table 2.1 lists some examples of the terms we found. These terms restrict, in various ways, how the data can be used: for example P_1 says that Navteq prohibits users from joining their map data with other datasets: if the user wants to join, she needs to purchase the data at a higher price; P_2 limits which users can access a given data item depending on the context; P_3 limits the amount of data that can be retrieved in a time window, etc. These are much simplified descriptions. In our survey, the average length of the terms of use document was 4489 words or about 8.3 pages; all were written in natural language, most often using legal terms, were often difficult to read, understand, and remember, and were sometimes ambiguous.

While individual end-users happily ignore terms of use (commonly clicking the *I-agree* box without even opening the accompanying document), when a major firm acquires a valuable database, it cannot ignore its terms of use; they risk significant losses if their employees fail to adhere to the terms of use. This means that the company is responsible for, and interested in monitoring how its employees access, view, use, and manipulate data from its valuable databases. This often applies also to data produced internally by a company. For example, all large Internet companies put significant value on their user data (and their user privacy) and restrict both the employees who can access the data and what they can do with that

[1] Foursquare [41], Yelp [121], Azure Marketplace [117], Twitter [107], Infochimps [54], Socrata [100], Xignite [120], Digital Folio [33], DataSift [30], World Bank [118], Navteq [76], data.gov.uk [27], and DataMarket [29].

	Examples of terms of use	Restriction type
P_1	Overlaying Navteq data with any other data is prohibited (Navteq [76])	Prohibit joins
P_2	Each book may be lent once for 2 weeks while being inaccessible by the lender (Amazon Kindle [58])	Group licenses
P_3	All queries, totaled over a month, may return up to 2M chars at the free tier (MS Translator [75])	Generating free samples
P_4	OAuth calls are permitted 350 requests per hour (from Twitter [107] and a similar policy at Foursquare [41])	Rate limiting
P_5	Queries that try to identify an individual referenced in the database are prohibited (MIMIC II [72])	Limit information disclosure
P_6	You are required to display all attribution information and any proprietary notices associated with the Foursquare Data (Foursquare [41] and similar policies in Yelp [121] and World Bank [118])	Attribution and provenance
P_7	Don't aggregate or blend our star ratings and review counts with other providers. You may show content from multiple providers, but Yelp data should stand on its own (Yelp [121])	Disallow aggregations but allow joins and unions

Table 2.1: Example policies from commercial data sellers and their informal classification.

data: *e.g.*, end-users may have agreed that their data be used to improve a certain product but for no other purpose. The company must respect this restriction.

To date, firms have few ways to enforce terms of use and typically rely on access restrictions and extensive employee training.

In this chapter, we propose to specify the usage policies formally, and check them automatically at query time. We start by introducing a formal language, based on SQL, for specifying usage policies, which is rich enough to express all policies encountered in our survey (such as those in Table 2.1). Then, we present the DataLawyer system for enforcing data use policies automatically, inside a relational database management system (DBMS). DataLawyer is used as a middleware layer on top of a relational DBMS that allows users to run normal SQL queries, but before letting a query execute, it checks all policies. If any policy is violated, the query is rejected and the user is informed about the violation; otherwise, the query is evaluated in normal fashion.

The major challenge in any policy enforcement system is performance. Without the system, users would issue regular SQL queries. In the case of long-running analytical queries, the overhead of policy checking is easily amortized. In all other cases, however, policy checking can significantly slow down performance (Figure 2.1 in §2.4, for example, shows how a baseline non-optimized policy checking approach can impose a high and *growing* overhead).

Automatic policy enforcement is expensive because the system needs to record a significant amount of information about each query in a *usage log*: *e.g.*, the user identifier and query time, the query text, the query result, the provenance expression. In addition, it needs to execute every policy against the data and the usage log in order to check compliance. Done naïvely, this can become multiple orders of magnitude slower than running the SQL query alone. High overhead presents a major barrier for adoption: a company would not adopt a policy enforcement system, if it significantly slows down its daily operations.

To address the performance challenge, we propose novel algorithms to efficiently evaluate policies. These algorithms are based on query-rewriting techniques that leverage the separation of policies into data (the usage log component) and query (the declaratively specified policies). We develop two major optimizations.

The first optimization *log compaction*, addresses the problem of *high and growing* overhead of policy evaluation due to a growing usage log. This optimization removes from the log tuples that are no longer necessary for future policy checking. This optimization is based on the observation that, while in principle one could write policies that check the log arbitrarily far in the past, in practice policies tend to look for specific event patterns that are restricted to a limited subset of the log. For example, P_1 in Table 2.1 checks that Navteq data is not joined with other data sets. Since we know this policy was enforced for all past queries, we only need to keep the tail of the log generated by the current query. P_3 only requires the maintenance of the past month of data related to the free tier of service. Notice that log compaction is much harder than the tail-compaction used in recovery logs, which simply deletes the tail of the log after the last checkpoint: in our case we need to reason about the

semantics of the policies. We show that, for each policy specified in the system, one can compute an *absolute witness*, which is a subset of the log that is guaranteed to be sufficient to evaluate that policy, both now and in the future. The log compaction is obtained by replacing the log with the union of all absolute witnesses for all policies.

The second optimization, *interleaved evaluation*, represents an advanced method to efficiently evaluate expensive policies. We start from the observation that, by far, the most common case is when all policies are satisfied: this is when users issue queries that comply with the policies, and this is when they expect to see no significant slowdown over using the system without policy enforcement. Our second observation is that, when a policy is satisfied, it is usually because there is a fragment of the policy that already proves it is satisfied. In interleaved evaluation, we evaluate simplified versions of the original policies, and stop evaluation early, when we have determined that there are no violations.

Contributions In summary, this chapter makes the following contributions:

1. In §2.2, we propose a novel model to specify policies with the desiderata that any such model be sufficiently flexible to express a variety of policies, while being compact and precise. Our key idea is to first capture a relevant subset of user and database actions taken during query execution; and then to specify the policies *declaratively* over those logs as states of the log that are inconsistent with the intent of the policies. We describe the semantics of our data model and show how the common policies found in real world terms of use can be concisely and precisely expressed in our data model.

2. In §2.3, we present our optimized policy evaluation methods including log compaction and interleaved evaluation.

3. Lastly, in §2.4, we show experimental results from evaluating variants of policies defined in Table 2.1 on the MIMIC II [72] database. Our results demonstrate the practicality of our approach and the importance of our optimizations. While it is possible with DataLawyer to write policies that perform expensive checks, DataLawyer's optimiza-

tions enable the system to keep the overhead constant and, in many cases, cut that overhead to a few percent of the total query runtime, which is from 10× up to 330× less than an unoptimized implementation.

The source code [28] for DataLawyer's implementation for PostgreSQL is available publicly.

2.1 Motivation

DataLawyer's goal is to enable data sellers to specify precise data use policies and to help data buyers to use the data without violating any of the policies.

DataLawyer should not be confused with an access control system [40]; such systems restrict users to a *fixed* set of authorization privileges (or access modes), which are strictly limited to reading or writing columns or rows. In contrast, terms of use refer to complex scenarios, *e.g.*, in Table 2.1, P_1 specifies that the data may not be joined with any external datasets, P_4 limits the rate of queries, P_7 disallows aggregation; none of these are captured by access control systems.

DataLawyer is also not intended to protect against a malicious user; with some patience and effort a malicious user can extract the entire data without breaching any policy, then store it on her own device and use at will. The system is designed to help honest users comply with terms of use that are difficult to read, understand, and remember. It is also meant to help large corporations monitor how their valuable data is used internally by their employees.

Finally, we note another potential application of DataLawyer: usage-based data pricing. Wang et al. [114] postulate that the value of data is both intrinsic (*e.g.*, based on its completeness and accuracy) and extrinsic (*i.e.*, it depends on the context in which it is used). Data owners frequently control the extrinsic value of data by limiting the kind of operations allowed on it. For example, Factual [36], an online data vendor, prices its data based both on volume and on what the buyer uses the data for: data used in ads is priced differently

from data used in applications and prices can also vary between applications. DataLawyer can be used to compute the price of the data dynamically, *e.g.*, based on how the data was used during the last billing period.

2.2 Policies

In this section, we present the policy specification formalism, the usage log, and their semantics.

2.2.1 Policy Specification

For each term of use, the data owner defines a policy π, which is a SQL query of the following form:

```
SELECT DISTINCT [error-message] FROM ...
WHERE ... GROUP BY ... HAVING ...
```

The `FROM` clause contains base tables, or `select-from-where-groupby-having` subqueries. The `WHERE` and `HAVING` clauses are conjunctions of atomic predicates without subqueries.

The SQL query may refer both to the database and to a *usage log* that we describe in §2.2.2. If the policy π returns the empty set, denoted $\pi = \emptyset$, then the policy is satisfied. Otherwise, we say that π returns *true*, denoted $\pi \neq \emptyset$; in that case a violation has been detected and the error-message specified in the policy is returned to the user.

We illustrate our policy language with two examples.

Example 2.2.1. *P_{5b} is a concrete variant of P_5 in Table 2.1:*

P_{5b}: Stop queries where fewer than 10 `patient`*s contribute to any output tuple.*

Policy P_5 is from the MIMIC II (Multiparameter Intelligent Monitoring in Intensive Care) [72] database that contains readings from patient monitoring systems and clinical data collected at an ICU over seven years. In our system, the policy is specified as follows:

```
SELECT DISTINCT 'P5b violated: Fewer than 10
  patients contribute to an answer' AS errorMessage
FROM Provenance p
WHERE p.irid = 'patients'
GROUP BY p.ts, p.otid
HAVING COUNT(distinct p.itid) < 10
```

The table `Provenance(ts, otid, irid, itid)` *in the* `FROM` *clause is part of the usage log (described in detail below). A record in* `Provenance` *represents the fact that input tuple* `itid` *from input relation* `irid` *belongs to the provenance of the output tuple* `otid` *of the query executed at time* `ts`*. To simplify the presentation, we assume that timestamps are taken from an integer clock with sufficient granularity that each query has a unique* `ts` *attribute.*[2] *In the examples, we further assume that timestamps are expressed in seconds. Thus, the policy simply checks if there exists some query that has an output tuple whose provenance has fewer than 10 distinct input tuples from the* `patients` *table, which is one of the tables in the MIMIC II database*[3]*. If the answer is non-empty, then the policy has been violated. The policy appears to check* all *queries, but our system only evaluates it on the* last *query, namely the query currently requested by the user (because all previous queries have already been checked and are known to satisfy all policies); we explain this and other optimizations in §2.3.*

Example 2.2.2. *We now illustrate a more complex policy, involving temporal restrictions:*

> *P_{2b}: At most 10 distinct users from the group* `'Students'` *are allowed to query* `patient`*s in any window of 14 days.*

This policy generalizes P_2 in Table 2.1, and is adapted to the same MIMIC database, for illustration purposes. The policy is expressed as follows in our language:

---

[2] We could relax this assumption by adding a separate, unique `query id` attribute but that complicates examples.

[3] Its schema is `patients(pid, dob, sex)`.

```
SELECT DISTINCT 'P2b violated: More than 10 users
   executed queries in 14 days.' AS errorMessage
FROM Users u, Schema s, Groups g, Clock c
WHERE u.ts = s.ts and s.irid = 'patients'
  and u.uid = g.uid AND g.gid = 'Students'
  and u.ts > c.ts - 1209600
HAVING COUNT(distinct u.uid) > 10
```

Notice that the query has no ***GROUP BY*** *clause: it checks if the total number of distinct user id's that referred to* ***patients*** *over the last 14 days is greater than 10. This policy refers to two other tables in the log:* ***Users****, which records the user id of the user issuing the query, and* ***Schema****, which records the schema information for each query; both are described below. The* `Clock` *relation has a single row and a single column updated by the system*[4].

2.2.2 Usage Log

We now describe the *usage log*, denoted by $\mathbf{L}$, that captures all features of queries executed on the database that are necessary to enforce a set of data use policies. The log consists of m relations, $\mathbf{L} = (R_1, \ldots, R_m)$, where each relation R_i captures features of a particular type. Any feature can be stored in the log; the only requirement is that each relation has a timestamp attribute, $\mathtt{R}_i\mathtt{(ts,...)}$. In addition, the system defines m *log-generating functions* (referred to as functions for brevity), $\mathbf{f} = (f_1, \ldots, f_m)$. When a query q is executed on the database instance D, if it satisfies all policies then, for each relation R_i, the system uses the function f_i to compute the set of features $S_i = f_i(q, D)$ to be appended to R_i: it then updates $R_i = R_i \cup (\{t\} \times S_i)$, where t is the current timestamp.

In addition to the usage log, the system also exposes the `Clock` relation that has a single row with a single attribute corresponding to the current time (see Example 2.2.2).

Example 2.2.3. *The usage log in our DataLawyer system prototype consists of the following three relations:*

[4] Alternatively, we could have used a function like `CURRENT_TIMESTAMP()`, with the same semantics as `c.ts`.

```
Schema(ts, ocid, irid, icid, agg)
Users(ts, uid)
Provenance(ts, otid, irid, itid)
```

`Schema` records the schema information of each query: a record `(ts, ocid, irid, icid, agg)` represents the fact that the answer to the query executed at time `ts` contains a column `ocid`, which stores a value derived from the input column `icid` from the input relation `irid`; `agg` indicates whether an aggregate was used. For example, the query, `SELECT T.A AS K, (T.B + T.C) AS L FROM T`, generates three rows in `Schema`:

```
(ts, K, T, A, false)
(ts, L, T, B, false)
(ts, L, T, C, false)
```

The log-generating function $f_{Schema}(q, D)$ takes as input a SQL query q and computes all tuples that need to be inserted in `Schema` on behalf of q, by performing static analysis on q; this function does not need the database instance D.

The `Provenance` relation contains complete provenance information. We use the set of contributing tuples provenance*, also called* lineage *[103], where, for each output tuple `otid` we record all contributing input tuples `irid`, `itid`. The function $f_{Provenance}(q, D)$ computes the provenance of q on D by running a `SELECT * ...` query derived from q, similarly to Perm [43]. Finally, `Users` stores, for each query, the id of the user who executed that query.*

In the rest of the chapter, we assume that the schema of the usage log is that given in Example 2.2.3. We emphasize, however, that all our optimization techniques apply to an arbitrary schema: to add a new relation R_i to the log, the systems administrator only has to write the corresponding log-generating function $f_i(q, D)$.

2.2.3 Semantics

We can now define the semantics of our policy manager. We denote by $\mathbf{L}_t$ the log up to timestamp t, and denote $\mathbf{\Pi} = \{\pi_1, \ldots, \pi_p\}$ the set of policies defined in the system.

Informally, when a user asks a query q over the database D at time t, the system first appends $\{t\} \times f_i(q, D)$ to each log relation R_i, to produce a tentative log $\mathbf{L}'_t$. Then, it checks all policies on the new log; if all return $\emptyset$, in notation $\mathbf{\Pi}(t, \mathbf{L}'_t, D) = \emptyset$, then the query is executed and the answer is returned to the user; otherwise the query is rejected and the log is reverted to $\mathbf{L}_{t-1}$. Formally:

$$\begin{aligned} \mathbf{L}_0 &= \emptyset \\ \mathbf{L}'_t &= \mathbf{L}_{t-1} \cup (\{t\} \times \mathbf{f}(q, D)) \\ \mathbf{L}_t &= \begin{cases} \mathbf{L}'_t & \text{if } \mathbf{\Pi}(t, \mathbf{L}'_t, D)) = \emptyset \\ \mathbf{L}_{t-1} & \text{otherwise} \end{cases} \end{aligned} \tag{2.1}$$

Here $\mathbf{f}(q, D)$ denotes all functions $f_1, \ldots, f_m$. Each policy $\pi(t, \mathbf{L}_t, D)$ may refer to the log, the database, and also the timestamp t, obtained from `Clock` (see Example 2.2.2).

2.3 The DataLawyer System

When the user issues a new query, a naïve way to check the policies is to apply directly the semantics of Eq.(2.1): generate the increment of the log $\mathbf{L}$, write it to disk, then iterate over the policies and evaluate them. If any violation is found, revert the log. We do not consider this naïve strategy. Instead, our NoOpt Algorithm 1 incorporates the following straightforward optimizations:

1. Generate only those logs R_i mentioned in the policy definitions. For example, if no policy uses `Provenance`, then do not generate that log at all.

2. Store the generated increments to the logs $\mathbf{f}(q, D)$ in temporary tables in memory and keep them there while checking the policies. Write them to disk only when all policies are satisfied. Without this optimization, in the case of failure, the log increments have to be deleted from disk.

Even with these two optimizations, checking policies with the NoOpt algorithm is impractical because (as we show) it can be 1-2 orders of magnitude slower than the query that

the user wants to run. Additionally, with NoOpt, policy checking times increase as more queries execute against the database. In this section, we first present optimizations that (1) reduce the size of the log to keep policy-checking times constant (and low), and (2) use an optimized evaluation strategy for the set of policies to reduce policy-checking times compared to NoOpt. We then put these optimizations together and show how to apply them for a *set* of policies defined on a dataset.

Algorithm 1: The NoOpt Algorithm

Input : Timestamp t, query q, database D.

Output: Update the log or abort.

begin

    BEGIN TRANSACTION

    $\mathbf{L} \leftarrow \mathbf{L} \cup (\{t\} \times \mathbf{f}(q, D))$

1   $\pi_{union} \leftarrow \pi_1 \cup \cdots \cup \pi_k$

2   **if** $\pi_{union}(t, \mathbf{L}, D) = \emptyset$ **then** COMMIT;

3   **else** ABORT;

2.3.1 Data Minimization

In theory, the log can grow forever, and the policies may inspect the entire log, from the beginning of time. In practice this never happens. We describe optimization techniques that exploit the fact that policies look only for restricted events back in time.

Time-Independent Policies

We start with a very simple optimization, which identifies when a policy depends only on the current query, and not on the log history. For example, policy P_1 in Table 2.1 prohibits joins of a dataset with other datasets: it only depends on the current query q and not the log history. In other words, we do not need to examine the entire log and check what previous queries have done.

We call a policy *time-independent* if it can be checked by examining only the increment of the log instead of the entire log: formally, denoting $\mathbf{L}_{past} = \mathbf{L}_{t-1}$ and $\mathbf{L}_{present} = \mathbf{L}_t - \mathbf{L}_{t-1}$,

then $\pi(\mathbf{L}_t, D) = \pi(\mathbf{L}_{past}, D) \cup \pi(\mathbf{L}_{present}, D)$. We give here a syntactic criterion for time-independence. One subtlety is that a time-independent policy is usually not written in a way in which it refers only to the current time, but checks a property for all timestamps: we need to use the fact that the policy was true in the past to infer that it suffices to check only the current time. Formally, a SQL policy π is time-independent if it, and all its subqueries in the `FROM` clause, satisfy the following conditions: (a) all timestamp attributes from all relations are joined, and (b) if π contains any aggregates then the group-by attributes include the timestamp. If π is time-independent, we rewrite it to a policy denoted by π_{ind} by adding a selection requiring that the timestamps, `ts`, refer to the current clock, `c.ts`.

Example 2.3.1. *Policy P_1 and its optimization are:*

```
P1: SELECT DISTINCT 'No external joins allowed'
    FROM Schema p1, Schema p2
    WHERE p1.ts=p2.ts and p1.irid='Navteq' and p2.irid != 'Navteq'

P1_IND: SELECT DISTINCT 'No external joins allowed'
        FROM Schema p1, Schema p2, Clock c
        WHERE p1.ts = c.ts and p2.ts = c.ts and p1.ts = p2.ts
         and p1.irid = 'Navteq' and p2.irid != 'Navteq'
```

Thus, we restrict the policy to check only the current timestamp. This is correct only *because we know that the policy was satisfied in the past. In addition to restricting policy evaluation to the log increment, we can further optimize the system by not appending to the log at all: this is handled by the log-compaction optimization, discussed next.*

Log Compaction

The log compaction optimization removes from the log those entries that are guaranteed to be unnecessary now, and at any time in the future[5]. Notice that this is far more complex that the tail compaction in recovery logs, which deletes the entire log preceding the last successful checkpoint. In our case, the compaction must take into account the semantics of

[5]If a new policy is added at time t, DataLawyer restricts its history to start at time t by adding extra predicates on the timestamp attributes.

the policies and reason about which tuples will never be used in the future.

Example 2.3.2. *Consider policy P_{2b} from Example 2.2.2: no more than 10 distinct users from group* `'Student'` *may execute a query on* `patients` *in any window of 14 days. Assume for a moment that this is the only policy in our system. Then, it is obvious that we can (a) store in the log only entries belonging to users in the group* `'Student'` *and referring to table* `patients`*, and (b) remove all entries older than 14 days. By repeating this kind of reasoning to all policies we can compute a subset of the log that is sufficient to check all policies in the future.*

We give now the formal definition. Fan [38] defines a witness for a query Q to be a subset of the database s.t. Q returns the same answer on the witness as on the entire database. Adapting to our setting, let π be any query (Boolean or not), and $\mathbf{L}_t, D$ be the current log and the current database. A *witness* for π is a subset $\mathbf{L}_t^w \subseteq \mathbf{L}_t$ such that $\pi(t, \mathbf{L}_t, D) = \pi(t, \mathbf{L}_t^w, D)$. We call the tuples $T = \mathbf{L}_t - \mathbf{L}_t^w$ *dispensable tuples* (for the given witness). We could remove the dispensable tuples from the log without affecting the policy at the current time. They may, however, become necessary in the future. If $t \leq t'$ then we write $\mathbf{L}_t \leq \mathbf{L}_{t'}$ to denote the fact that $\mathbf{L}_{t'}$ is an extension of the log from time t to time t'.

Definition 2.3.1. *Let π be a query (Boolean or not). A set of tuples $T \subseteq \mathbf{L}_t$ is called* absolutely dispensable *for π, if for any future evolution of the log $L_{t'} \geq L_t$, $\pi(t', \mathbf{L}_{t'}, D) = \pi(t', \mathbf{L}_{t'} - T, D)$. We call $\mathbf{L}_t^w = \mathbf{L}_t - T$ an* absolute witness.

For simplicity, we will refer to absolute witness as witness. The Log Compaction Algorithm 2 examines each policy π in turn, and computes a witness $\mathbf{R}_{i,\pi}^w$ required by that policy, for each log relation $\mathtt{R}_i$. Then, it takes the union of all witnesses required by all policies, as well as by their subqueries occurring in the `FROM` clause. The heart of the algorithm consist of computing $\mathbf{R}_{i,\pi}^w$, the witness for $\mathtt{R}_i$ required by the policy (or subquery) π. Computing a minimal witness is NP-hard in general since computing a witness [38] is NP-hard and we can reduce the problem of computing the witness to a query π on a database D to computing the absolute witness of a query π' over a database D'. D' has an extra timestamp

Algorithm 2: Log Compaction Algorithm: `compact`

Input : A set of policies $\mathbf{\Pi}$

Output: Witness tuples for each log relations $\mathtt{R}_i \in \mathbf{L}_t$

begin

1 **foreach** R_i **do** $\mathbf{R}_i^w \leftarrow \emptyset$

2 **foreach** $\pi \in \mathbf{\Pi}$ **do**

foreach *subquery* Q_i *in* π*'s* `FROM`*-clause* **do**

3 $(\mathbf{R}_1^w, \ldots, \mathbf{R}_m^w) \leftarrow (\mathbf{R}_1^w, \ldots, \mathbf{R}_m^w) \cup \mathtt{compact}(\{Q_i\})$

foreach *log relation* R_i *in* π*'s* `FROM`*-clause* **do**

4 $\mathbf{R}_i^w \leftarrow \mathbf{R}_i^w \cup \mathbf{R}_{i,\pi}^w$ // See text.

return $(\mathbf{R}_1^w, \ldots, \mathbf{R}_m^w)$

attribute, set to 0 for every tuple, for each relation; while π' is equal to π but with a predicate that only selects tuples with a timestamp equal to 0. Thus, any new tuples added to D' (with a larger timestamp) would be irrelevant for computing absolute witnesses and hence, the absolute witness would be identical to the witness at time timestamp 0. Thus, we settle for heuristics, based on the structure of the policy π. If the policy has subqueries in the `FROM` clause, when we handle them separately. For example, to compute the witness for `SELECT ...FROM Subquery, R1, R2 WHERE ...HAVING ...` we compute separately the witness for `Subquery` and for the modified query $\pi =$ `SELECT * FROM R1, R2 WHERE ...`, then union the witnesses. In the remainder of this section we assume that π is a policy without subqueries, and show how to compute a witness $\mathtt{R}_{i,\pi}^w$.

Note that setting $\mathtt{R}_{i,\pi}^w = \mathtt{R}_i$ always gives us a correct witness, which is equivalent to not doing any log compaction. We now describe how to compute a smaller witness for certain policies π, starting with simple ones and successively generalizing to more complex ones.

No Clock, Full Query Consider a query π that does not refer to the clock (current time), and also has no projections:

$$\pi = \texttt{SELECT * FROM } \mathtt{R}_1, \ldots, \mathtt{R}_m, \mathtt{D}_1, \ldots, \mathtt{D}_q \texttt{ WHERE} \ldots$$

Here, $\mathtt{R}_i$, $i = 1, m$ are part of the usage log, and $\mathtt{D}_j$, $j = 1, q$ are part of the database, e.g. `Groups` in Example 2.2.2. Note that, while all policies in our system are Boolean queries, Def. 2.3.1 also applies to full queries and we start here by describing how to derive the witness $\mathbf{R}^w_{i,\pi}$ for a full query π. Let $\mathtt{R}_i$'s *neighborhood* $N(\mathrm{R}_i) = \{\mathtt{R}_{i_1}, \ldots, \mathtt{R}_{i_v}\}$ be the set of all other log relations that equijoin on the timestamp with $\mathtt{R}_i$, directly or indirectly. Note that this set may be empty.

Lemma 2.3.1. *The following queries define a witness for the policy π and relations R_i:*

$$R^w_{i,\pi} = \texttt{SELECT DISTINCT } R_i.* \quad \texttt{FROM } R_i, R_{i_1}, \ldots, R_{i_v}, D_1, \ldots, D_q \texttt{ WHERE } \ldots \tag{2.2}$$

The `FROM` *clause contains R_i, its neighborhood, and the database relations, and the* `WHERE` *clause contains all conditions in π that refer only to the relations in the* `FROM` *clause. If the same relation name R_i occurs multiple times in π (self-joins), then the witness $R^w_{i,\pi}$ is obtained as the union of the queries (2.2), one for each occurrence of R_i in the* `FROM` *clause (see Example 2.3.4 below).*

(Proof sketch) We outline the proof that $\mathtt{R}^w_{i,\pi}$, in Eq. 2.2, correctly computes an absolute witness of relation $\mathtt{R}_i$ for policy π. We consider the possible cases. First, the case when $\mathtt{R}_i$'s neighborhood is empty: then the query simply selects those tuples in $\mathtt{R}_i$ that satisfy all predicates on $\mathtt{R}_i$ in the policy: obviously, all tuples that do not satisfy these predicates are dispensable for evaluating the policy, both now and in the future. Second, if the neighborhood is non-empty, then $\mathtt{R}_i$ is semi-joined with the other $\mathtt{R}_{i_j}$'s (this is a semi-join reduction [1]): all other tuples are dispensable now, and are also dispensable in the future because all the $\mathtt{R}_{i_j}$'s are joined on the timestamp, and no new tuples are being added at a current, or past timestamp.

No Clock, Boolean queries We now generalize the algorithm to Boolean queries π, still without reference to the `Clock`. If π has a `HAVING` clause, then we drop the `GROUP-BY` and `HAVING` clauses, replace `SELECT` with `SELECT *` to treat the policy as a full query, and compute an absolute witness for the full query using Eq.(2.2); in other words, we do not take advantage of the fact that π is Boolean. Otherwise:

$$\pi = \texttt{SELECT DISTINCT 'Error' FROM } \mathtt{R}_1, \ldots, \mathtt{R}_m, \mathtt{D}_1, \ldots, \mathtt{D}_q \texttt{ WHERE } \ldots$$

In this case we can compute a smaller witness than that for the full query. Let $N(\mathtt{R}_i) = \{\mathtt{R}_{i_1}, \ldots\}$, be the neighborhood of $\mathtt{R}_i$, and denote X the set of all attributes of $\mathtt{R}_i$ occurring in a join predicate.

Lemma 2.3.2. *The following query (obtained by modifying Eq.(2.2)) computes an absolute witness to π:*

$$\begin{aligned} R^w_{i,\pi} = \; & \textit{SELECT DISTINCT ON } (R_i.X), R_i.* \\ & \textit{FROM } R_i, R_{i_1}, \ldots, R_{i_v}, D_1, ..., D_q \textit{ WHERE } \ldots \end{aligned} \tag{2.3}$$

(Proof sketch) Recall that the `DISTINCT ON` statement in SQL nondeterministically chooses a single witness from an entire group of tuples. For example, `SELECT DISTINCT ON (R.A), R.B FROM R` chooses nondeterministically a value `R.B` for each distinct value `R.A`. In other words, the witness is computed by nondeterministically choosing any tuple that contributes to the output. Notice that the absolute witness is not unique: for each distinct value of `X`, the algorithm can choose any tuple with those values of the attributes `X`.

Adding the Clock Finally, we consider queries (Boolean or not) referring to `Clock`; recall that this is done through an expression `Clock c` in the `FROM` clause. We assume all predicates on the clock are of the form `c.ts op expression`, where `op` is one of $<, \leq, >, \geq, =$, in other words `op` cannot be $\neq$; we apply a set of simple transformation rules to rewrite expressions

like $\texttt{u.ts} > \texttt{c.ts} - 5$ into $\texttt{c.ts} < \texttt{u.ts} + 5$. We don't perform log compaction on policies that have an inequality operator, $\neq$, on the clock. Furthermore, we replace every equality predicate $\texttt{c.ts} = \texttt{expr}$ with $\texttt{c.ts} \leq \texttt{expr and c.ts} \geq \texttt{expr}$.

Lemma 2.3.3. *Let π be a policy where all predicates on the clock are of the form* `c.ts op expr` *where* `op` $\in \{<, \leq, >, \geq\}$. *Then, an absolute witness can be computed by the query (Eq. 2.2) or (Eq. 2.3) (depending on whether π is Boolean or not), with the following modifications: (a) Drop predicates of the form* `c.ts` $>$ `expression`, *(b) Replace every predicates of the form* `c.ts` $<$ `expression` *(or $\leq$), with* `currenttime` $+ 1 <$ `expression` *(or $\leq$), where* `currenttime` *is a constant that represents the current value of the clock.*

Notice that a time-independent policy π_{ind} (Sect. 2.3.1) will return an empty witness, in other words it does not contribute anything to the log. Indeed, such a policy contains the predicate $\texttt{c.ts} = \texttt{R}_i\texttt{.ts}$, which is rewritten to $\texttt{currenttime} + 1 \leq \texttt{R}_i\texttt{.ts}$, which evaluates to false because all new tuples in $\texttt{R}_i$ have the current time-stamp.

(Proof sketch) Without dropping or modifying the predicates referring to the clock, the expressions (Eq. 2.2) or (Eq. 2.3) will compute a witness, but not necessarily an absolute witness; we note that, for (Eq. 2.3), all attributes occurring in `expression` of $\texttt{c.ts} <$ `expression` are included in the `DISTINCT ON` attributes $\texttt{R}_i\texttt{.X}$ (they are considered as occurring in a join). By dropping the predicate $\texttt{c.ts} >$ `expression` we increase the set of witnesses, and ensure that we also include all witnesses in the future, when `c.ts` will be larger. Similarly, by modifying $\texttt{c.ts} <$ `expression` to $\texttt{currenttime} + 1 <$ `expression` we drop all tuples that will be dispensable starting with the next time stamp.

We illustrate log compaction with two examples.

Example 2.3.3. *Continuing Example 2.3.2, we show how DataLawyer computes the absolute witness for the query* `P2b` *in Example 2.2.2. First, transform the policy into a full query. Since the join is on the timestamp, the neighborhood of* `Users` *and of* `Schema` *includes the other: $N($*`Users`*$) = \{$*`Schema`*$\}$ and $N($*`Schema`*$) = \{$*`Users`*$\}$. Moreover, we update the predicate on time. Therefore, our system computes the absolute witness for* `Users` *as:*

```
SELECT DISTINCT u.*
FROM Users u, Schema s, Groups g
WHERE u.ts = s.ts and s.irid = 'patients' and u.uid = g.uid
  and g.gid = 'Student' and u.ts > currentTime + 1 - 1209600
```

In other words, we only record users from **'Student'** *and only if they have issued a query on* **Patients** *in the last 14 days, less one time unit. Note that other policies compute their own absolute witnesses for* **Users***: the system takes their union. Similarly for* **Schema***:*

```
SELECT DISTINCT s.*
FROM Users u, Schema s, Groups g
WHERE u.ts = s.ts and s.irid = 'patients' and u.uid = g.uid
  and g.gid = 'Student' and u.ts > currentTime + 1 - 1209600
```

Example 2.3.4. *We now illustrate how we do log compaction for* **P1_OPT** *in Example 2.3.1. Notice that this is a* **DISTINCT** *query, and has a self-join, so the absolute witness is obtained by taking the union of two queries:*

```
Schema_w:
  (SELECT DISTINCT ON (p1.ts), p1.*
   FROM Schema p1, Schema p2
   WHERE p1.ts = currentTime+1 and p2.ts = currentTime+1
     and p1.ts = p2.ts and p1.irid = 'Navteq' and p2 != 'Navteq')
  UNION
  (SELECT DISTINCT ON (p2.ts), p2.*
   FROM Schema p1, Schema p2
   WHERE p1.ts = currentTime+1 and p2.ts = currentTime+1
     and p1.ts = p2.ts and p1.irid = 'Navteq' and p2 != 'Navteq')
```

This query, however, returns the empty set, because all occurrences in **Schema** *have the timestamp strictly less than* **currentTime***. As a consequence, if this were the only policy, then the system will not generate any log at all.*

2.3.2 Policy Minimization

Next, we focus on the policies themselves, $\mathbf{\Pi} = \{\pi_1, \ldots, \pi_k\}$ and describe optimized ways to compute them.

Interleaved Policy Evaluation

Recall that a policy i is satisfied when π_i returns false. Hence, a query can proceed when all π_i return false. We make two observations. First, by far the most common case is when the policies evaluate to false. This is the normal use of the database, when users ask queries that comply with the policies: our main goal is to speed up this case. Second, if a policy π_i returns false (the common case), it is often for a simple reason, for example because some part of π_i is false, e.g. one predicate or a join of only two relations; it suffices to find a partial expression of π_i that evaluates to false, then we do not need to compute the entire policy. Based on this intuition we develop the following optimization.

We review two standard definitions. (1) A policy query π is *monotone* if, for any two instances $\mathbf{L} \subseteq \mathbf{L}'$ and $D \subseteq D'$, we have $\pi(t, \mathbf{L}, D) \subseteq \pi(t, \mathbf{L}', D')$. All SPJU queries, and Boolean queries with aggregate conditions of the form `having count([distinct] x) > k` are monotone. In contrast, conditions of the form `having count(...) < k` are non-monotone. (2) Given two policy queries π, π' we say that π is *contained* in π' if, for all $\mathbf{L}, D$, $\pi(t, \mathbf{L}, D) \subseteq \pi'(t, \mathbf{L}, D)$. Since policies are Boolean queries, we denote containment by $\pi \Rightarrow \pi'$, which means that if π is true then π' is necessarily true (but not the other way around).

Let $\mathbf{S} \subseteq \mathbf{L}$ be a subset of the log relations. The *partial policy* for π and $\mathbf{S}$, in notation $\pi_{\mathbf{S}}$, is the policy obtained from π by simply removing all references to relations in $\mathbf{L} - \mathbf{S}$ and also removing the `having` condition if it refers to any relations in $\mathbf{L} - \mathbf{S}$. That is, the partial policy performs only the joins on the relations in $\mathbf{S}$ and in the database D. Note that the query is always syntactically correct. We prove the following:

Lemma 2.3.4. *Suppose π is a monotone policy without aggregates. Then, for any partial policy, we have $\pi \Rightarrow \pi_{\mathbf{S}}$. The same holds for a monotone policy with aggregates, if all relations in $\mathbf{L} - \mathbf{S}$ are joined on their keys.*

Proof. (Sketch) First, if π is a Conjunctive Query without aggregates, then, there exists a query homomorphism from $\pi_{\mathbf{S}} \rightarrow \pi$, which maps every atom of $\pi_{\mathbf{S}}$ to the same atom in

π; by the classic result on conjunctive query containment [22] we conclude that $\pi \Rightarrow \pi_{\mathbf{S}}$. For queries that have an aggregate condition `having count(...) > k` we note that the relations in $\mathbf{L} - \mathbf{S}$ cannot raise the count, because they are joined on their keys. □

Algorithm 3: Interleaved Policy Evaluation

input : A set of monotone policies $\mathbf{\Pi}$ and a query q

output: `true` if a violation occurs, `false` otherwise

begin
- $\mathbf{S} \leftarrow \emptyset$
- **for** $f_i \in \mathbf{f}$ **do**
  - // Update log with its increment and timestamp t
  - $R_i \leftarrow R_i \cup (\{t\} \times f_i(q, D))$
  - $\mathbf{S} \leftarrow \mathbf{S} \cup R_i$
  - **for** $\pi_k \in \mathbf{\Pi}$ **do**
    - $\pi' \leftarrow \pi_{k,\mathbf{S}}$
    - **if** $\pi'(t, \mathbf{S}, D) = \emptyset$ **then** $\mathbf{\Pi} \leftarrow \mathbf{\Pi} - \{\pi_k\}$
    - // See §2.3.3, Improved Partial Policies
  - **if** $\mathbf{\Pi} = \emptyset$ **then** `break`;
- **return** $\mathbf{\Pi} \neq \emptyset$

Based on the lemma, Algorithm 3 describes an optimized strategy for evaluating a set of monotone policies $\mathbf{\Pi}$. We add one by one the log relations R_i to the set $\mathbf{S}$. At each step, we compute the log function f_i to obtain all new tuples added by the current query q to $\mathtt{R}_i$, then check for all policies π_k the conditions that refer just to the current log relations in $\mathbf{S}$: if any such policy returns false, we remove it from $\mathbf{\Pi}$. In §2.3.3, we present an extension to interleaved that can also remove policies that do not return false. Next, we add a new log relation R_i to $\mathbf{S}$ and iterate. We stop when either all log relations have been added to $\mathbf{S}$ or when $\mathbf{\Pi}$ becomes empty. If $\mathbf{\Pi} = \emptyset$, then it means that all policies have been found to be false (the common case); otherwise, there was at least one violation.

An important decision is in which order to add the log relations R_i to $\mathbf{S}$. Our current system uses a fixed order, which is chosen experimentally, offline, by optimizing over an existing log. In our prototype implementation, the order was experimentally found to be: `Users` followed by `Schema` followed by `Provenance`.

Example 2.3.5. *Consider policy* `P2b` *from Example 2.2.2, and suppose we add the log relations in this order to* **S***:* `Users`, `Schema`*. Then we obtain two partial policies in addition to the full policy* `P2b`*:*

```
P2d: SELECT DISTINCT 1
     FROM Groups g, Clock c
     WHERE g.gid = 'Student'
P2c: SELECT DISTINCT 1
     FROM Users u, Groups g, Clock c
     WHERE u.uid = g.uid AND g.gid = 'Student'
       and u.ts > c.ts - 1209600
     HAVING COUNT(distinct u.uid) > 10
P2b: SELECT DISTINCT 1
     FROM Users u, Schema s, Groups g, Clock c
     WHERE u.ts = s.ts and s.irid = 'patients'
       and u.uid = g.uid AND g.gid = 'Student'
       and u.ts > c.ts - 1209600
     HAVING COUNT(distinct u.uid) > 10
```

The system starts optimistically by checking the first partial policy, `P2d`*; if there are no users in the* `Student` *group, then the policy is guaranteed to be satisfied. Otherwise the system proceeds with the second partial policy* `P2c`*, which checks if at least 10 users from that group have asked any queries in the last 14 days; if there are no such users then the policy is satisfied. Only if there are such users does the system proceeds with the full policy* `P2b`*.*

Policy Unification

Finally, we describe a simple, but very effective optimization, which consolidates multiple policies with the same structure but different constants into a single policy that uses a separate table for the constants. Variants of this technique have been employed in different settings in prior research [42] and is an example of transforming *queries into data*. We explain this technique through an example.

Example 2.3.6. *Consider parameterized policies:*

```
Px = SELECT DISTINCT 'Error' FROM Users u, Groups g
     WHERE u.uid = g.uid AND g.gid = X
     HAVING COUNT(distinct u.uid) > 10
```

Here `X = {'Student', 'Postdoc', ...}`. *We unify them in a single policy:*

```
P1 = SELECT DISTINCT 'Error' FROM Users u, Groups g, Constants c
     WHERE u.uid = g.uid AND g.gid = c.const
     GROUP BY c.const HAVING COUNT(distinct u.uid) > 10
```

The new table ***`Constants`*** *contains all constants* ***`'Student', 'Postdoc', ...`*** *used in the policies.*

2.3.3 Advanced Optimizations

We describe two advanced optimizations that extend the previous optimizations. Both apply to policies where all log-generating functions join on the timestamp.

Preemptive Log Compaction. The optimization is to compute the partial query `LCQ'` for the log compaction query `LCQ` using only the logs that have been generated. If `LCQ'` is empty, `LCQ` would also be empty, so we might as well not generate the remaining logs.

Improved Partial Policies. For interleaved execution (§2.3.2), we only stop early, if a partial policy is satisfied (*i.e.*, produces the empty output). But we can do better. If the partial policy produces a non-empty output, and that output does not depend on the latest increment to the logs (at the current time), then no tuple from the current timestamp would contribute to the output of the policy. But since the policy was tested to be valid in the past, it will continue to be valid in the current timestamp. We evaluate this technique in Section 2.4.4.

Combining Preemptive Log Compaction and Improved Partial Policies In practice, both the above advanced optimizations can be used together. The idea behind *improved partial policies* also extends to *preemptive log compaction* where further logs need not be generated if the partial query `LCQ'` for the log compaction query `LCQ` does not depend on any tuples of the newly generated log increments.

2.3.4 Putting It All Together

DataLawyer puts all optimizations together as follows: **Offline Phase.** Perform the following static analysis on the policies:

1. Apply policy unification (§2.3.2). Denote $\mathbf{\Pi}$ the resulting set of policies.

2. For each time-independent policy $\pi \in \mathbf{\Pi}$, replace it by its optimized rewriting π_{ind} (§2.3.1). Let $\mathbf{\Pi}_{mon} \subseteq \mathbf{\Pi}$ denote the set of monotone policies (§2.3.2).

Online Phase. For each query q, perform the following actions, in order.

1. Run the Interleaved Policy Evaluation Algorithm 3 on the monotone policies $\mathbf{\Pi}_{mon}$. If it returns `true`, abort and make no changes to the usage logs. Else, let $\mathbf{L}_{gen}$ denote the log relations computed by the algorithm.

2. For each non-monotone policy $\pi \in \mathbf{\Pi} \setminus \mathbf{\Pi}_{mon}$, compute the log relations R not yet in $\mathbf{L}_{gen}$ and add them to $\mathbf{L}_{gen}$, by applying the corresponding log function f. Then, evaluate π. If any policy returns `true`, abort.

3. Run log compaction (Algorithm 2) over $\mathbf{\Pi}$. Recall (§2.3.1) that only time-dependent policies contribute anything to the log. As a further optimization, do Preemptive Log Compaction (§2.3.3) to prune out policies that do not require log compaction.

4. Flush log to disk. Execute q.

2.4 Evaluation

We evaluate the overhead of policy checking with DataLawyer compared to only executing queries in PostgreSQL (§2.4.1). We also study the performance of DataLawyer's optimizations compared to the NoOpt strategy: log compaction (§2.4.2), time-independent policies (§2.4.3), interleaved policy evaluation (§2.4.4), and policy unification (§2.4.5).

	Policy
P_1	A maximum of `10 distinct` users can query the database from the group of users from university `'X'` in any window of `200ms`
P_2	User with `uid = 1` can not `join poe_order` with any other relation except `poe_med`
P_3	User with `uid = 1` can not execute any query on relation `d_patients` that returns more than 100 tuples
P_4	No output tuple on a query over `chartevents` for `uid = 1` should have less than or equal to 3 input tuples contributing to it
P_5	In no span of `3s`, aggregated over all queries, can user with `uid = 1` produce output that uses more than half the total tuples in `d_patients`.
P_6	In any span of `300ms`, user with `uid = 1` should not use the same input tuple from `d_patients` more than `1000` times.

Table 2.2: The policies used in the experiments. P_1 uses the cheapest log-generating function (`Users`). Note, group `'X'` contains user 1 but not user 0. P_2 uses both `Users` and `Schema` log-generation functions. The remaining policies are the most expensive policies and use the `Provenance` log-generating function.

We run all experiments on the MIMIC-II dataset, which is an anonymized dataset of readings from advanced Intensive Care Unit patient monitoring systems for over 33000 patients, collected over a period of seven years. The subset of data we experiment on is over 21GB in size. All experiments are conducted on a single server running PostgreSQL 9.2 over OS X 10.9.4, equipped with a 2.7 GHz Intel Core i7 processor and 16 GB DDR3 RAM.

The experimental setup consists of enforcing the policies in Table 2.2, which are adapted to our dataset from the policies we introduced in Table 2.1. We experiment with two users, with `uid = 0` or `uid = 1`. The users repeatedly submit one of four distinct queries as shown in Table 2.3. The query times range from $0.25ms$ to approximately $2s$.

2.4.1 Overhead of DataLawyer with All Optimizations Enabled

We first address the fundamental question of DataLawyer's overall practicality: What is the overhead of policy evaluation with DataLawyer compared with plain query evaluation with

	Time	Query
W_1	$0.25ms$	`SELECT * FROM d_patients WHERE subject_id = 186`
W_2	$15.69ms$	`SELECT c.subject_id, p.sex, COUNT(c.subject_id) FROM chartevents c, d_patients p WHERE c.subject_id = 489 AND p.subject_id = c.subject_id AND itemid = 211 GROUP BY c.subject_id, p.sex HAVING COUNT(c.subject_id) > 1`
W_3	$170.43ms$	`SELECT c.subject_id, p.sex, COUNT(c.subject_id) FROM chartevents c, d_patients p WHERE c.subject_id < 1000 AND c.subject_id > 930 AND p.subject_id = c.subject_id AND itemid = 211 GROUP BY c.subject_id, p.sex HAVING COUNT(c.subject_id) > 23`
W_4	$1756.6ms$	`SELECT c.subject_id, p.sex, COUNT(c.subject_id) FROM chartevents c, d_patients p WHERE c.subject_id < 1450 AND c.subject_id > 800 AND p.subject_id = c.subject_id AND itemid = 211 GROUP BY c.subject_id, p.sex HAVING COUNT(c.subject_id) > 1000`

Table 2.3: Queries used in experiments. Queries selected to cover a wide range of runtimes.

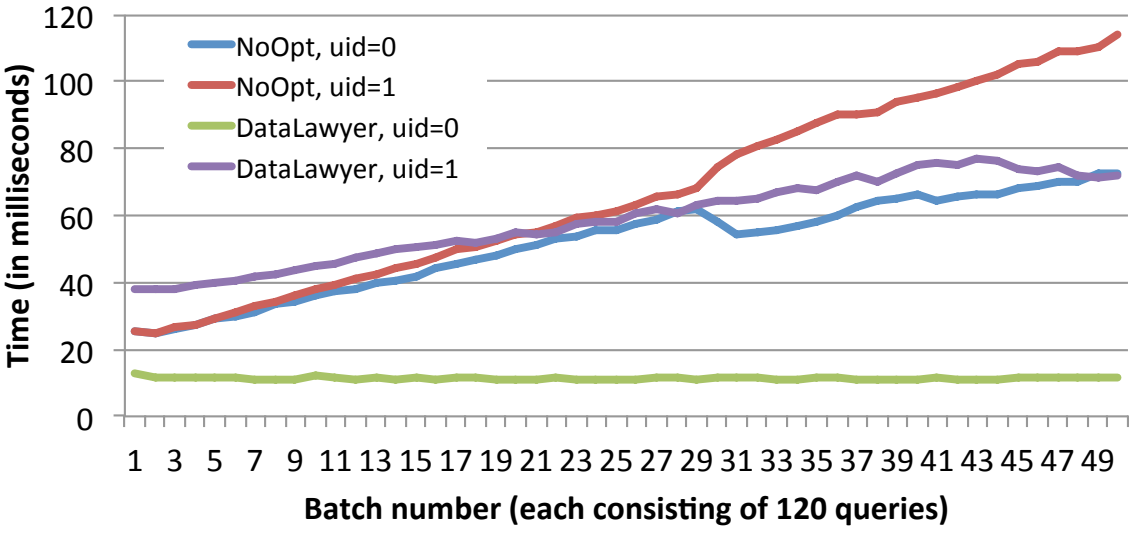


Figure 2.1: Policy and query evaluation time for DataLawyer and NoOpt on policy P_6 and query W_1 (fastest). DataLawyer's overhead stabilizes while NoOpt's grows continuously and quickly exceeds DataLawyer's overhead. Queries are submitted in batches of 120. The x-axis shows the batch number. The y-axis shows the average query and policy evaluation time for each batch.

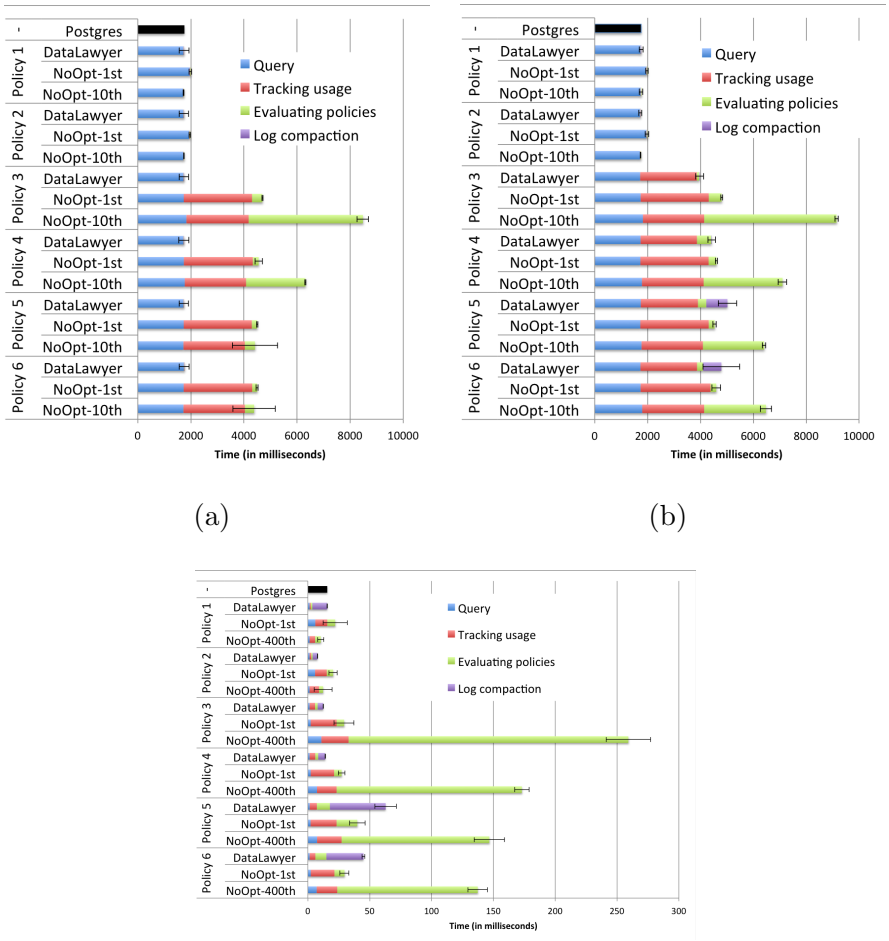


Figure 2.2: Policy and query evaluation time (in ms) for DataLawyer and NoOpt for all policies. Figures 2.2a and 2.2b show times for query W_4 for users with `uid=0` and `uid=1`, respectively; Figure 2.2c shows times for query W_2 for `uid=1`. The topmost bar is the query's evaluation time on an unmodified PostgreSQL (warm cache). DataLawyer's numbers were measured over a warm cache, once the overhead stabilizes. For NoOpt, we show the time for the first query (cold cache) and for the 10^{th} query (warm) for the first two figures, and the 400^{th} query (warm) for the last one. Error bars show the standard deviation over 12 runs for NoOpt and 50 runs (or more) for DataLawyer.

PostgreSQL and compared with NoOpt?

To answer these questions, we execute (multiple times) each query from Table 2.3 while enforcing one policy at the time. This lets us measure the performance of DataLawyer (with all optimizations turned on) and NoOpt, both with increasingly expensive policies and with increasingly expensive queries. We measure the query execution time, the overhead of tracking usage, the overhead of evaluating policies, and additionally, for DataLawyer, the overhead of compacting the logs. Further, for each policy-query combination, we run the experiment as user 0 (where DataLawyer can quickly infer through interleaved policy evaluation that no policy is applicable) and as user 1 (where the policies must eventually be evaluated in full to determine compliance).

Figure 2.1 shows how policy checking overhead grows continuously for NoOpt, while it quickly stabilizes to an approximately constant overhead for DataLawyer. The figure shows what happens for query W_1 and policy P_6 but the same trends occur for all policies and queries. In fact, for user 0, query W_4, and policy P_4, the overhead is $14ms$ for DataLawyer, while exceeding $2.7s$ for NoOpt after just 10 queries, leading to an almost $330\times$ reduction in overhead. The cause for the growing overhead with NoOpt is the increasing usage history. DataLawyer's log compaction optimization prunes the parts of the log that are no longer needed, keeping the overhead constant after an initial ramp-up period. Of course, this pruning initially adds overhead compared with NoOpt.

We consider the overhead of policy evaluation in more detail, focusing on W_4 (long query) and W_2 (short query). For NoOpt, because the overhead grows, we measure the overhead after the first and tenth query for W_4 and the first and 400^{th} query for W_2. For DataLawyer, we measure the overhead once it stabilizes. Figure 4.4 shows the results.

As seen, for long queries (W_4) and cheap policies (P_1 and P_2), the overhead of policy checking is negligible for both DataLawyer and NoOpt[6]. For short queries (W_2), even for cheap policies, the overhead becomes visible. But, the policy checking overhead remains low

[6]For policies 1 and 2, it appears as if the later evaluations are faster for NoOpt, but that is because the first query was over a cold cache. In the long run, this advantage vanishes.

(below 50 ms), maintaining the interactive speeds of these short queries.

Overheads become significant for the more expensive policies P_3 through P_6, which all use the `Provenance` log-generating function. As seen in Figure 2.1, the overhead with NoOpt grows quickly. For example, the total time taken to check and execute queries for policy P_3 increases by 1.8× and 1.9× for Figures 2.2a and 2.2b, respectively, between the first and the tenth query while increasing by 8.8× for Figure 2.2c between the 1^{st} and the 400^{th}. In contrast, DataLawyer maintains a significantly lower and constant overhead.

These overheads have two components: the overhead of tracking usage and the overhead of evaluating the policies. For NoOpt, the overhead of the former is solely dependent on the usage logs mentioned in the policy definition. For a given query, the overhead is thus approximately constant across policies that use the same logs (*e.g.*, P_3 through P_6). For DataLawyer, the overhead of tracking usage is split into tracking the usage and compacting the log. Here, interleaved query evaluation and preemptive log compaction (§2.3.3) enable DataLawyer to avoid generating any usage logs in some cases such as for user 0 on query W_4 in Figure 2.2a. In other cases, DataLawyer generates the logs and stores them in temporary tables in memory. Unlike NoOpt, DataLawyer compacts these tables before writing anything to disk, which is why the overhead of tracking usage is smaller for DataLawyer than for NoOpt even when both use the same logs, especially apparent for query W_2. Log compaction can add a significant overhead, which nevertheless pays off within the first 10 to 400 queries in this experiment. Log compaction's overhead is a function of existing log size, log increment size, and policy complexity. Hence, for policies P_1 and P_2 that rely on small usage logs, the overhead is tiny, while for policies P_5 and P_6, that rely on all three logs and require multiple joins and aggregations, the overheads are noticeable. Interestingly, in multi-threaded systems, one can return the result of the query to the user before log compaction finishes, thus the effective latency seen by the user may, in some cases such as for W_2 (policies P_5 and P_6), be as little as 23% of the time reported by a single-threaded system.

The overhead for evaluating policies is what dominates the overhead for NoOpt in later stages, while it stays constant and small for DataLawyer. Log compaction helps to keep

the policy evaluation time small and constant for DataLawyer as illustrated in Figure 2.2b. Additionally, when applicable, as it does in case of user 0, interleaved evaluation allows DataLawyer to evaluate policies with practically no overhead; in our experiments, the maximum overhead over all policies and queries for user 0 was $3ms$ whereas the corresponding overhead for user 1 was $540ms$.

Even with all optimizations, for expensive queries like W_4, and for the most expensive policies, DataLawyer imposes a relative overhead up to $2\times$ to $3\times$ for `uid=1`. In general, for `uid=1`, a 100% overhead is unavoidable for policies P_3 through P_6, since they need the provenance, which is usually more expensive to generate than evaluating the query.

2.4.2 Log Compaction Optimization

As the end-to-end results show, log compaction (§2.3.1) is crucial for DataLawyer to maintain a constant overhead as more queries are executed against the database.

Log compaction removes dispensable tuples from the usage log and this reduces the time spent in policy evaluation for policies that are not time-independent. Note that both DataLawyer and NoOpt keep the newly generated usage log increments in memory, only pushing them to disk after verifying all policies. Thus, log compaction may also reduce the tuples from the latest log increment that are appended to the usage log, after checking each valid query. Here, we measure three phases of log compaction: (a) *marking*: the log compaction queries are executed over the disk-resident log and its in-memory increment, to determine which tuples to retain, and they are marked, (b) *delete*: the unmarked tuples are deleted, and (c) *insert*: the remaining tuples in the increment are appended to the log on disk.

To determine which tuples need to be removed DataLawyer must execute possibly multiple log compaction queries. Therefore, unless enough tuples are pruned this can be a significant overhead. Figure 2.3 shows the overhead of this optimization for three of the six policies: policies 1, 5, and 6 for the four queries by user with `uid=1`. No log pruning is needed for time-independent policies 2, 3, and 4 and hence they are not shown in the graph.

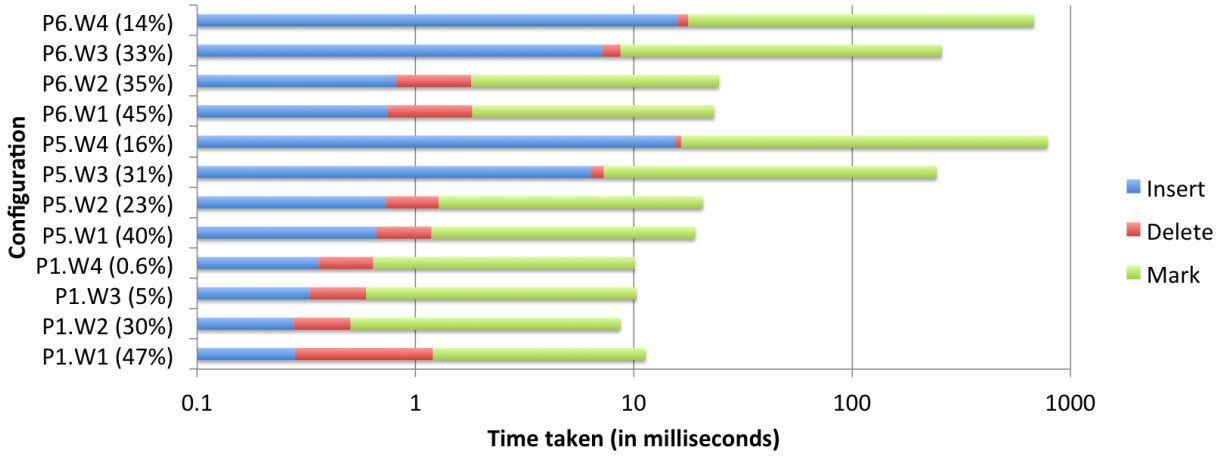


Figure 2.3: Overheads of Log Compaction: Time taken in the three phases of log compaction. Mark identifies the tuples to discard; Delete removes them; Insert appends the new tuples that were not marked for deletion. A configuration such as P6.W3 is interpreted as query W3 tested for policy 6. The percentages in parenthesis is the fraction of time spent in log compaction compared to the total policy checking and query evaluation time.

We explain what DataLawyer prunes for each policy: for P_1, the algorithm only retains the latest timestamp of the latest query, and only by the users in the group 'X'; for P_5, it only retains log entries for user 1's queries over `d_patients` in the window specified and only retains the latest instance of the tuples accessed by the user; and for P_6, it prunes out the tuples outside the sliding window.

For policies 1, 5, and 6, and for all queries, the bulk of the overhead of compacting the usage log is the "marking" stage where the tuples to be retained are selected. The high overhead is because 3 passes are made over the usage logs: the first to unmark all tuples, the next to compute the log compaction query which generates a set of tuple ids to retain, and the third pass to mark these tuples for retention. In contrast, NoOpt's overhead, apart from computing the usage logs, is about the same as the "insert" phase (since NoOpt does not prune any tuples, it may take slightly longer). Interestingly, as Figure 4.4 shows, in spite of

Count	P_2	P_2 - No ti	P_3	P_3 - No ti	P_4	P_4 - No ti
1	205	222	491	924	653	1355
5	198	208	473	881	643	2732
10	202	211	471	900	685	5110
15	212	229	480	949	648	8046
20	199	210	475	894	655	11809

Table 2.4: Policy and query evaluation time (in ms) for DataLawyer after executing multiple counts of query W_3 with time-independent policies 2, 3, and 4. Runtimes are reported with and without (represented as "No ti") the time-independent optimization. In both cases, all other optimizations are enabled.

such high overhead for pruning the log, the optimization pays off rapidly.

In our experiments, DataLawyer prunes the log after each new query. Such eager pruning, however, is not necessary. Instead, DataLawyer could compact the log less frequently or whenever the system has idle resources to further reduce the policy checking overhead.

2.4.3 Time-Independent Policies Optimization

We now test the impact of the time-independent optimization (§2.3.1) in the presence of the other optimizations. Recall that for this optimization, DataLawyer automatically adds extra constraints on the `ts` attribute, to enable log compaction to later prune the entire log. In fact, in our implementation, DataLawyer flags time-independent policies and never stores the log on disk in the first place thus completely avoiding any log-compaction-related checks and deletes.

Policies 2, 3, and 4 are time independent. Table 2.4 shows the time taken by DataLawyer to evaluate these policies on query W_3, once with this optimization on, and once without. The primary benefit of the time-independent optimization over basic log compaction, is that this optimization allows DataLawyer to prune the log even for policies that involve aggregates but have no sliding time windows. For example, for policies 3 and 4, log compaction on the

original policies without the added predicates on time does not prune any tuples. As a result, the overhead for both policies grows over time. The log compaction algorithm can not reason over aggregates and instead, we compact the corresponding full query, which ends up selecting all tuples for retention. In contrast, identifying the policies as time-independent permits the system to discard these tuples.

A secondary benefit is that, although the optimization can not avoid generating the logs, it avoids running the log compaction tests or appending any tuples to the disk.

Overall, in our experiments, this optimization halved the policy evaluation and query time for policies 3 and 4. Not much difference is seen in the case of policy 2 since it is a much cheaper policy to check and it produces a tiny log.

Thus, to ensure high performance, both optimizations, time-independence and log compaction prove beneficial.

2.4.4 Interleaved Policy Evaluation

We now evaluate the benefit of interleaved policy evaluation (§2.3.2). We quantify both the benefit when the optimization leads to early pruning as well the extra overhead when it does not. The overhead is due to executing multiple queries, each an approximation of the policy, instead of directly executing the original policy.

We evaluate each policy in isolation, for each query, once for user 0 and then for user 1. By design, interleaved execution prunes the policy after generating the cheapest log, `Users`, for user 0; and this provides an upper bound on its benefits. For the other user, interleaved execution only leads to an overhead with no pruning. For comparison, we provide the runtime with this optimization turned off.

Figure 2.4 shows that for user 0, interleaved policy evaluation can cut the runtime by more than half compared with not using the optimization. The resulting policy checking overhead drops to within 2.5% of the query evaluation time and remains nearly constant across all policies.

In user 1 though, interleaved execution does not lead to early pruning. DataLawyer

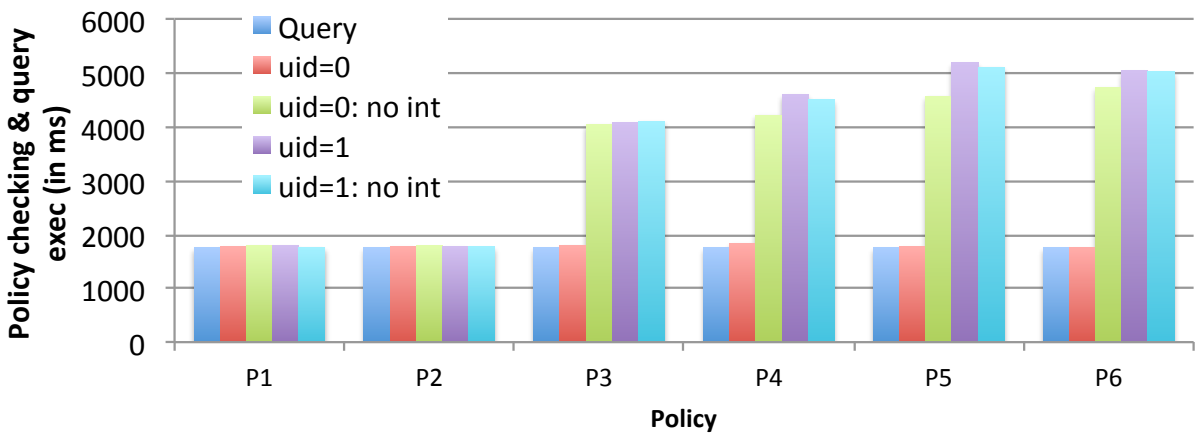


Figure 2.4: Policy and query evaluation time (in msecs) for each policy and query W_4. We consider two users, `uid=0` and `uid=1`, and for two versions of DataLawyer, one with all the optimizations and one with all optimizations but interleaved execution (indicated by "no int").

without interleaved performs better than with interleaved. However, the differences are small. For the query shown, the maximum difference was of 1.7% of the runtime without interleaved. Even for the other queries (not shown), the differences are small, both in absolute as well as relative terms.

Finally, we test a mix of both the easy and the hard cases to highlight the benefit of the advanced optimizations (§2.3.3). Figure 2.5 shows a run of 10 queries from W_4 with a single policy P_5. Here, we alternate between a query from user with `uid = 0`, the easy case, and a query from the user with `uid = 1`, the hard case. Without the advanced optimizations, all but the first easy-case queries take about the same amount of time as the hard case to check the policy and execute the query; this is because the partial policies do not return empty answers, instead, they return the tuples in the logs generated and stored by the hard cases. However, with the improved partial policies, DataLawyer can check that the partial policy is not using any part of the log generated by the easy case and hence can terminate early, and using preemptive log compaction, it can also avoid the log compaction overhead for the easy case. Thus, with advanced optimizations, the policy checking overheads of queries can

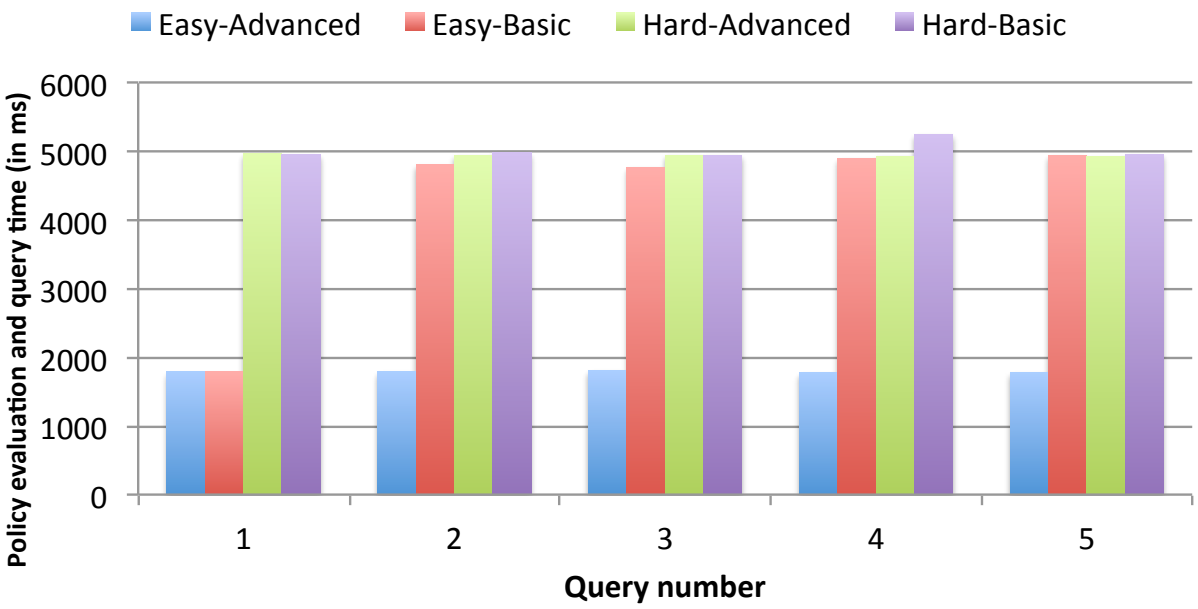


Figure 2.5: Advanced Interleaved Optimization: A comparison of the time to verify P_5 on an easy and hard mix of queries from workload W_4 with and without the advanced optimizations of §2.3.3. The easy and hard queries alternate with a total of 5 such queries each.

be isolated from each other.

Another benefit of interleaved optimization is that in the presence of multiple policies, as opposed to a single policy as in this experiment, the benefits are additive, while the overheads are sub-additive. The benefits add up since the time saved by early pruning for each policy is independent of whether another policy is pruned or not, whereas the overheads are sub-additive since the log generation, which can be expensive as for policies 3, 4, 5, and 6 for `Provenance`, is done once and the output is shared across the policies.

2.4.5 Policy Unification Optimization

We now answer how the time to evaluate policies changes as we increase the number of policies, where policies are identical except for their parameters.

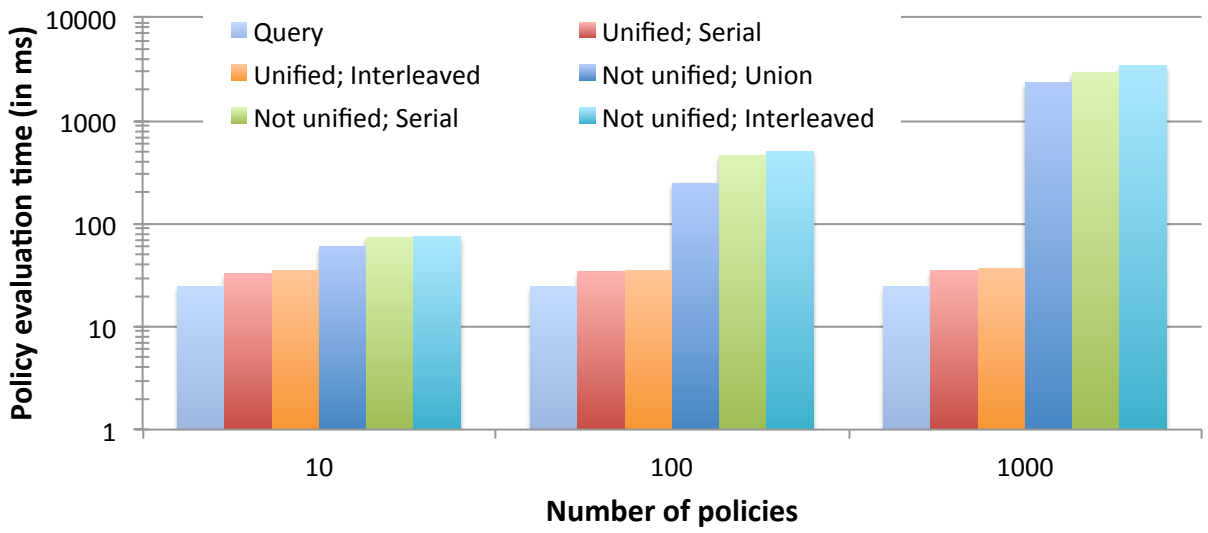


Figure 2.6: Policy Unification: A comparison of the average time to verify the policies (using either union, serial, or interleaved policy evaluation) and execute the query as we scale up the number of policies that can be unified to a single policy.

In this experiment, we check policy P_1 for query W_1. We vary the number of policies by two orders of magnitude by running three experiments: (a) 10 users each running 1000 queries (and a policy like P_1 for each user), (b) 100 users and 100 queries each (and a policy like P_1 for each user), and (c) 1000 users running one query each (and a policy like P_1 for each user). All queries execute in a round robin fashion. In this setting, the total number of queries executed remains constant but the number of users and thus the number of policies grows from 10 to 1000.

For the case when the policies are not unified, we compare three policy evaluation strategies : 1) **union**: Union all the (boolean) policies and execute one large policy, 2) **serial**: Execute policies one at a time, and 3) **interleaved**: Use interleaved execution (§2.3.2). For the case of the single unified policy, we compare serial (serial and union are identical with one policy) to interleaved.

Figure 2.6 shows that irrespective of the strategy, without unification, policy checking

time grows linearly with the number of policies. The union is the most efficient because it avoids the overhead of multiple JDBC calls (serial takes from 23% to 87% more time to check the same policies); while interleaved is costlier (up to 16% for 1000 users) than serial since interleaved makes 2× more JDBC calls that serial for the specific policy used in this experiment.

Alternatively, after unifying policies to one, to evaluate all the policies takes constant time, even after scaling up by two orders of magnitude, and *independently* of the policy evaluation algorithm used. This is because the unified policy introduces a small dataset (a max of 1000 rows in this experiment) to join with, which easily fits in memory.

This shows that naïve policy checking time is linear in the number of policies; but the equivalent unified policy runs in constant time irrespective of the number of merged policies.

2.5 Discussion

DataLawyer's approach enables the expression of a wide variety of policies, but there are limitations to what can be expressed. We identify two limitations: First, boolean policies only allow accept/reject semantics. As a result, DataLawyer cannot support policies that require other semantics, such as for example creating a log entry when a violation occurs. Second, DataLawyer does not support policies defined over the actions of DataLawyer itself. Were that allowed, we could define a policy preventing DataLawyer from rejecting two successive queries from the same user leading to unenforceable sets of policies.

Another limitation of DataLawyer is that it cannot enforce *all* expressible policies *efficiently*. Currently, we only have full support for policies with monotone aggregate conditions (*e.g.*`having count([distinct] x) > k`, see §2.3.2), and only limited support for non-monotonic aggregates. Additionally, some policies are *"hard"* because they require storing a significant amount of history. An interesting area of future work is to use approximate policies to improve performance: The system first runs a simpler test that quickly validates most queries, but occasionally flags a valid query as suspicious and spends extra time to do the precise check.

A benefit of DataLawyer's approach is its potential extensibility to new domains by defining one or more new log-generating functions. These functions can be arbitrary pieces of code. We give two examples. First, consider a policy that restricts queries from 'mobile' devices to output sizes of 10 tuples. To enable such a policy in DataLawyer one has to write a new log-generating function that parses the database connection string or the user-agent headers and populates a new table in the usage log with device information; the policy itself is a simple SQL query over the new usage log. As another example, consider a tweak on policy P_4 from Table 2.1 to make it sensitive to the server load: "no user should be able to issue more than 50 requests per hour when the system load exceeds 80%." To implement such a policy one must, (a) define a log-generating function to populate the usage log with the current system load, and (b) write the corresponding SQL query.

Two important future research problems are both related to usability. The first is to help users debug queries that are deemed non-compliant. The other is to reduce the effort of translating text policies to our framework. Our survey indicated that there is a lot more structure to these policies and it may be possible to come up with templates (domain specific, if required) that can be later tweaked to get the set of policies for an organization. Policy generation by example might be another useful direction for future work.

Finally, we note that, while DataLawyer is an important step toward automatic enforcement of data use policies, it does not obviate the need for signed agreements and lawyers, because it does not control what happens to the data once it leaves the system.

2.6 Conclusion

We developed DataLawyer, a middleware system to specify and enforce data use policies on relational databases. Our approach includes a SQL-based formalism to precisely define policies and novel algorithms to automatically and efficiently evaluate them. Experiments on a real dataset from the health-care domain demonstrate overhead reductions of up to 330× compared to a direct implementation of such a system on existing databases.

Chapter 3

PRICING COMPUTATION

Over the past several years, "cloud computing" has emerged as an important new paradigm for building and using software systems. Multiple vendors offer cloud computing infrastructures, platforms, and software systems including Amazon [7], Microsoft [14], Google [47], Salesforce [95], and others. As part of their services, cloud providers now offer data-management-in-the-cloud options ranging from highly-scalable systems with simplified query interfaces (*e.g.*, Windows Azure Storage [15], Amazon SimpleDB [12], Google App Engine Datastore [48]), to smaller-scale but fully relational systems (SQL Azure [71], Amazon RDS [9]), to data intensive scalable computing systems (Amazon Elastic MapReduce [8], to highly-scalable unstructured data stores (Amazon S3 [11]), and to systems that focus on small-scale data integration (Google Fusion Tables [46]).

Existing data-management-as-a-service systems offer multiple options for users to trade-off price and performance, which we call generically *optimizations*. They include views [3] and indexes (*e.g.*, users can create indexes in SQL Azure and Amazon RDSautomatically indexes data), but also the choice of physical location of data –which affects latency and price (*e.g.*, Amazon S3)– how data is partitioned (*e.g.*, Amazon SimpleDB data "domains"across SQL Azure instances), and the degree of data replication (*e.g.*, Amazon S3 standard and reduced-redundancy storagedeployment, Amazon). Cloud systems have an incentive to enable all the right optimizations, because this increases their customer's satisfaction and can also optimize the cloud's overall performance.

Today, data owners most commonly pay all costs associated with hosting and querying their data, whether by themselves or by others. Data owners also choose, when possible, the optimizations that should be applied to their data. However, there is a growing trend

toward letting users collaborate with each other by sharing data and splitting data access costs. For example, in the Amazon S3 storage service, users can currently share their data with select other users, with each user paying his or her own data access charges [10].

The combination of data sharing and optimizations creates a major challenge: how to *price optimizations when one optimization can benefit multiple users*. Implementing these optimizations imposes a cost on the cloud that needs to be recovered: resources spent on implementing and maintaining optimizations are resources that cannot be sold for query processing. The question is how to decide what optimization to implement and how to share its cost among users.

A recently-proposed approach by Kantere, Dash, *et al.*, [26, 56] addresses this problem by asking users to indicate their willingness to pay for different query performance values, observing the query workload, and deciding on the optimizations to implement based on optimizations that would have been helpful in the past (*i.e.*, based on *regret*). The cost of implemented optimizations is amortized to future queries that use them. This approach, however, has two key limitations as we show in Section 5.2. First, it assumes that users in the cloud will truthfully reveal their valuations. In practice, users will try to game the system if doing so improves their utility. Second, this approach does not guarantee that the cost of an optimization is recovered.

Given these two observations, we develop a new approach to select and price optimizations in the cloud based on Mechanism Design [79, 87]. Mechanism Design (see Section 3.2) is an area of game theory whose goal is to choose a game structure and payment scheme such as to obtain the best possible outcome to an optimization problem in spite of *selfish players having to provide some input to the optimization*. Our goal is to enable the cloud to find the best configuration of optimizations. For this, it needs users (*i.e.*, selfish players) to reveal their valuation for these optimizations.

The most closely related approaches from the Mechanism Design literature are cost-sharing mechanisms [74]. Given a service with some cost, these mechanisms decide what users to service and how much the users should pay for the service. In this chapter, we show,

in Section 3.3, how to easily adapt this technique from the theory community to the simplest problem of pricing a single optimization when all users will access the system for a single time-period (*i.e.*, offline game).

The problem of pricing optimizations in the cloud, however, raises two additional challenges. First, in the cloud, users change their workloads, join and leave the system at any time. Such dynamism complicates the problem because users now have new ways of gaming the system: they can lie about the time when they need an optimization and they can emulate multiple users. Dynamism requires an online mechanism. Second, multiple optimizations are available in the cloud. In the simple case, the value that a user derives from a set of optimizations is simply the sum of individual optimization values. We call such optimizations additive. In other cases, the total value from a set of optimizations may be given by a more complex function. In this chapter, as a first step, we consider substitutive optimizations, where the user only wants to pay for one optimization in a set (see Section 3.4). For example, a user may be willing to pay either for an index that accelerates a join or for a materialized view that pre-computes the join but not both. Prior work in mechanism design does not handle all the requirements at a time (See Section 5.2). We develop a suite of mechanisms, that can handle all these challenges at the same time.

We seek the following three standard properties for our mechanisms. First, we want the mechanisms to be *truthful*, also known as *strategy-proof* [79], which means that every player should have an incentive to reveal her true value obtained from each optimization. The approach by Dash, Kantere *et al.* [26, 56] mentioned above is not truthful as we discuss in Section 5.2: users can benefit from lying about their value for an optimization. We also want online mechanisms to be resilient to multiple identities, which is another way to lie about value, and to misrepresentation of the time when a user needs an optimization. Second, we want the mechanisms to be *cost-recovering*, which means that the cloud should not lose money from performing the optimizations. In the approach by Dash, Kantere, *et al.* [26, 56], the cloud first decides to implement an optimization and then it amortizes the cost to future queries that use it. Cost-recovery is thus not guaranteed. Finally, we want the

mechanisms to be *efficient*, also known as *value-maximizing* [79], which means that we want it to maximize the total social utility of the system *i.e.*, the sum of user values minus the cost of the alternative selected. For example, if several users could benefit from an expensive optimization that none of them can afford to pay for individually, then the cloud should perform the optimization and divide the cost among the users.

In summary, we make the following four contributions:

We first show how the optimization pricing problem maps onto a cost-recovery mechanism design problem (Section 3.2). We also show how the Shapley Value Mechanism [74], which is known to be both cost-recovering and truthful, solves the problem of pricing a single optimization. We propose a direct extension of the mechanism to the case of additive optimizations in an *offline* scenario, where all users access the system for the same time-period. We call this basic mechanism *Add*^{Off} *Mechanism* (Section 3.3).

Second, we present a novel mechanism for the online scenario, where users come and go, called the *Add*^{On} *Mechanism*. Users of cloud services constantly join and leave the cloud, so in practice, optimizations in the cloud need to be designed for a dynamic setting. However, it turns out to be much more difficult to design mechanisms for the online setting: algorithms that are truthful or cost recovering in the static setting cease to be so in the dynamic setting, see [79, pp 412]. We prove our new mechanism to be both cost-recovering and truthful in the dynamic setting (Section 3.4).

Third, we extend both the Add$^{\text{Off}}$ Mechanism and the Add$^{\text{On}}$ Mechanism to the case where optimizations are inter-dependent: In this chapter, we consider substitutive optimizations, where the user derives a single value for any optimization in a set. However, implementing multiple optimizations from the set does not improve the user value. We call these mechanisms *Subst*^{Off} *Mechanism* and *Subst*^{On} *Mechanism* and prove them truthful and cost-recovering (Section 3.5).

A known result is that achieving both truthfulness in face of selfish agents and cost-recovery comes at the expense of total utility [74]. We experimentally compare our mechanisms against the state-of-the art approach based on regret accumulation [26]. We show that

our mechanisms produce up to a 3× higher utility and provide the same utility for ranges of optimization costs up to 12.5× higher than the state-of-the-art approach in addition to handling selfish users and while ensuring that the cloud recovers all costs.

3.1 Motivating Use-Case

Sciences are increasingly data rich [49]: A new model for discovery is to collect large, shared datasets and enable multiple researchers to analyze them. Examples of such shared datasets include the Sloan Digital Sky Survey [99] which resulted in the publication of over 9000 papers, the upcoming LSST [66], and others [55].

These scientific collaborations are the prime motivation for this work: a non-profit organization hosts the shared data and needs to recover all associated costs by fairly charging the investigators who use the data.

In this section, we present a concrete use-case for such shared data analysis from our colleagues in the astronomy department at the University of Washington.

An important component of astronomy research involves large universe simulations, where the universe is represented by a set of particles, which include dark matter, gas, and stars. All particles are points in a 3D space with properties that include position, mass, and velocity. Every few simulation time steps, the simulator outputs a snapshot of the state of the simulated universe. Each snapshot records all properties of all particles at the time of the snapshot. Simulations of this type currently have between 10^8 to 10^9 particles (with approximately 100 bytes per particle) and output a few dozen to a few hundred snapshots per run.

For each snapshot, astronomers first run a clustering algorithm to detect clusters, called *halos*. Some halos correspond to galaxies. A significant component of the research involves studying the evolution of these halos over time. Different researchers focus on different types of halos. In particular, our astronomer colleague indicated that: "There are in general three or four different halo mass ranges that different people focus on: high mass which corresponds to a cluster, Milky Way mass, slightly less than Milky Way mass and low mass/dwarf galaxies.

[...] For example, I've been looking for Milky Way Mass galaxies, but another person in our group might be interested in the same sort of galaxies, but at a lower mass range. [The simulation] also helps us identify what environment a given halo forms in – one person might be interested in a Milky Way mass galaxy that forms in relative isolation, another person might be interested in finding a Milky Way mass galaxy that forms near many other galaxies (a rich, cluster-like environment)." [65]. Different scientists focus on different particle types and on the simulation time steps that correspond to interesting time-periods in the evolution of the halos that they study [65]. Thus, different users may need different optimizations, and the challenge is to decide which ones to implement, and who pays for them.

Both indexes and materialized views improve the performance of the SQL queries in this use-case. In Section 3.6.2, we evaluate our mechanisms on real data and queries from this use-case. We consider optimizations in the form of materialized views. Since different scientists query different parts of the data, they benefit from different materialized views.

3.2 A Mechanism Design Problem

In this section, we show how to model the problem of selecting and pricing optimizations in the cloud as a *mechanism design* [79] problem. We further show that our problem requires a type of mechanism called *cost-sharing mechanism*. In this chapter, we assume that every optimization is binary, *i.e.*, the cloud either implements it or not. We do not consider continuous optimizations (*e.g.*, degree of replication).

We consider a set of users, $I = \{1, \ldots, m\}$, who are using a cloud service provider (*a.k.a.*, *cloud*) to access and query several data sets. Any user can potentially access any data set. Let $J = \{1, \ldots, n\}$ denote the set of all potential optimizations that the cloud could offer for these datasets. For example, j may represent an index; or the fact that a data set is replicated in a second data center; or may represent an expensive fuzzy join between two popular public datasets, which is precomputed and stored as a materialized view. Once the cloud decides to do an optimization j, it may restrict access to j to only certain users; a *grant pair* (i, j) indicates that user i has been granted permission to use the optimization j. While

grant permissions artificially prevent a user from accessing an optimization, this restriction is necessary to ensure that users reveal their true value for an optimization and pay accordingly. A *configuration*, also called *alternative* is a set of optimizations j and a set of grant pairs[1] (i, j). We denote an alternative with a and the set of all possible alternatives with A. We also denote $S_j = \{i \mid (i, j) \in a\}$ the set of users who are serviced by the optimization j in alternative a.

The goal of the mechanism will be to select a configuration $a \in A$. The decision will be based on the optimization costs and their values to users, which will determine the users' willingness to pay for various optimizations.

Values to Users. Each user i obtains a certain value $v_{ij} \geq 0$ from each optimization j: *e.g.*, monetary savings obtained from increased performance or the ability to do a more complex data analysis. When multiple optimizations are performed, the total value to a user is given by $V_i(a) \geq 0$, and is obtained by aggregating the values v_{ij} for all grant pairs $(i, j) \in a$. In this and the following two sections, we consider *additive optimizations*, where the value is given by:

$$V_i(a) = \sum_{(i,j)\in a} v_{ij} \quad \geq 0 \tag{3.1}$$

In Section 3.5 we will consider *substitutive optimizations*.

An important assumption in mechanism design is that users try to lie about their true values: when asked for their value v_{ij}, user i replies with a bid b_{ij}. In the case of an additive value function, we denote $B_i(a) = \sum_{(i,j)\in a} b_{ij}$, where $B_i(a)$ is user i's *bid* about her value $V_i(a)$.

Cost to the Cloud. For each implemented optimization $j \in J$, the cloud incurs an optimization cost $C_j > 0$, which includes the initial cost of implementing the optimization (*e.g.*, building an index) and any possible maintenance costs (*e.g.*, updating the index) for the duration of the service. This cost is an opportunity cost: the resources used to perform

[1] We assume that, if an alternative contains a grant pair (i, j), then it also contains the optimization j.

the optimization cannot be sold to other users. The cost of an alternative a is then given by:

$$C(a) = \sum_{j \in a} C_j \tag{3.2}$$

While each cost C_j is small, the combined cost $C(a)$ may be large since the number of potential optimizations is large.

Payments. Once an outcome a is determined, each user i who is granted access to an optimization j must pay some amount p_{ij}. This payment is called the user's *cost-share*, and is determined based on all users' bids[2], $(b_{ij})_{i=1,m;j=1,n}$. Denoting $P_i = \sum_j p_{ij}$ the total payment for user i, her *utility* is defined as $U_i(a) = V_i(a) - P_i$. A standard assumption in Mechanism Design is that users are "utility maximizers", *i.e.*, they try to bid so as to maximize their utility [79, 87].

Cost-Sharing Mechanism Design Problem. After collecting all bids, a mechanism chooses an outcome $a_0 \in A$ that optimizes some global value function. In the case of cloud based optimizations, we will always aim to optimize the *total social utility* ("total utility" for short): The outcome's total value (Eq. 3.1) minus the outcome's cost (Eq. 3.2). Formally, the mechanism chooses the following outcome a_0:

$$a_0 = \arg\max_{a \in A} \left(\sum_{i \in I} B_i(a) - C(a) \right) \tag{3.3}$$

Such a mechanism is called *efficient* [74]. Note that the mechanism does not know the true values $V_i(a)$, but uses the bids $B_i(a)$ instead. The goal of mechanism *design* is to define the payment functions p_{ij} in such a way that all users have an incentive to bid their true values, $B_i = V_i$. A mechanism is called *strategy-proof* [79, 87], or *truthful*, if no user can improve her utility $U_i(a)$ by bidding untruthfully $B_i \neq V_i$. Truthful mechanisms are highly desirable, because when users reveal their true values, the mechanism is in a better position to select the optimal alternative.

---

[2]This is a very important point: the payment depends not only on the outcome a, but on all bids. For example, in the *second bidders' auction*, the payment of the winner is the second highest bid [87].

Another desired property for cost-sharing mechanisms is to be *cost-recovering*, *i.e.*, to only pick outcomes a_0 so that:

$$C(a_0) \leq \sum_i P_i \tag{3.4}$$

Example 3.2.1. *Consider the following mechanism. The cloud collects all bids b_{ij}. If $c_j \leq \sum_i b_{ij}$ then it performs the optimization j and asks each user to pay b_{ij} ($p_{ij} = b_{ij}$). Clearly it is cost recovering. However, it is not truthful: a user i will simply lie and declare a much lower value $b_{ij} \ll v_{ij}$, hoping that the optimization will be performed anyway and she will end up paying much less. The challenge in designing any mechanism is to ensure that it is truthful.*

Formally, a mechanism is defined as follows:

Definition 3.2.1. *A mechanism $(f, P_1, \cdots, P_m)$ consists of a function $f : \left(\mathbb{R}^A\right)^m \to A$ (called* social choice function*) and a vector of payment functions $P_1, \cdots, P_m$, where $P_i : \left(\mathbb{R}^A\right)^m \to \mathbb{R}$ is the amount that user i pays.*

The mechanisms works as follows. After collecting bids $B_1, \ldots, B_m$ from all users[3], it chooses the alternative $a = f(B_1, \ldots, B_m)$, and each user i must pay $P_i(B_1, \ldots, B_m)$.

While we would like to design mechanisms that maximize the total social utility Eq.(3.3), it is a well-known result that one cannot achieve cost-recovery (*a.k.a.*budget balance), truthfulness and efficiency [74] at the same time. In our setting, we choose to ensure only truthfulness and cost-recovery, Eq.(3.4), at the expense of some efficiency loss. Indeed, if the cloud cannot recover its cost, it will not implement the loss-making optimization.

3.3 A Mechanism for Static Collaborations

We now show how to use the Shapley Value Mechanism [74], which has many desirable properties, to solve the problem of selecting and pricing *additive* optimizations for *one time-*

[3]Each bid B_i is a function $A \to \mathbb{R}$.

Symbol	Description
$\mathbf{i}, \mathbf{j}, \mathbf{t}, \mathbf{a}$	index for users, optimizations, time-slots, and outcomes.
$\mathbf{I}, \mathbf{J}, \mathbf{T}, \mathbf{A}$	sets of users, optimizations, time-slots, and outcomes.
$\mathbf{S_j(t)}$	users serviced by optimization j at time t.
$\mathbf{CS_j(t)}$	all users serviced by optimization j up until time t.
$\mathbf{v_{ij}(t)}$	user i's true (private) value for opt j at time t.
$\mathbf{b_{ij}(t)}$	user i's stated (public) value for opt j at time t.
$\mathbf{B_i}$	$\mathbf{B_i} = (b_{ij})_{i=1,m;j=1,n}$.
$\mathbf{V_i(a)}$	user i's total, true (private) value for outcome a.
$\mathbf{B_i(a)}$	the stated (public) value of outcome a for user i.
$\mathbf{p_{ij}}$	user i's payment for optimization j.
$\mathbf{P_i}$	user i's total payment.
$\mathbf{U_i(a)}$	i's utility for outcome a.
$\mathbf{C(a)}$	outcome a's cost.
$\mathbf{C_j}$	optimization j's cost.
$\mathbf{s_i}$	slot when user i enters the system.
$\mathbf{e_i}$	slot when user i pays and leaves the system.

Table 3.1: Symbol Table. For symbols with the argument time t, we drop t for offline mechanisms.

period (*i.e.*, offline game). We extend it to online settings, where users come and go across multiple time-periods in Sec. 3.4 and to substitutive optimizations in Sec. 3.5.

3.3.1 Background: Shapley Value Mechanism

We start by reviewing the Shapley Value Mechanism [74], shown in Mechanism 3.3.1. Fix a single optimization j, let C_j be its cost and $b_{1j}, \ldots, b_{mj}$ the users' bids for this optimization. The Shapley Value Mechanism determines whether to perform the optimization or not, and, computes the set of serviced users $S_j \subseteq \{1, \ldots, m\}$, and how much they have to pay,

p_{ij}. Recall that a configuration, a, contains all grant pairs (i, j) such that $i \in S_j$. The mechanism starts by setting S_j to the set of all users, and divides the cost C_j evenly among them: $p = C_j/|S_j|$. If p is larger than a user's bid b_{ij}, that user is removed from S_j. The mechanism then re-computes a new price by dividing the cost evenly among the smaller set of users. As a result, the cost per user, $C_j/|S_j|$, may increase and additional users may need to be removed from the set S_j. The process continues until either no users remain or no further users need to be removed from S_j. Each serviced user, $i \in S_j$, pays the same amount, $p_{ij} = C_j/|S_j|$; each non-serviced user, $i \notin S_j$, pays nothing, $p_{ij} = 0$. If $S_j = \emptyset$ then no subset of users has bid enough to pay for the optimization, and it is not implemented at all. It is obvious that this mechanism is cost recovering, since $\sum_i p_{ij} = C_j$. The mechanism has also been proven to be truthful [74]: if the user i bids the true value $b_{ij} = v_{ij}$ then her utility (which is $v_{ij} - p_{ij}$, if $i \in S_j$, and 0 otherwise) is no smaller than her utility under any other bid. Indeed, suppose she bids low, $b_{ij} < v_{ij}$. Then one of two cases holds. Either she is removed from the set of serviced users S_j: in this case her utility drops to 0. Or she remains in S_j: in this case her payment p_{ij} remains unchanged, and so does her utility. Hence, she cannot increase her utility by underbidding; the reader may check that she cannot increase it by overbidding.

3.3.2 Add^{Off} Mechanism

We now propose our first mechanism for cloud optimization, under the simplest setting, when the optimizations are done offline and are additive; we remove these restrictions in the next sections. Our mechanism, called Add$^{\text{Off}}$, iterates over all optimizations and runs the Shapley Value Mechanism for each one. It adds to a, the grant pairs for all serviced users, and it implements the optimization when the set is not empty. A user pays the sum of all per-optimization payments. Since Add$^{\text{Off}}$ runs the Shapley Value Mechanism, independently, for each optimization, it follows directly that it remains truthful and cost-recovering, as the latter.

As mentioned above, it is a known result that no mechanism achieves truthfulness and

budget-balance while also being efficient. An important property of the Shapley Value mechanism is that it *minimizes utility lost* due to the cost-recovery constraint [74]. We show, in Section 4.5, how this property lets Add$^{\text{Off}}$ achieve high utility in face of selfish users compared to existing optimization pricing techniques.

Mechanism 3.3.1 Shapley Value Mechanism for computing the set of users to be serviced by an optimization j, and their cost-share p_{ij} .

Shapley-Mech

Require: Optimization cost C_j; bids $b_{1j}, \ldots, b_{mj}$.

Ensure: Serviced users S_j; cost shares $p_{1j}, \ldots, p_{mj}$

$S_j \leftarrow \{1, \ldots, m\}$ /* the set of serviced users */

repeat

$p \leftarrow \frac{C_j}{|S_j|}$ /* divide cost evenly */

$S_j \leftarrow \{i \mid i \in S_j, p \geq b_{ij}\}$ /* users still willing to pay */

until S_j remains unchanged, or $S_j = \emptyset$

$p_{ij} \leftarrow p$ if $i \in S_j$ /* serviced users pay same amount */

$p_{ij} \leftarrow 0$ if $i \notin S_j$. /* non-serviced users don't pay */

return $(S_j, (p_{ij})_{i=1,m})$

3.4 A Mechanism for Dynamic Collaborations

The simple offline mechanism in the previous section is insufficient for optimizations in the cloud, because cloud users change over time. In this section, we develop a new *online mechanism* for pricing cloud optimizations, which assumes users join and leave the system at any time. In general, if one applies a truthful offline mechanism to an online setting, the resulting mechanism is no longer truthful [79, pp.412]; similarly, applying an offline cost recovering mechanisms to an online setting may render it non-recovering. Our new mechanism is specifically designed for an online setting, and we prove that it is both truthful and cost recovering. We continue to restrict our discussion to additive optimizations (we drop

this assumption in the next section), and therefore, without loss of generality, we discuss the mechanism assuming a single optimization j.

The cost of an optimization has two components: an initial implementation cost (*e.g.*, building an index) and a maintenance cost (*i.e.*, cost of index storage and index maintenance). To avoid oscillations where users can afford the initial cost of implementing an optimization but not its maintenance cost, we propose an approach where the cloud computes a single, fixed cost C_j for each optimization, j. That cost captures both the initial implementation cost and the maintenance cost for some extended period of time T (*e.g.*, a month). Users are allowed to join and leave at anytime during T. However, at the end of the time-period T, the cost of the optimization is re-computed and all interested users must purchase the optimization again.

3.4.1 Add^{On} Mechanism

We divide T into time-slots numbered $1 \dots z$. These time-slots denote the smallest time interval for which a service is provided to any user. If T is a month, slots could correspond to hours, days, or weeks. The value for user i is a tuple $\theta_{ij} = (s_i, e_i, v_{ij})$, where $s_i \in 1 \dots z$ is the time when i enters the system (for *e.g.*, by opening an account) and $e_i \in 1 \dots z$ is the time when the user exits from the system, and $v_{ij}(t)$ is a function representing her value at time t. $v_{ij}(t)$ can be any arbitrary non-negative function and may be such that the user only uses the optimization for a subset of time-slots in $[s_i, e_i]$. The interpretation is the following. At each time t, if $t \in [s_i, e_i]$ and the user gets access to optimization i at time t, then she obtains a value equal to $v_{ij}(t)$; otherwise she does not obtain any value at time t. Her total value is the sum of these unit values over all time slots t. We assume that whenever $t < s_i$ or $t > e_i$ then $v_{ij}(t) = 0$. Of special interest to us is the case when $v_{ij}(t) = v_{ij}$ is a constant value throughout the interval $t \in [s_i, e_i]$.

Users bid for the optimization j, by declaring their values as $\theta_{ij} = (s_i, e_i, b_{ij})$, where $b_{ij}(t)$ is a function of time over the interval $t \in [s_i, e_i]$. Bids are collected by the cloud at each time slot $t \in [1, z]$: a bid cannot be retroactive ($s_i < t$), but users are allowed to

revise their future bids ($b_{ij}(t')$, $t' \geq t$) upwards[4]. For example, at time $t = 1$, user 1 bids $(1, 3, [10, 10, 10])$, meaning $b_{1j}(1) = b_{1j}(2) = b_{1j}(3) = 10$; at time $t = 2$ she may revise her bids as $b_{1j}(2) = 20, b_{1j}(3) = 10$. At each time slot t, the cloud needs to determine the set of serviced users $S_j(t)$, based on the current bids. When a user i leaves the system at time e_i, then she has to pay a certain amount p_{ij}.

Example 3.4.1. *Consider one optimization j, with cost $C_j = 100$, and two users with values: $\theta_{1j} = (1, 1, [101])$, $\theta_{2j} = (1, 2, [26, 26])$. Thus, user 1 obtains a value of 101 at $t = 1$ if she can access the optimization; user 2 obtains a value 26 at each of the times $t = 1, 2$, if she has access to the optimization. Consider the following naïve adaptation of the Shapley Value Mechanism to a dynamic setting. Run the mechanism at each time slot, until it decides to implement the optimization: at that point the cloud has recovered the cost, and will continue to offer the optimization for free to new users. In our example, the optimization will be performed at $t = 1$, each user pays 50, and user 2's utility is $52 - 50 = 2$. The problem is that the mechanism is not truthful: user 2 may cheat by bidding $(2, 2, [26])$, in other words she hides her value during the first time slot. Now the entire cost of the optimization is paid by user 1, at $t = 1$, and user 2 gets a free ride at $t = 2$, obtaining a utility of $26 - 0 = 26$.*

Add$^{\text{On}}$ Mechanism, shown in Mechanism 3.4.1, computes for each time slot $t \in [1, z]$ the set of serviced users $S_j(t)$, and computes the payment p_{ij} for each user i leaving at time t. It works by running a modified Shapley-Value Mechanism at each time-slot t, which we explain next.

Suppose that no users are serviced yet. Then, the regular Shapley-Value Mechanism is run at time t, on the bids $\sum_{\tau > t} b_{ij}(\tau)$. The sum is computed separately for each user and each optimization. If the outcome is not to perform the optimization, then $S_j(t) = \emptyset$ and the system tries the next time slot $t+1$. If the outcome is to perform the optimization, then the system sets $S_j(t)$ to the set of all users served at that time slot, and also continues with the next time slot $t+1$. As new bids arrive, or future bids $b_{ij}(\tau)$, $\tau > t$ are revised upwards,

[4]As a consequence, e_i can only increase.

the cost-shares for all users are re-computed and may be lowered: as a consequence, users that could not be serviced at time t may be serviced at time $t+1$. Of course, users who no longer need the optimization (because $e_i < t$) are removed from S_j. Denote the cumulative set of serviced users as $CS_j(t) = \bigcup_{\tau \leq t} S_j(\tau)$. The key modification to the Shapley-Value mechanism is to have it operate on $CS_j(t)$ rather than $S_j(t)$. This is ensured as follows: once a user is serviced at some time τ, $i \in S_j(\tau)$, all its future bid are assumed to be ∞: this ensures that the Shapley-Value Mechanism will always include i in $CS_j(t)$. Finally, whenever a user's bid expires, *i.e.*$t = e_i$, then the user's payment is computed at that time slot, by dividing C_j by the number of all serviced users $CS_j(t)$: this is precisely the payment returned by the Shapley-Value mechanism. The users actually serviced, $S_j(t)$, are the active users in $CS_j(t)$, *i.e.*$i \in CS_j(t)$ and $t \leq e_i$: the set of grant permissions (i, j) at time t is $\{(i, j) \mid i \in S_j(t)\}$. In other words, once a user i is serviced, then she is guaranteed to pay her cost-share, and this helps to service more users in the future.

Example 3.4.2. *Let's revisit Example 3.4.1, and assume the users bid truthfully* $(1, 1, [101])$ *and* $(1, 2, [26, 26])$ *respectively. At time* $t = 1$ *both users are serviced,* $S_j(1) = CS_j(1) = \{1, 2\}$. *User* 1 *leaves at this time, so she pays* $C_j/2 = 50$. *At time* $t = 2$ *user* 2 *is serviced, hence the cumulative set of serviced users is* $CS_j(2) = \{1, 2\}$. *User* 2 *leaves at this time, so she pays* $C_j/2 = 50$*: her total utility is* $52 - 50 = 2$. *Assume that user 2 is lying and bids* $(2, 2, [26])$. *Then* $CS_j(1) = \{1\}$ *and user* 1 *pays* 100 *when leaving. At time* 2*, user* 2 *is in no feasible set since the payment required of her is* 50 *(with* $CS_j(2) = \{1, 2\}$*) but it exceeds her reported value. Thus user* 2 *gets a utility of* 0 *and has reduced her utility by lying.*

Example 3.4.3. *For a more complex example, consider an optimization with cost* $C_j = 100$ *and with four users bidding* $(1, 1, [101]), (1, 3, [16, 16, 16]), (2, 2, [26]), (2, 2, [26])$. *Then* $CS_j(1) = \{1\}$, $CS_j(2) = \{1, 2, 3, 4\}$, $CS_j(3) = \{1, 2, 3, 4\}$. *Note that user* 2 *is not included in* $CS_j(1)$ *because her bid* 48 *is below* $C_j/2$. *At time* $t = 2$ *her remaining total value is only* 32*: however, since now there are four users, each users' share is* $C_j/4$ *and therefore all users are included in* $CS_j(2)$*, and in* $CS_j(3)$. *Users 1,2,3,4 leave at times* $t = 1$, $t = 3$, $t = 2$,

$t = 2$ *respectively, so they pay* $100, 25, 25, 25$.

Mechanism 3.4.1 Add$^{\mathbf{On}}$ Mechanism.

Require: Optimization j; cost C_j; bids $(s_i, e_i, b_{ij})_{i=1,m}$.

Ensure: Serviced users $(S_j(t))_{t=1,z}$; payments $(p_{ij})_{i=1,m}$

$CS_j(0) \leftarrow \emptyset \quad p_{ij} \leftarrow 0, \forall i = 1, m$

for each time slot $t = 1, z$ **do**

  for each user $i = 1, m$ **do**

    if $i \in CS_j(t-1)$ **then**

      $b'_{ij} \leftarrow \infty$ /* force user i to be serviced */

    else if $t \geq s_i$ **then**

      $b'_{ij} \leftarrow \sum_{\tau \geq t} b_{ij}(\tau)$ /* remaining value know at t */

    else

      $b'_{ij} \leftarrow 0$ /* prune users not yet seen */

    end if

  end for

  /* Update the set of serviced users */

  $(CS_j(t), (p'_{ij})_{i=1,m}) \leftarrow$ Shapley-Mech$(C_j, (b'_{ij})_{i=1,m})$

  $S_j(t) \leftarrow \{i \mid i \in CS_j(t), t \leq e_i\}$ /*service active users*/

  for $i = 1, m$ **do**

    if $e_i = t$ **then**

      $p_{ij} \leftarrow p'_{ij}$ /* user i pays when her bid expires */

    end if

  end for

end for

return $((S_j(t))_{t=1,z}, (p_{ij})_{i=1,m})$.

3.4.2 Properties

We prove that Add$^{\text{On}}$ has three important properties: (1) it is truthful, (2) it is cost recovering, and (3) it is resilient to multiple identities, which is another way of cheating.

Truthful The definition of a truthful mechanism in the dynamic setting is more subtle than in the static setting. In a static scenario, the mechanism is called truthful if for *any set of bids*, user i cannot obtain more utility by bidding $b_{ij} \neq v_{ij}$ than by bidding her true value $b_{ij} = v_{ij}$. In the dynamic case, user utilities depend not only on the other bids happening until now, but also on what will happen in the future. We assume the *model-free* [79] framework to define truthfulness in the dynamic case: it assumes that bidders have no knowledge of the future agents and their preferences. At each time t, every agent assumes their worst utility over all future bids, and they bid to maximize this worst utility [79].

Example 3.4.4. *Consider Example 3.4.3. User 2 bids* $(1, 3, [16, 16, 16])$*, thus she could obtain a value 16 at each of the three time slots* $t = 1, 2, 3$*; but she is serviced only at time slots* $t = 2, 3$*, hence her value is* $16 + 16 = 32$*. She pays* 25*, thus her utility is* $32 - 25 = 7$*. Suppose that she cheats, by overbidding* $(1, 3, [17, 17, 17])$*. Now she is serviced at all three time slots, but still pays only* 25 *(because when she leaves there are four users in* CS_j*). Thus, for the particular bids in Example 3.4.3, user 2 could improve her utility by cheating. In a model-free framework, however, users do not know the future, and they must assume the worst case scenario. In our example, the worst case utility for user* 2 *at* $t = 1$ *(when she places her bid) corresponds to the case when no new bids arrive in the future: in this case, if she overbids* ≥ 50*, she ends up paying 50, and her utility is* $48 - 50 = -2$*. If she underbids, her worst case utility is still 0. By cheating at* $t = 1$*, user 3 cannot increase her worst case utility.*

With the *model-free* notion of truthfulness [79], a dynamic mechanism is called truthful if, for each user, revealing her true preferences maximize the minimum utility she can receive, over all possible future user preferences. This definition of truthfulness reduces to the classic

definition of truthfulness for the static case (with a single time slot).

Proposition 3.4.1. *Add^{On} Mechanism is truthful.*

Proof. (Sketch) Consider a user i bidding at time t, *i.e.*, her bid is (s_i, e_i, b_{ij}) and $t \leq s_i$ (bids cannot be placed for the past). We claim that its minimum utility over all future user's preferences (at times $t+1, t+2, \ldots$) is when no new bids arrive in the future. Indeed, any new bids in the future can only decrease the payment due by user i (by increasing the set $S_j(e_i)$, hence decreasing her payment $p_{ij} = \frac{C_j}{|S_j(e_i)|}$), and can only increase her value at every future time slot $t' \leq s_i$, by including i in a set $S_j(t')$ where it was previously not included. Thus, the minimum utility for user i is when no new bids arrive after time t. But in that case, Add$^{\text{On}}$ degenerates to one round of the Shapley-Value Mechanism, run at time t, which we saw was truthful. □

Cost-recovering Intuitively, Add$^{\text{On}}$ is cost-recovering since it is like the Shapley-Value Mechanism applied to a modified game. We now prove that Add$^{\text{On}}$ is cost-recovering.

Proposition 3.4.2. *Add^{On} Mechanism is cost-recovering.*

Proof. Consider the last time slot, $t = z$, of the algorithm. Assume w.l.o.g. that $CS_j(z) \neq \emptyset$: otherwise, if $CS_j(z) = \emptyset$, then the optimization is not implemented at all during the time period $T = 1 \ldots z$, and the cost-recovering property Eq.(3.4) holds trivially. Let p'_{ij} be the payments determined by Shapley-Value Mechanism for the time slot z (see Mechanism 3.4.1): by definition, this mechanism ensures $\sum_i p'_{ij} = C_j$. Consider any user i. We claim that its real payment is $p_{ij} \geq p'_{ij}$. Indeed, if $i \notin CS_j(z)$ then $p_{ij} = p'_{ij} = 0$, otherwise $p_{ij} = C_j/|CS_j(e_i)|$ and $p'_{ij} = C_j/|CS_j(z)|$ where e_i is the time when the users' bid expires, and the claim follows from the fact that $CS_j(e_i) \subseteq CS_j(z)$. Hence, $\sum_i p_{ij} \geq \sum_i p'_{ij} = C_j$, proving the proposition. □

Multiple Identities A user could create multiple identities and place a separate bid for each identity. If at least one identity is given access to the optimization, then the user obtains

her full value (by running her queries under that identity). However, the user is responsible for paying on behalf of all identities. It turns out that a user can increase her utility this way: by creating more identities, she could help many more users to be serviced and thus decrease her total payment. For a simple example, consider an optimization whose cost is $C_j = 101$ and a user Alice whose value is $(1, 1, [101])$. Suppose there are 99 other users whose values are $(1, 1, [1])$. Of the 100 users, only Alice is serviced, because even if all the other 99 users were serviced, each payment would be $101/100 = 1.01$ which exceeds their value of 1. However, if Alice creates two identities, each bidding (say) $(1, 1, [101])$, then Add^{On} will see 101 users, and now it can service all of them. Each user pays $101/101 = 1$. Alice pays 2, once for each identity. Thus, her utility has increased from $101 - 101 = 0$ to $101 - 2 = 99$. Add^{On} does not prevent such ways of gaming the system, because they are indistinguishable from collaborations. For example, instead of cheating, Alice could ask her friend Bob (whose value is at least 1) to participate in the game, then reimburse her for her payment: this is indistinguishable from creating a fake identity. On the other hand, there is nothing wrong with that: through her action, Alice helped more users being serviced, accepting to pay slightly more than the share of the other users. We can prove that this holds in general.

Proposition 3.4.3. *Suppose a user i can increase her utility under Add^{Off} or Add^{On} by creating multiple identities $i_1, i_2, \ldots$ Then no other users' utility decreases.*

Proof. (Sketch) Consider two games, one with user i with a single account and one with user i creating k identities $i_1, \ldots, i_k$ and associated bids. Her utility can increase by creating dummy identities only if the total payment by the dummies is less than the total payment without the dummies. Let user i's payment with no dummies be p_i and the total payment of her dummies be p'_i. Since creating dummies increases i's utility $p'_i < p_i$, and the payment per dummy (which would be the payment per user as well with the dummy accounts) is $p'_i/k < p'_i < p_i$. Thus, for all users served in the game with no dummies are surely served with dummies too since the payment per user did not increase. Hence the utility of no user decreases. □

Mechanism 3.5.1 Subst$^{\text{Off}}$ Mechanism: Cost-sharing mechanism for *substitutable* optimizations for a *single* slot.

Require: Opts. J; costs $(C_j)_{j=1,n}$; bids $(b_{ij})_{i=1,m;j=1,n}$

Ensure: Alternative $a \in A$; cost shares $(p_{ij})_{i=1,m;j=1,n}$

$a \leftarrow \emptyset \quad p_{ij} \leftarrow 0, \forall i = 1, m \;\; \forall j = 1, n$

loop

for each optimization j in J **do**

/* Compute serviced users, discard payments */

$(S_j, (p'_{ij})_{i=1,m}) \leftarrow$ Shapley-Mech$(C_j, (b_{ij})_{i=1,m})$

end for

/* Find the smallest cost-share optimization */

$J^f \leftarrow \{j \in J \mid S_j \neq \emptyset\}$ /* Set of feasible opts */

if $J^f \neq \emptyset$ **then**

$j_{min} \leftarrow \arg\min_{j \in J^f} \left(C_j / |S_j|\right)$

$a \leftarrow a \cup \{j_{min}\}$ /* Perform optimization j_{min} */

for each user $i \in S_{j_{min}}$ **do**

$a \leftarrow a \cup \{(i, j_{min})\}$

$p_{ij_{min}} \leftarrow C_{j_{min}} / |S_{j_{min}}|$

$b_{ij} \leftarrow 0 \quad \forall j \in J$ /* Remove i from future loops */

end for

$C_{j_{min}} \leftarrow \infty$ /* Remove j_{min} from future loops */

else

return $(a, (p_{ij})_{i=1,m;j=1,n})$

end if

end loop

3.5 Mechanisms for Substitutable Optimizations

In this section, we relax the requirement that optimizations be independent. Indeed,

Mechanism 3.5.2 Subst$^{\mathbf{On}}$ Mechanism Cost-sharing mechanism for *substitutable* optimizations, for *multiple* slots.

Require: Opts J; costs $(C_j)_{j=1,n}$; bids $(s_i, e_i, (b_{ij})_{j=1,n})_{i=1,m}$.

Ensure: Serviced users $(S_j(t))_{t=1,z}$; payments $(p_{ij})_{i=1,m}$

$a \leftarrow \emptyset \quad p_{ij} \leftarrow 0, \forall i = 1, m$

for each time slot $t = 1, z$ **do**

  for each user $i = 1, m$ **do**

    if $\exists j \in J. (i, j) \in a$ **then**

      $b'_{ij} \leftarrow \infty$ /* force user i to be serviced */

      $b'_{ij'} \leftarrow 0 \ \ \forall j' \in J, j' \neq j$ /* force i to only use j */

    else if $t \geq s_i$ **then**

      $b'_{ij} \leftarrow \sum_{\tau \geq t} b_{ij}(\tau)$ /* remaining value know at t */

    else

      $b'_{ij} \leftarrow 0$ /* prune users not yet seen*/

    end if

  end for

  /* Update the set of serviced users */

  $(a, (p'_{ij})_{i=1,m;j=1,n}) \leftarrow \text{Subst}^{\text{Off}}(J, (C_j)_{j=1,n}, (b'_{ij})_{i=1,m;j=1,n})$

  $S_j(t) \leftarrow \{i \,|\, \exists j.(i, j) \in a, t \leq e_i\}$

  for $i = 1, m$ **do**

    if $e_i = t$ **then**

      $p_{ij} \leftarrow p'_{ij}$ /* user i pays when her bid expires */

    end if

  end for

end for

return $((S_j(t))_{j=1,n;t=1,z}, (p_{ij})_{i=1:m,j=1:n})$

when multiple optimizations (*e.g.*, indexes or materialized views) exist, the value to the user from a set of optimizations can be a complex combination of individual optimization values.

In this section, we consider the case of substitutable optimizations. Formally, each user defines a set of substitutable optimizations $J_i \subseteq J$ such that $\forall j, k \in J_i : v_{ij} = v_{ik} = v_i > 0$. Additionally, given an outcome a, $V_i(a) = v_i$ if $\exists j \in J_i : (i, j) \in a$ and $V_i(a) = 0$ otherwise. In comparison to the substitutable valuation, the valuation function that we previously used was the *sum*: $V_i(a) = \sum_{(i,j)\in a} v_{ij}$. With substitutable valuations, a user bid takes the form $\theta_i = (J_i, v_i)$, where J_i is the set of substitutable optimizations and v_i is the user value if she is granted access to at least one optimization in J_i.

Substitutable optimizations capture the case where implementing any optimization from a set (*e.g.*, indexes, materialized views, or replication) can speed-up a workload by a similar amount. The user does not have any preference as to which optimization is responsible for the speed-up. However, she gets no added value from multiple optimizations being implemented at the same time either because the optimizations cannot be used together (*e.g.*, a materialized view may remove the need for a specific index) or because she gets no added value from further performance gains.

3.5.1 Subst^{Off} Mechanism

We first consider the static game where all users bid and use the system for the same time-period.

Example 3.5.1. *Consider a set of three optimizations with costs* $C_1 = 60$, $C_2 = 180$, *and* $C_3 = 100$. *The bid* $(\{1, 2\}, 100)$ *indicates that a user has value* 100 *if she is granted access to either optimization 1 or 2. Three other example bids include* $(\{3\}, 101)$, $(\{1, 2, 3\}, 60)$, *and* $(\{2\}, 70)$, *for users* $\{2, 3, 4\}$, *respectively.*

The challenge with substitutable optimizations is that users may define overlapping but different sets of optimizations as in Example 3.5.1. Users also have several new ways of cheating. In addition to lying about their value v_i, they may lie about the optimizations they want by either bidding for ones that do not benefit them or by not bidding for the ones that do. They can also emulate multiple users with different optimization sets. Our

mechanisms are truthful for all but for the case of cheating with multiple dummy users.

To address this challenge, we develop the mechanism shown in Mechanism 3.5.1. The Subst$^{\text{Off}}$ Mechanism first runs the Shapley Value mechanism for each optimization independently. It then selects the optimization that yields the lowest cost-share for a non-empty set of serviced users. These users will be serviced by that optimization at the computed cost-share. The mechanism then repeats the analysis for the remaining users and optimizations.

Example 3.5.2. *Consider example 3.5.1. Subst^{Off} first identifies optimization* 1 *as having the lowest cost-share with* $S_1 = \{1, 3\}$ *and cost-share* $\frac{60}{2} = 30$*, and thus implements optimization* 1 *and services users* 1 *and* 3*. Next, Subst^{Off} considers the remaining users* $\{2, 4\}$ *and the remaining optimizations* $\{2, 3\}$*. For these optimizations,* $S_2 = \emptyset$ *while* $S_3 = \{2\}$*. Optimization* 3 *is thus implemented and user* 2 *is given access to it. User* 4 *gets access to no optimization.*

We now prove that Subst$^{\text{Off}}$ is both truthful and cost-recovering.

Proposition 3.5.1. *The Subst^{Off} Mechanism is cost-recovering (budget-balanced).*

Proof. This property follows directly from the mechanism construction: When the mechanism implements an optimization, it splits the optimization cost across all serviced users. It then discards the serviced users from consideration for further optimizations. □

Proposition 3.5.2. *Subst^{Off} is truthful.*

Proof. We prove by induction on $|J|$. For any user i the following holds.

Base case: When $|J| = 1$, the mechanism is identical to Add$^{\text{Off}}$ Mechanism which is truthful for single optimizations (refer to Section 3.3.2).

Inductive case: Now, assume that the mechanism is truthful for $|J| \leq n$. Consider $|J'| = n + 1$. Let j be the optimization found by Mechanism 3.5.1 with the minimum cost-per-user, p_{ij}, with feasible user set S_j. If $i \in S_j$, increasing her bid $b_{ij} > v_{ij}$ will not reduce p_{ij} (and hence not change her utility). Similarly, reducing $b_{ij} < v_{ij}$ leads to either

the same value for p_{ij} (so her utility is unchanged) or increases p_{ij} enough to lead to the denial of optimization j to i and a zero utility. User i might still get serviced a higher-priced optimization but that would also reduce i's utility. If $i \notin S_j$, then

1. the minimum price to access j is more than i's value for j and hence increasing her bid to obtain the optimization would lead to negative utility.

2. $v_{ij} = 0$: in this case, i might want to increase p_{ij} for some j with the hope that j will not get implemented and hence some users from S_j might contribute to another optimization j' that i is interested in. However, bidding any non-negative value for j can only decrease p_{ij} further and increasing the bid for an optimization $j' \neq j$ has no impact on p_{ij}. If i belongs to the feasible set of optimization j' then increasing her bid will not reduce $p_{ij'}$ below p_{ij} since increasing the bid beyond $p_{ij'}$ does not decrease $p_{ij'}$. Reduce $b_{ij'}$ below $p_{ij'}$ will remove i from j' service set and render a utility of zero from j'. If i does not belong to the feasible set of any optimization j' that it is interested in it implies that the minimum price to access j' is more that i's value for j' and increasing her bid to obtain the optimization would lead to negative utility.

Thus, the optimization j with the minimum cost per user is implemented and $I \leftarrow I \setminus S_j$ and $J \leftarrow J' \setminus \{j\}$.

By induction, the mechanism will be utility-maximizing, and hence truthful, for the smaller set of users and the smaller set of optimizations. □

Example 3.5.3. *Consider example 3.5.2. If user* 3*, with the intent of cheating, bid any value in the range* $[30, \infty)$*, the outcome and her utility would remain unchanged. If she bid below* 30*, however, she would not be serviced by optimization* 1 *as her bid would be below the cost-share. She would not get serviced by any other optimization, either, because their cost-shares are higher than that of* 1*, which was the optimization with the lowest cost-share. Her utility would be* $(0 < 30)$*. Finally, if she, being untruthful, did not bid for optimization* 1*, even though it benefited her, and bid* $(\{2, 3\}, 60)$*, then both optimization* 1 *and* 2 *would tie*

for lowest cost-share at 60. *Let us assume Subst*^{Off} *would randomly choose and implement optimization* 2, *then she would be granted access to this optimization and would pay the cost-share of* 60 *achieving a strictly lower utility of* 0.

3.5.2 Subst^{On} *Mechanism*

We now consider substitutable optimizations, but in a dynamic setting where users can join and leave the system in any time-slot. Given substitutable optimizations J_i, user i bids $\omega_i = (s_i, e_i, b_i, J_i)$, with $[s_i, e_i]$ as the requested interval of service and $b_i(t)$ is the value she gets at time t.

Subst$^{\text{On}}$ Mechanism, shown in Mechanism 3.5.2, works by running Subst$^{\text{Off}}$ at each time-slot t with the residual value of all the users seen. The first time a user i is granted access to optimization j her bid for j is updated to ∞ (so that she is always in the feasible set of j), while her bids for the other optimizations are updated to 0 (so that she remains serviced only by optimization j).

Example 3.5.4. *Consider the following bids for three optimizations,* $\{1, 2, 3\}$, *with costs* $C_1 = 60, C_2 = 100, C_3 = 50$. *User* 1 *bids* $(1, 2, 100, \{1, 2\})$ *that is interpreted as follows: she values any optimization in* $\{1, 2\}$ *at* 100 *for the time-slots* $[1, 2]$. *User* 2 *bids* $(2, 3, 100, \{1, 2, 3\})$ *and user* 3 *bids* $(3, 3, 100, \{3\})$. *At* $t = 1$, *Subst*^{On} *runs Subst*^{Off} *with user* 1 *(the only user at that time) and ends up implementing optimization* 1, *with payment of* 60. *Then, Subst*^{On} *updates user* 1*'s bid to optimization* $\{1\}$ *valued at* ∞. *At time* $t = 2$, *Subst*^{On} *runs Subst*^{Off} *with users* $\{1, 2\}$ *and ends up granting user* 2 *access to optimization* 1 *with the new payments for both users being* $60/2 = 30$. *User* 1 *leaves after paying* 30, *while user* 2*'s bids are updated to optimization* $\{1\}$ *valued at* ∞. *At time* $t = 3$, *Subst*^{On} *again executes Subst*^{Off} *with all three users (although user* 1 *left, she is included while invoking Subst*^{Off}, *to compute the proper cost-share for user* 2*), and ends up implementing optimization* 3, *but only for user* 3, *at a payment of* 50. *User* 2 *is not serviced optimization* 3 *since she is already using optimization* 1 *and Subst*^{On} *does not allow her to switch to a new optimization. The*

system ends with user 2 *paying* 30 *and user* 3 *paying* 50. *The inability to switch is crucial for truthfulness: Otherwise, a new user, say user* 4, *who prefers optimization* $\{1, 3\}$, *arriving at time* $t = 3$, *might only bid for optimization* 3 *hoping that user* 2 *switches to optimization* 3. *With the switch each would pay* $50/3 = 16.7$, *while without the switch user* 2 *pays* $60/2 = 30$ *(as before) and users* $\{3, 4\}$ *pay* $50/2 = 25$.

We proof that Subst$^{\text{On}}$ is both truthful and cost-recovering.

Proposition 3.5.3. *The Subst^{On} Mechanism is cost-recovering (budget-balanced).*

Proof. This property follows directly from the mechanism construction: When the mechanism implements an optimization, it splits the optimization cost across all serviced users. It then discards the serviced users from consideration for further optimizations. □

Proposition 3.5.4. *The Subst^{On} Mechanism is truthful.*

Proof. (Sketch) We claim that for all known users at time t their minimum utility over all future users' preference (at times $t + 1, t + 2, \ldots$) is when no bids arrive in the future. Indeed, any new future bids can only reduce the payment due by user i by increasing the set $S_j(e_i)$, hence decreasing her payment $p_{ij} = C_j/|S_j(e_i)|$. It can also only increase her value at every future time slot $t' \leq s_i$, by including i in a set $S_j(t')$ where it was previously not included. Thus, the minimum utility for user i is when no new bid arrive after time t. In that case, however, Subst$^{\text{On}}$ reduces to Subst$^{\text{Off}}$, executed at time t, which is truthful by Proposition 3.5.2. □

Multiple Identities Unlike for Add$^{\text{Off}}$ and Add$^{\text{On}}$ Mechanisms, for Subst$^{\text{Off}}$ and Subst$^{\text{On}}$ Mechanisms dummy users can increase their own utility at the expense of other users. Consider users $\{1, 2, 3\}$ with single-slot bids $(\{1\}, 5)$, $(\{1, 2\}, 2.51)$, and $(\{2\}, 7)$ for optimizations $\{1, 2\}$ with costs $C_1 = 6$ and $C_2 = 5$. With no dummy users, optimization 2 is implemented with a payment of 2.5 and utilities of 0.01 for user 2 and 4.5 for user 3. If user 1 creates two identities $1'$ and $1''$ that make a bid of 2.5 each for optimization 1, then both optimizations

are implemented with optimization 1 serving $\{1', 1'', 2\}$ with utilities of 1, 0.51, and 2 for users 1, 2, and 3 respectively. Note that user 3's utility has reduced. However, to cheat user 1 needed to know the number of other users and their bids. In practice, she is unlikely to know this information. She may try guessing, but as we now show, in the worst case, her guess can lead to a reduction in her utility. Thus, being truthful is the optimal strategy without knowing the other bids.

Proposition 3.5.5. *Constructing multiple identities can, in the worst case, lead to a reduction in utility as compared to the utility with a single identity.*

Proof. Consider user 1 who wants optimization $\{1\}$, n_{12} identical users who want any optimization amongst optimizations $\{1, 2\}$, and n_2 identical users who want optimization $\{2\}$. If optimization 1 is not implemented because the cost-per-user for optimization 2 is lower, *i.e.*, $C_1/(1 + n_{12}) > C_2/(n_{12} + n_2)$, then user 1 may create n_1 dummy bids (apart from her original bid) and cause the n_{12} users who wanted either of $\{1, 2\}$ to be serviced optimization 1 instead of optimization 2. Thus, the payment-per-user for optimization 1 would become less than that of optimization 2, *i.e.*, $C_1/(1 + n_1 + n_{12}) \leq C_2/(n_{12} + n_2)$. This leads to a total payment of $(1 + n_1)C_1/(1 + n_1 + n_{12})$ for user 1 and an increase in utility of $C_1 - (1 + n_1)C_1/(1 + n_1 + n_{12}) = C_1 n_{12}/(1 + n_1 + n_{12}) \leq C_2 n_{12}/(n_{12} + n_2)$. This gain can become arbitrarily small depending on the value of C_2 and the ratio $\frac{n_2}{n_{12}}$. Further, the minimum number of dummy users that need to be created is $n_1 \geq C_1/C_2(n_{12}+n_2) - n_{12} - 1$. Again, this value can be made arbitrarily high be increasing n_2. Suppose then that user 1 guesses a high enough value of n_1 and creates n_1 dummy users. Then, although she ends up increasing her utility in this game, in the worst case (shown below), this may cause her to pay more for optimization 1 than she would have paid had she not cheated. In the worst case, consider another game with n_0 identical users (apart from 1 and her dummies), who also bid only for optimization $\{1\}$. Without cheating user 1 pays $C_1/(1 + n_0)$, while with cheating she pays $(1+n_1)C_1/(1+n_0+n_1)$. The extra money she pays is $C_1 \frac{n_1}{2(2+n_1)}$. This value can be made arbitrarily close to $C_1/2$ as n_1 increases. Thus, user 1 may end up paying extra

while cheating with multiple dummy accounts. Since we assume that she has no *a priori* information about the bids by the other players and their numbers, bidding with multiple identities is no better, in the worst case, than bidding with a single user, and vice versa. □

3.6 Evaluation

Our mechanisms guarantee truthfulness and cost-recovery, but they do not optimize total utility. In this section, we empirically evaluate the total utility that our solutions provide. We focus on the two online mechanisms (*i.e.*, Add^{On} Mechanism and Subst^{On} Mechanism) and compare them to the state-of-the-art regret-based approach (Section 3.6.1) [26, 56]. The experiments consist of both the motivating use-case (Section 3.1) and simulated scenario (Sections 3.6.3 through 3.6.6).

3.6.1 Regret-Based Amortization

Kantere, Dash, *et al.* [26, 56] proposed a regret-based approach (called Regret, henceforth) to select optimizations. They developed a detailed economy of the cloud and considered detailed query plans for computing regret. In this paper, we abstract and evaluate the performance of the core regret-based approach without the surrounding economy or query plan details. We briefly describe the algorithm.

The regret for an optimization j at time t, termed $R_j(t)$, is defined as the total *value* that would have been realized, over all users, until time t (and excluding it) had j been implemented and the users serviced. Formally, $R_j(t) = \sum_{\tau<t} \sum_{i \in I} v_{ij}(\tau)$, where I is the set of all users and v_{ij} is the value that user i has for optimization j. The policy we adopt as to when to implement the optimization is the greedy approach [79] where the optimization is implemented at time slot t when $c_j \leq R_j(t)$. For the case of substitutable optimizations, once an optimization j is implemented for a user i, user i stops benefiting from and contributing to the regret of other optimizations in $J \setminus \{j\}$.

To recoup the cost c_j, each future user who gets access to optimization j pays a price p_j until the cost is amortized. Let t_r^j be the time at which the Regret algorithm implements

j. To fix p_j, we look at the remaining value in the game assuming perfect knowledge of future users and their values. We then choose a p_j that minimizes the cloud loss. Let $I_j(p, t_r^j) = \{i \mid \sum_{t>t_r^j} v_{ij}(t) \geq p\}$. Then $p_j = \arg\min_p \max\{c - p \times |I_j(p, t_r^j)|, 0\}$. (In case of multiple choices for p_j we choose the one with the lowest magnitude since that maximizes the user utilities.) Thus, our price point is the optimal choice to minimize the cloud loss. It upper-bounds how well Regret would work in practice. The total social utility (*a.k.a.*total utility) for Regret is defined the same way as for the mechanisms: the total value realized by the users for the slots they are serviced minus the implemented optimizations' costs. The cloud balance is the costs of the optimizations minus the total payments by the users. A negative balance means that the cloud incurs a loss.

Our approach thus computes regret the same way as Kantere, Dash, *et al.* [56, 26] except that, in their approach, users assign values to individual queries. Our approach aggregates this information and assigns values to workloads spanning over larger ranges of time.

3.6.2 Evaluation on the Motivating Use-Case

The workload from the motivating use-case in Section 3.1 traces the evolution of halos over 27 snapshots of a universe simulation. Each astronomer starts with a subset of halos, γ, in the final snapshot at t_{27} and, for each halo $g \in \gamma$, she (a) computes the halos in each previous snapshot contributing the most *particles* to g, and (b) recursively computes a chain of halos $h_1^g, \ldots, h_{26}^g, h_{27}^g = g$ such that h_t^g contributes the most *mass* to the halo h_{t+1}^g in the next snapshot. Our optimizations materialize the following relation for each snapshot: `(particleID, haloID)` to speed-up the queries.

We experiment with six users with differing workloads: two workloads (provided to us by the astronomers) use all 27 snapshots to determine the ancestors of halo sets γ_1 and γ_2, respectively. Based on the astronomers' feedback, we construct two extra users for each of γ_1 and γ_2: one user examines every 2^{nd} snapshot and the other every 4^{th} snapshot. This simulates faster, exploratory studies of the data.

In this experiment, we measure the total utility (Sec. 3.2) for the Add$^{\text{On}}$ and Regret

approaches.

We take each optimization's cost to be the cost of storing the materialized view. To compute this cost, we use the dollar amount per GB of storage of a yearly subscription of the Amazon EC2 High-Memory Extra Large Instance. This yields an average cost of $2.31 per optimization.[5]

We take the money saved by completing queries early to be the value of an optimization (one has to pay Amazon for each hour of use in addition to the subscription). For the six users, the runtime of their workload without any optimizations is $81, 36, 16, 83, 44$ and 17 min. Materializing the view on the last snapshot saves $44, 18, 8, 39, 23$, and 9 min which corresponds to monetary savings of $18, 7, 3, 16, 9$, and 4 cents for a single execution of the workloads. The other optimizations reduce runtime by 2.5 min each for a saving of 1 cent. Since, the optimizations affect different queries in the workload, we take them to be additive.

We now consider a year-long time-period and assume that users use the service in multiples of a quarter (3 months). We exhaustively explore all the ways that the six users can bid for slots. For each alternative, we further vary the number of times that each user executes her workload while using the system and compute the total utility achieved by each approach. Figure 3.1 shows the average and standard deviation of the total utilities across the 10^6 alternatives. The figure shows results for low (1 workload execution/quarter) to medium (1 workload execution/day) intensity use. In these conditions, Add^{On} yields a total utility between 28% and 47% higher than Regret. Additionally, the cloud never makes a loss with Add^{On} while the loss by Regret can be up to a substantial 92% of Regret's utility.[6] The baseline cost is the total cost of executing the workloads with no optimizations. Add^{On} and Regret yield utilities of 28%-47% and 16%-40% of the base line cost, respectively. In the case of Add^{On}, since the users pay entirely for the cost, the utility is the savings for the group;

[5]We could have used other cloud storage costs. We chose this one as is was the most similar to our local machine where we obtained this experiment's space and runtime numbers.

[6]In the case of a scientific collaboration, we can also assume that one of the researchers pays the cloud to implement the optimization. She then asks the others to pay her back. That researcher is then the one to incur the loss. In this case, the utilities represent the amount saved by the group.

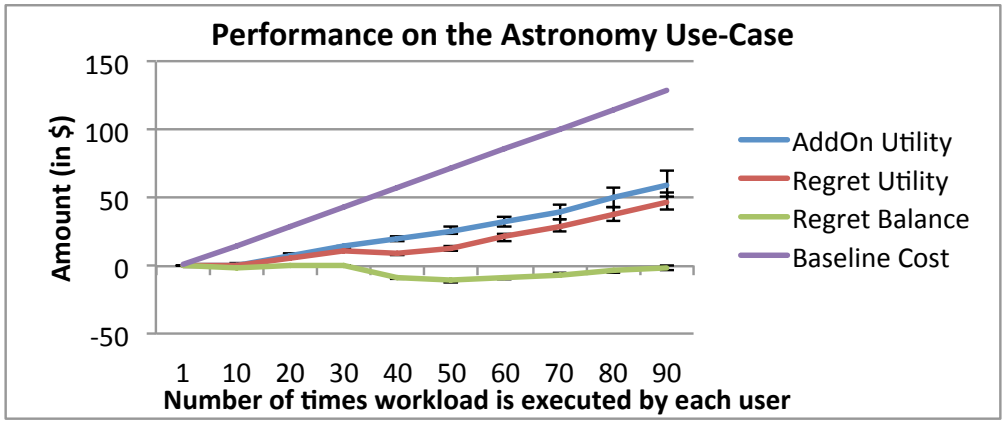


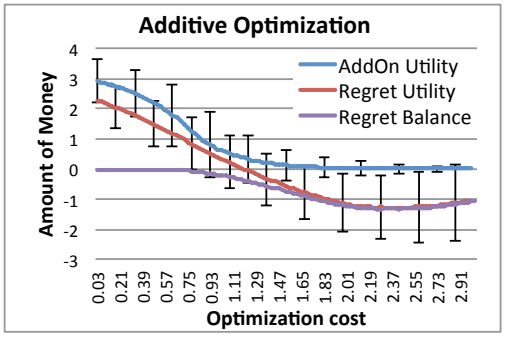
Figure 3.1: Operating *expenses* without optimization and total utility (equal to total money saved) by Add^{On} and Regret for the astronomy workload and an Amazon EC2 subscription, as users execute their workloads more frequently.

for Regret, since the users do not always pay the entire cost, the users save more than the utility, but at the expense of the cloud, who subsidizes upto 48% of their savings.

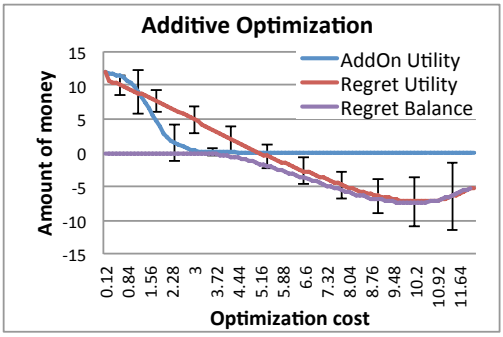
3.6.3 Collaboration Size

In this section and the following, we use a variety of simulated configurations to explore how our mechanisms and the Regret approach compare in different settings. In all cases, we measure the total utility. The goal of the approaches is to maximize this value.

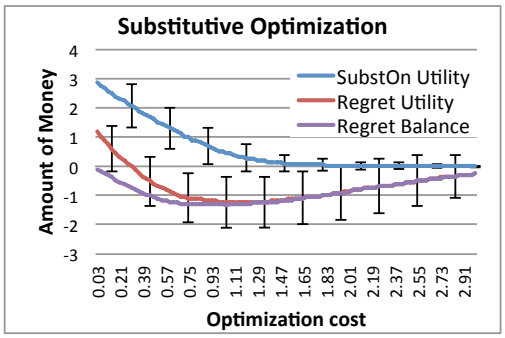
The first key parameter affecting utility is the cost of optimizations as a proportion of the user values. This ratio affects the number of users that are necessary to cover the optimizations' cost. In all simulations, we change this proportion by varying the per-optimization cost along the x-axis while keeping the average user values fixed. In this section, we measure the utility of both approaches when the total number of users available to cover the optimizations' cost is either small (small collaborations) or large (large collaborations). For both approaches, larger collaborations allow users to collectively buy costlier optimizations and yield higher utilities. We experiment with a small group of 6 users and a large one with



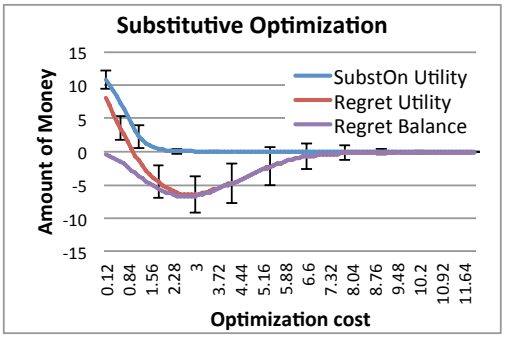
(a) Small Collaboration



(b) Large Collaboration



(c) Small Collaboration



(d) Large Collaboration

Figure 3.2: Total utility as a function of optimization cost for different collaboration sizes. Also showing regret balance (optimization costs minus user payments). Add^{On} and Subst^{On} outperform Regret for a large range of optimization costs, for both additive and substitutive optimizations, and for both low and high degree of collaboration amongst users. Further, they never incur a loss, while Regret can incur significant loss. Detailed analysis in §3.6.3.

24 users. We let users pick *one* service slot, uniformly at random, from 12 slots[7]. This gives us an expected number of users/slot of 0.5 and 2, respectively.

[7]The number 12 was chosen since 2, 3, 4, and 6 divide it perfectly and give us a larger space of parameter values to experiment with as compared to some other number like 10 or 15. The other parameter values were chosen to be multiples of 12 for ease of understanding.

Additive Optimizations

We first consider additive optimizations. We only consider one optimization since optimizations are independent.

For small collaborations, Figure 3.2a shows that as we move from cheap to costly optimizations, Regret provides good total utility, but then quickly leads to cloud loss, followed by negative total utility; while Add^{On} never leads to cloud loss or negative utilities. Negative utilities by Regret imply that the optimization was implemented but it failed to provide enough value to justify its implementation. Restricting our attention to the costs where Regret yields a positive utility, Add^{On} achieves an average total utility $1.43\times$ higher than Regret. Further, while Regret leads to cloud loss (curve "Regret Balance" in the figure) at a cost of 0.18, even for optimizations $7\times$ costlier, Add^{On} yields substantial utility (taken to be 0.3, 10% of total user value). Regret under-performs against Add^{On} for two reasons. First, for cheap optimizations that should be implemented, Regret loses user value while building up regret. Second, for costly optimizations, Regret suffers a loss and negative total utility since it implements the optimization even when the available future values is insufficient to recoup the cost.

For larger collaborations, Figure 3.2b shows that as we move to costlier optimizations, Add^{On} provides worse utility than Regret. Intuitively, Add^{On} looses some opportunities to implement optimizations because it is more cautious than Regret: To avoid losses, Add^{On} only implements an optimization when it is certain to recoup the costs given *current* information. The benefit of Regret, however, is limited: Regret soon starts losing money and leads to negative total utility. In fact, only in less than 10% of the range where Regret achieves a positive utility ($[0, 4.92]$), does it also outperform Add^{On} *and* yields no loss. Over the entire range of costs in $[0, 3.0]$ the average total utility of Add^{On} is 0.87 while that of Regret is -0.63.

For large collaborations, Add^{On} utilities sharply decrease after a point because when costs increase, the payment per user increases super-linearly, since Add^{On} prunes out users

for whom the payments are larger than the value. No users are pruned by Regret and thus it sees a linear reduction in utilities with increasing costs.

Interestingly, the range of costs for which Regret makes a loss depends on the number of users who bid. It yields a loss at a cost of 0.18 for the small group (Figure 3.2a) and 1.80 for the large one (Figure 3.2b). Thus, without knowing the future users, the cloud can not know when to avoid Regret.

Substitutive Optimizations

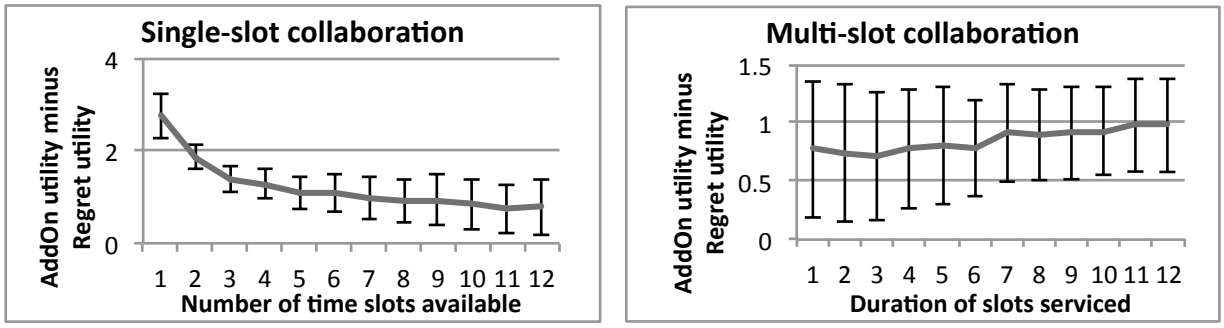
To compare Subst$^{\text{On}}$ and Regret in the case of substitutive optimizations, we consider a scenario with 12 optimizations. Each user selects 3 optimizations, uniformly at random, as the set of substitutes (Section 3.6.6 experiments with other ratios). Unlike the additive case, the costs of the 12 optimizations are sampled uniformly from $[0, 2c]$ so that c is the average optimization cost: this is to simulate that not all substitutes are equally expensive. Thus the x-axes of Figures 3.2c and 3.2d are the mean value of the optimizations.

Compared to the corresponding additive optimizations in Figures 3.2a and 3.2b, both Subst$^{\text{On}}$ and Regret achieve lower overall utility. Indeed, with substitutes, each optimization has fewer users bidding for it and, once an optimization is implemented, the serviced users no longer pay for the other optimizations. Hence, fewer optimizations are implemented and, in the case of Regret, there are fewer users over whom the costs can be amortized. In the scenarios shown, Regret yields a loss earlier than in the additive case.

When averaged over those costs for which Regret yields positive utility, Subst$^{\text{On}}$ yields 1.63× and 3× more utility than Regret for group sizes of 24 and 6, respectively.

3.6.4 Overlap in Usage

The second key parameter that affects utility is how the user values are distributed across time. We study this parameter using a small group of 6 users collaborating on a single, additive optimization. We vary the degree of user overlap and its manner. First, we repeat the experiment from Figure 3.2a while decreasing the total number of slots from 12 to 1.



(a) More collaboration on the left. x-axis is the total number of slots. The users bid for 1 slot.

(b) Less collaboration on the left. X-axis is the # of contiguous slots that each user bids for.

Figure 3.3: Add$^{\text{On}}$ vs Regret performance with varying degree of collaboration. (§3.6.4)

Figure 3.3a shows that, with fewer slots to sample from and hence with increased overlap amongst users, Add$^{\text{On}}$ generates 0.77 to 2.75 more utility, on average, than Regret. Thus, Add$^{\text{On}}$ gets 25%-91% of the total user value (3.0) as additional utility over Regret. Decreasing the number of slots, increases the probability that Add$^{\text{On}}$ finds enough value in some slot to justify implementing the optimization. In contrast, regret accumulation stays unchanged.

Next, we study what happens when user values are spread across an interval rather than being concentrated in a single time-slot. The setup in Figure 3.3b is identical to the additive case with the group size of 6 in Figure 3.2a except that instead of bidding for only one slot, users bid as $(s_i, s_i + d - 1)$, where d is the duration of the service and is varied on the x-axis. s_i is chosen uniformly at random from 12 slots. Users divide their values, chosen uniformly at random from $[0, 1)$, equally among all d time slots in their bids. The average extra value that Add$^{\text{On}}$ generates over Regret increases from 0.77 to 0.98. Indeed, as users spread their value across multiple time-slots, Add$^{\text{On}}$ becomes more likely to find a single time-slot with sufficient value to justify implementing the optimization.

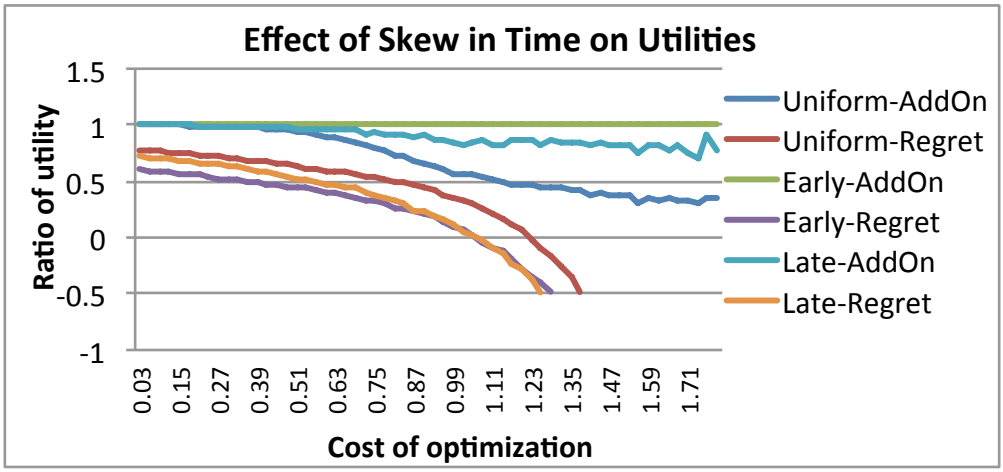


Figure 3.4: Add^{On} improves while Regret worsens with temporal skew. Ratios taken with the utility of Add^{On} with users clustered early. (Section 3.6.5)

3.6.5 Arrival Skew

We now consider the small collaboration of 6 users bidding for a single optimization, where they arrive: (a) *uniformly* at random in one of 12 slots, (b) *early* following an exponential distribution with mean 1.2[8], (c) *late* following a distribution that is $12 - t$ with t sampled exponentially with mean 1.2. Case (b) simulates datasets that become stale, while (c) simulates datasets that become popular over time. We look at the ratio of the utility in different settings to that of the utility of Add^{On} with *early* arrivals. Figure 3.4 shows that total utility by Add^{On} improves while that for Regret worsens with irregular arrivals. Add^{On} outperforms Regret substantially as user arrival becomes non-uniform (and Regret soon starts generating negative utilities). With skew, Add^{On} improves due to increased chances of finding a slot with enough value to pay for all costs. For *e.g.*, with Add^{On}, early arrivals can be 6.7× and 1.8× more efficient that uniform and late, respectively. On the other hand, Regret worsens since skew increases the chance that more regret is accumulated than

[8]With mean 1.2, the maximum starting time slot of 6 users in 1000 runs was 12 as it is in case (a).

required[9]. For *e.g.*, with Regret, at the cost of 0.54, late and uniform arrivals have 16% and 40% higher total utility than early arrivals, respectively. This points to an interesting property of the mechanism-design-based approach: the approach performs much better as non-uniformity increases.

3.6.6 Selectivity of Substitutes

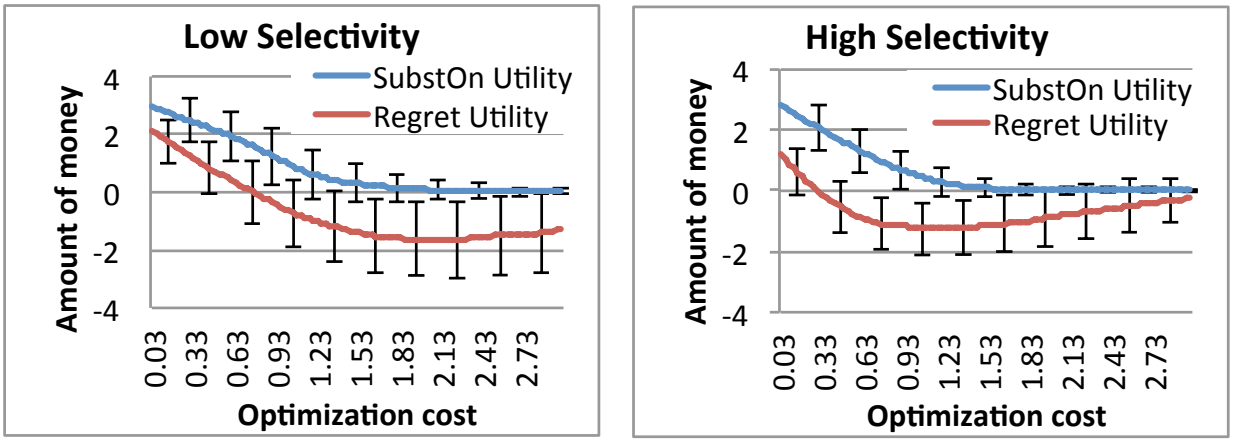
We now vary the selectivity of the substitutes, that is defined as the ratio of the number of substitutable optimizations to the total number of optimizations. Figures 3.5a and 3.5b show the total utility for selectivities of 0.75 and 0.25, where each user chooses 3 optimizations uniformly at random from 4 and 12 optimizations, respectively. The figures show that, with more selective users, absolute utilities derived by both algorithms decrease. For *e.g.*, Regret goes from a utility of 1.10 to -0.23 while Subst$^{\text{On}}$ goes from 2.38 to 1.90 for the optimization cost of 0.36 as selectivity increases. Indeed, with more selective users, the number of users per optimization decreases and more optimizations have to be be implemented to satisfy the users. For Figures 3.5a and 3.5b, Subst$^{\text{On}}$ yields an average total utility of 1.0 for optimizations that are 2.5× and 12.5× costlier than those at which Regret generates utilities of 1.0, respectively.

Summary. In summary, our mechanism-based approaches not only guarantee truthfullness and cost-recovery but also yield utility that frequently exceeds that of Regret. Our approaches work especially well in scenarios where many users derive significant value from an optimization during the same time-slot. They under-perform compared to Regret in scenarios where users value the same optimization but during non-overlapping periods.

3.7 Conclusions

We studied how a cloud data service provider should activate and price optimizations that benefit many users. We have shown how the problem can be modeled as an instance of

[9]Regret is computed after every time slot hence it increases in discrete values. The difference in regret and the optimization cost is wasted value and is smaller for uniform arrival.



(a) Each user chooses 3 uniformly random optimizations out of 4.

(b) Each user chooses 3 uniformly random optimizations out of 12.

Figure 3.5: Effect of change in selectivities of substitutable optimization on total utility. (Section 3.6.6)

cost-recovery mechanism design. We also showed how the Shapley Value mechanism solves the problem of pricing a single optimization in an offline game. We then developed a series of mechanisms that enable the pricing of either additive or substitutive optimizations in either an offline or an online game. We proved analytically that our mechanisms are truthful and cost-recovering. Through simulations, we demonstrated that our mechanisms also yield high utility compared with a regret-based state-of-the-art approach.

Chapter 4

PRICING DATA

In this chapter, we consider the problem of building applications on top of data purchased online. In particular, we focus on the challenge of keeping track of the data purchased by applications to avoid charging applications twice in the case they request the same data twice.

The most common method for selling data online is to make it available through a RESTful API [45, 31, 100, 37, 117, 120, 92, 21]. Existing APIs enable buyers to submit requests for data in the form of parameterized queries. For example, to purchase data from Twitter, one can specify keywords of interest, say a username, in the API call and Twitter returns all activity, up to an API defined limit, from the user. Typically, sellers charge buyers based on how much data they purchase. That is, the cost of an API call is the sum of the cost of the tuples returned by that call [117].

In many scenarios, buyers need to make repeated calls to the seller's API. One example is when purchased data drives an application and the use of that application determines the data that needs to be purchased. In those scenarios, buyers may inadvertently purchase the same data twice. In fact, it is hard to build applications that never purchase the same data again. We illustrate with a concrete example:

Example 4.0.1. *Bob sells data on how many people have visited a given business (examples of such services are Yelp [121] and Foursquare [41]). To do so, Bob provides an API* `checkins(lat, long, r, t)`*, where* `(lat, long)` *define the latitude and longitude of a circle's center with radius* `r`*.* `checkins` *returns the list of users who have visited businesses, after a timestamp* `t`*, that lie in the circle.*

In our example, Alice first makes an API call, $\texttt{checkins}(\texttt{x}_1, \texttt{y}_1, \texttt{r}, \texttt{t})$*, waits for some*

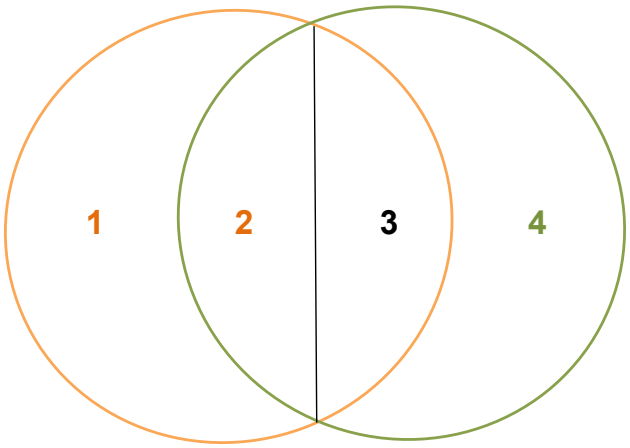


Figure 4.1: An illustration of the limitations of `REST` data APIs. Bob sells data that returns a list of users who have visted any business in a user-specified region after a certain timestamp. The data API takes a center point, a radius, and the time; it then returns the user list, who have visited a business that lies in the circle defined by the center after the specified time. In the illustration, Alice makes two API calls with identical radius and times. The first API call returns users in the regions 1, 2, and 3. Then, new users visit the region 3. Following this, Alice makes another API call with a different center point such that the resulting region partially overlaps with the region from the first API call. With today's pricing methods, Alive will pay twice for data in region 2 and the old data from region 3.

time, and makes another API call, $\texttt{checkins}(\texttt{x}_2, \texttt{y}_2, \texttt{r}, \texttt{t})$*, for a different center but the same radius. In Figure 4.1, the first call refers to the areas 1, 2, and 3; while the second refers to the areas 2, 3, and 4. Between the two calls, some businesses in area 3 recieve new visits.*

Currently, if Alice executes the two queries, she will pay twice for the data in area 2 and for the old data from area 3. Since she does not know what updates were made to the data, she must make the API call to know if the data were updated.

To alleviate this problem, Alice can change the time in the second call to `t + 1` *so that she only gets the updates. But even then, she may end up paying for redundant data if there were any checkins with time* $t > t + 1$ *in her first call. This happens when such customers*

visit a business in regions 2, 3, or 4 after time `t + 1`*, thus, being part of the answer returned to Alice during both API calls. In fact, in this example, it is impossible to avoid overpaying with the current APIs.*

However, even for static datasets for which Alice is aware that no updates will be made to the data, to avoid paying twice for the data in regions 2 and 3, Alice must make multiple calls representing many tiny circles that exactly cover the area 4 and nothing outside it[1]*.*

Today, sellers only keep track of the total amount of data purchased by a buyer but not the details of the purchased data. A primary drawback of only storing limited data about user purchases is that they put the burden on the buyer to never purchase the same data twice. However, for a buyer who makes multiple calls to the APIs where some tuples that were purchased previously are purchased again, she will pay more than once for data that she already bought through earlier queries.

Sellers may store additional information to enable pricing that accounts for prior API calls, but that can impose space and time overheads that are in the order of the data size and the number of previous API calls.

Alternatively, buyers may attempt to cache the result of API calls and modify their API calls to only ask for new data. Caching will, however, be unusable in the following cases:

- *Updates*: For datasets such as weather and traffic, the underlying data is updated over time. In such cases, it may not be possible to predict when the updates are made to the subset of data that a buyer is interested in and the only way to know of an update is to redo the call to the data API.

- *Caching restrictions*: Some APIs such as Yelp [121] prohibit all forms of caching of the their data, while others, such as Twitter [107], prohibit caching of certain forms of data while permitting caching for the other parts of the data. Thus, even if the buyer knew

[1]Only a finite number of such API calls are needed since we assume the domains for `lat, long, radius` to be finite.

that they would require a newly purchased data item in the future, they are prohibited from caching it and reusing it when the need arises.

Thus, in both circumstances above, the buyer can not avoid making multiple API calls and must incur the cost of repeat purchases of the same data.

An alternative is to store enough information about a buyer's purchase history so that the seller can figure out if a data item has already been paid for by the buyer. It may be beneficial for sellers to provide a service that only charges for data once so as to enable *price discrimination*. Although there are customers who may pay the full price of the data and not worry about paying extra, there are price-conscious customers who may not buy the dataset unless the data is available within their budget. Providing an avenue for such customers to optimize and reduce their data costs can increase revenues.

As we evaluate in §4.5, the computational overhead of storing the purchase history at the seller is significant. Moreover, this might dissusade those customers who prefer that their querying history not be stored permanently at the seller.

To address the above challenges, in this chapter, we propose lightweight modifications to data APIs to achieve the following three goals:

1. *Optimal history-aware pricing*: We provide a method to price API calls so that buyers are only charged once for data that they have purchased and that has not been updated. We refer to this as *history-aware* pricing.

2. *Constant overheads*: We provide a method to support history-aware pricing that only requires the seller to store a constant amount of state per buyer. Currently, sellers do store such information so as to keep track of a user's aggregate use of their services.

3. *Anonymity*: In addition to the above cost and performance properties, we also provide anonymity to the buyers about what data they purchase and when the purchases are made. That is, the seller need not retain any identifying information about the user that can recreate a user's query history.

The key idea behind our approach is the notion of refunds: buyers buy data as needed but have the ability to ask for refunds of data that they had already purchased before. Thus, the payment for data is conducted in two steps: the usual payment when data is received and another round where the buyer asks for refunds. While asking for refunds, the buyer *proves* to the seller that she has been charged multiple times for the same data. The proofs are constructed so as to protect against tampering by the buyer even when the buyer is not truthful or can collude with other buyers.

In this chapter, we make the following contributions:

1. In §4.3, we propose the notion of refunds as a way to provide optimal, history-aware pricing for data APIs. We describe the construction of refunds for a single-buyer setting with no updates and prove properties about the correctness and optimality of such a system.
2. In §4.4, we propose a generic and extensible framework to support refunds. We then show how it can be extended to accommodate multiple buyers, updates, and optimizations to reduce the computational and communication overheads of using refunds.
3. In §4.5, we evaluate empirically and compare the refund-based approaches to approaches that store user history at the server as well as approaches that do not provide optimal pricing. We show that even for workloads that make 100 API calls, significant cost savings, from 10× to 99×, can be obtained through the use of refunds, compared to history-agnostic pricing. The associated performance overheads, compared to history-agnostic pricing, are no larger than 2× in the best case (when no refunds need to be asked) and 5× in the worst case (when the entire query is repeated). We find that the optimizations we develop in §4.4 cut overheads by a factor of 12× for group coupons as compared to individual tuple-based coupons.

We first define our problem setting and describe two algorithms, that are not based on refunds, to manage the pricing of API calls. In §4.5, we compare these algorithms against the refund-based pricing framework.

4.1 Problem Description

We first define the pricing functions, our settings and the attack model for the buyer. In our setting, Alice runs an application that acquires data from Bob, the data seller. Bob charges Alice separately for each output tuple in her answer set.

We assume a single relation D with schema $R(tid, ver, A_1, \ldots, A_m)$. Here the column `tid` is a primary key and `ver` indicates the version number of the tuple. The version numbers are initialized to 0 and are incremented every time an update is made to the tuple. We treat the version numbers as a space overhead of our approach. The pricing function generates a price per output tuple; that is, for a query Q over D, there is a pricing function p that assigns a price to each output tuple $t \in Q$, and the price of the result is $\sum_{t \in Q} p(t)$. This is a common way [117] to price relational data in commercial data markets. In the rest of the chapter, we assume that all tuples have the same price (taken to be a unit of some currency), but the techniques generalize to cases with non-uniform prices.

The query Q is a conjunctive query with unions but without joins and has the following form in SQL:

```
SELECT tid, ...
FROM R
WHERE ...
```

The `WHERE` clause contains a list of predicates and the `SELECT` clause does not contain any aggregates.

When Alice issues a query Q, Bob executes it and computes its price. Informally, our goal is to price a sequence of Alice's queries $(Q_1, Q_2, \ldots, Q_m)$ so that she only pays once for each tuple that belongs to any query, irrespective of how many queries the tuple belongs to.

For any framework that provides optimal history-aware pricing, our desiderata are:

1. Minimize state at the seller.
2. Minimize processing at the seller.

3. Keep data transfer overheads low.
4. Minimize query latency overheads.

4.2 Naïve Approaches

We now look at two classes of solutions to manage pricing: the first does not provide optimal pricing, in the sense that Alice would pay multiple times for the same tuple she purchases; the other does provide optimal history-aware pricing, but does not satisfy the first two requirements of the desiderata.

The naïve way to compute the prices in our setting is through two queries: '`result = Q(D)`' followed by '`SELECT COUNT(*) FROM result`'. Bob charges the amount calculated in the second query to Alice and returns a cursor to `result`. The two queries belong to a single transaction to prevent the data from being updated between the time when the price is computed and the cursor to `Q` is returned. We call this method `CountBlock`, where 'Block' indicates that the query's cost is computed before the cursor to the query's answer is returned to Alice. Another approach is to count the query cardinality as Alice advances the cursor. We call the latter approach `CountStream`. Note that both `CountBlock` and `CountStream` store no user query history at the seller, and hence they will charge Alice for each tuple that is returned, even if the tuple was purchased by Alice in a previous query.

Another approach is for Bob to track the tuples purchased by Alice. Bob stores a bit for each tuple from relation `R` and does so for each user. Whenever Alice buys a tuple, the associated bit is set; while, whenever Bob updates a tuple, the corresponding bit is cleared. Algorithm 4 captures the details of this approach. As before, the above steps are encapsulated in a single transaction to prevent updates to the data between the time when the price is computed and when the cursor to `Q` is returned. We refer to this approach as `HistoryPricing`. One drawback of this approach is that the seller must provide durable storage for user history. Another drawback is that buyer purchases can no longer be anonymous.

Algorithm 4: HistoryPricing

Input : Query `Q` and user id `u`.

Output: Maintains the total cost of data purchased by Alice and computes the query answer.

```
begin
    // costs(uid, cost) stores the cost of all of user uid's purchased tuples.
    // history(uid, history) stores a bit vector, for user uid, to remember their purchased tuples.
    cost ← SELECT cost FROM costs WHERE uid = u
    history ← SELECT history FROM historyStore WHERE uid = u
    tidList ← SELECT t.tid FROM (Q) AS t
    foreach id in tidList do
        if history[id] == 0 then
            cost ← cost + 1
            history[id] ← 1
    UPDATE costs AS t SET t.cost = cost WHERE t.uid = u
    UPDATE historyStore AS t SET t.history = history WHERE t.uid = u
    return cost, Q
```

4.3 Refunds

In this section, we propose refunds as a new mechanism to achieve optimal, history-aware pricing of a sequence of queries. With support for refunds, Alice can make multiple API calls without any modification to her queries. If she makes repeated purchases, they are identified and the extra amount she paid for the repeated purchases are refunded by Bob to Alice.

To support refunds, Bob computes additional information, called refund coupons, which he returns along with the results of Alice's queries. Bob continues to charge Alice as he normally would, without accounting for any previous queries from Alice. The coupons are designed so that if there is a common tuple `tid = id` in the result of two different queries, there is a coupon from the first query and a corresponding coupon from the second query such that Bob can inspect the two coupons to determine that they refer to the same tuple with `tid = id`. Given this, Bob knows that Alice was charged twice for `id` and he can refund the price of the tuple. It is Alice's responsibility to store the coupons, to detect repeat purchases, and to use the coupons to ask for refunds.

We now formally define the protocol to support refunds for a single seller and a single

buyer over a static database. We then generalize the protocol to multiple buyers and to support updates. In §4.4, we consider specialized optimizations to reduce the overheads of supporting refunds.

We define the protocol by the messages Alice and Bob send to each other. The protocol begins when Alice sends a query `Q` to Bob. Bob sends back two messages: `Q(D)` and `refunds(Q, D)`. Both messages are sets of tuples with the following properties:

1. The schema for `refunds(Q, D)` is `(tid, qid, digest)`, where `tid` is a tuple identifier, `qid` is a query identifier, and `digest` is the output of a hash function. The schema for `refunds` is independent of the schemas for `Q` and `D`. We call each tuple in `refunds(Q, D)` a coupon where coupon `c` is defined as

$$\mathtt{c} = (\mathtt{id}, \tau, \mathcal{H}(\mathtt{id} \oplus \tau \oplus \kappa)) \tag{4.1}$$

   Here `id` is the tuple identifier; τ is a unique identifier assigned by the server to each query such that τ is monotonically increasing; $\mathcal{H}$ is a cryptographic hash function, `SHA1` in our implementation; $\oplus$ is the XOR operation[2]; and, κ is a secret key only known to Bob. In the single-buyer protocol over static data, τ is an integer that is initialized to 0 and is incremented for each query Alice sends to Bob.

2. There is a one-to-one correspondence between tuples in `refunds(Q, D)` and tuples in `Q(D)`. That is,

$$\forall \mathtt{t} \in \mathtt{Q}(\mathtt{D}), \exists \rho \in \mathtt{refunds}(\mathtt{Q}, \mathtt{D}) : \mathtt{t}[\mathtt{tid}] = \rho[\mathtt{tid}], \text{ and}$$
$$\forall \rho \in \mathtt{refunds}(\mathtt{Q}, \mathtt{D}), \exists \mathtt{t} \in \mathtt{Q}(\mathtt{D}) : \rho[\mathtt{tid}] = \mathtt{t}[\mathtt{tid}]$$

We assume that Alice asks for full queries, that is `Q` does not project out any columns.

In case Alice gets the same tuple, with $\mathtt{tid} = \mathtt{id}$ twice, from queries $\mathtt{Q}_1$ and $\mathtt{Q}_2$, she will also get two coupons $\mathtt{c}_1$ and $\mathtt{c}_2$ such that $\mathtt{c}_1[\mathtt{tid}] = \mathtt{c}_2[\mathtt{tid}] = \mathtt{id}$. Note that we have assumed

[2]For simplicity, we assume that the different ids are integers. If not, they can be cast into a string type, such as `text` in PostgreSQL, along with the use of string concatenation in place of XOR.

that all tuples are identically priced[3]. If Alice detects repeat purchases, she can ask Bob for a refund by sending a message consisting of a pair of coupons for the same tuples. Bob verifies that the hash values of the returned coupons are the ones he previously computed and credits the refund to Alice. We call this protocol BasicRefunds. Formally, BasicRefunds is defined as follows:

1. Alice sends a refund message $\rho = \langle c_1 = (id_1, \tau_1, h_1), c_2 = (id_2, \tau_2, h_2)\rangle$.
2. Bob verifies the following: (a) $id_1 = id_2$, (b) $\tau_1 < \tau_2$, and (c) $\forall i \in \{1, 2\} : h_i = \mathcal{H}(id_i \oplus \tau_i \oplus \kappa)$.

Intuitively, the refund message ρ asks a refund for tuple $id = id_1 = id_2$ purchased for a query with $qid = \tau_2$ using the coupon for the same tuple purchased with a previous query with $qid = \tau_1$.

We now define the criteria for *safety* and *optimality* of any refund-based pricing protocol. Let $W = (M_1, \ldots, M_{n_q+n_r})$ be a sequence of messages from Alice to Bob consisting of n_q queries and n_r refund requests, where each M_i is either a query Q or a refund request ρ. Let $T(W) = \{(t_1, n_1), \ldots, (t_m, n_m)\}$ be the set of all tuples purchased by Alice over the n_q (possibly different) queries in W along with their counts. Given that $p : tid \to \mathbb{R}$ is the function that assigns prices to tuples, we denote by $P(W)$ the amount Alice pays for the queries in W:

$$P(W) = \sum_{Q \in W} \sum_{t \in Q(D)} p(t[tid]) = \sum_{(t,i) \in T(W)} i * p(t[tid]) \tag{4.2}$$

Similarly, $R(W)$ denotes the amount Bob refunds to Alice after processing W:

$$R(W) = \sum_{\rho \in W} p(\rho[tid]) \tag{4.3}$$

The two properties are now defined as:

[3]For non-uniform prices, use the coupon $\mathtt{c} = (\mathtt{id}, \mathtt{p}, \mathcal{H}(\mathtt{id} \oplus \mathtt{p} \oplus \kappa))$, where $\mathtt{p}$ is the tuple's price.

Safety A refund protocol is safe if Alice must pay at least once for each tuple she has purchased. Formally,

$$\forall W : P(W) - R(W) \geq \sum_{(t,n)\in T(W)} p(t[tid]) \tag{4.4}$$

Optimality A refund protocol is optimal if there is a way to ask for refunds so that Alice never overpays. Formally, if $Q_1, \ldots, Q_{n_q}$ are the queries in W and ρ'_i are refunds, then,

$$\forall W \, \exists W' = (Q_1, \ldots, Q_{n_q}, \rho'_1, \ldots, \rho'_{n_{r'}}) : P(W') - R(W') = \sum_{(t,n)\in T(W)} p(t[tid]) \tag{4.5}$$

That is, given the queries in a message sequence W, it is always possible to request refunds to obtain the maximum possible safe refund.

Before we analyze the safety and optimality of the BasicRefunds, we note that Alice only controls three aspects of the refund protocol: when she asks for refunds, the number of refund messages, and the coupons she uses for her refund messages. Note that she can not forge or create coupons of her own since $\mathcal{H}$ is a cryptographic hash and the secret key κ is only known to Bob. Thus, any hash value that Alice may try to forge will differ from the one Bob will compute when he verifies a refund request.

Lemma 4.3.1. *BasicRefunds is optimal.*

Proof. We use induction on the number of queries in W. The base case is of an empty sequence. It is easy to see that in this case no refunds are required. For the inductive case, we assume that for $i-1$ queries $(Q_1, \ldots, Q_{i-1})$, there is a sequence, W_{i-1}, of queries and refunds, that computes the optimal refund. For a new query Q_i, let the refund messages be $(\rho_{i1}, \ldots \rho_{ik})$ where each ρ_{ij} is a refund for a tuple t that has been purchased before. Refund ρ_{ij} is constructed by taking the coupon for t, received with query Q_i, and any coupon for the same tuple id $tid = t[tid]$ received with a previous purchase. Then the sequence is $W_i = W_{i-1} \cdot Q_i \cdot \rho_{i1} \cdot \ldots \cdot \rho_{ik}$ is optimal. Let $T_{new} = \{t \in Q_i(D) \wedge (t,n) \notin T(W_{i-1})\}$ and $T_{old} = \{t \in Q_i(D) \wedge (t,n) \in T(W_{i-1})\}$. Then,

$$
\begin{aligned}
P(W_i) - R(W_i) &= P(W_{i-1}) + P(Q_i) - R(W_{i-1}) - \sum_{j=1}^{k} R(\rho_{ij}) \\
&= P(W_{i-1}) - R(W_{i-1}) + \sum_{t \in T_{new}} p(t[tid]) + \sum_{t \in T_{old}} p(t[tid]) - \sum_{j=1}^{k} R(\rho_{ij}) \\
&= P(W_{i-1}) - R(W_{i-1}) + \sum_{t \in T_{new}} p(t[tid]) + \sum_{t \in T_{old}} p(t[tid]) - \sum_{t \in T_{old}} p(t[tid]) \\
&= P(W_{i-1}) - R(W_{i-1}) + \sum_{t \in T_{new}} p(t[tid]) \\
&= \sum_{(t,n) \in T(W_{i-1})} p(t[tid]) + \sum_{t \in T_{new}} p(t[tid]) \\
&= \sum_{(t,n) \in T(W_i)} p(t[tid])
\end{aligned}
$$

Hence, W_i is optimal. □

BasicRefunds is not safe, though. Given any non-empty sequence of messages W, W can repeat a non-empty query q, and repeatedly ask for refunds of a single tuple. That is, if $\langle c_1, c_2 \rangle$ is a legitimate refund request, Alice keeps sending the request multiple times and can thus get more as refunds than the cost of the data itself.

To handle this case, we modify BasicRefunds to MonotoneRefunds. MonotoneRefunds is both safe and optimal. To implement the protocol, Bob maintains an expected query id τ_{exp} for refunds by Alice. This is initialized to 0 when Alice registers with Bob. The protocol is as follows:

1. Alice sends a ⟨ `BEGIN REFUND` τ ⟩ message. Here τ is a query id.
2. Alice sends one or more refund messages. Each refund message $\rho = \langle c_1 = (id, \tau_1, h_1), c_2 = (id, \tau, h_2) \rangle$ uses the same query id τ for the second coupon as the τ specified in the ⟨ `BEGIN REFUND` τ ⟩ message.
3. Alice sends a ⟨ `END REFUND` τ ⟩ message.
4. Apart from checking that the digest of the message is equal to the computed hash value as in BasicRefunds, Bob also checks that (a) there is only one refund message for each tuple with $tid = id$, (b) the query id of all second coupons, τ are identical and equal to the τ is the ⟨ `BEGIN REFUND` τ ⟩ message, and (c) $\tau \geq \tau_{exp}$.

5. If any of the conditions are not met, all the coupons in the `BEING ...END` block are rejected. Otherwise, Bob credits the total refund to Alice and updates τ_{exp} to $\tau + 1$.

To check the uniqueness of refund messages in Step 4, Bob can use a hash table. To directly check the uniqueness within a DBMS, Bob can also store the refunds in a temporary table, `tempRefunds`, and run: `SELECT 1 FROM tempRefunds GROUP BY tid HAVING COUNT(*) > 1`. A non-empty answer indicates a repeated refund.

Lemma 4.3.2. *MonotoneRefunds is optimal.*

Proof. If τ_{latest} is the latest query id whose coupons have not been used for refunds, then $\tau_{exp} \leq \tau_{latest}$. This is because all refunds issued in W must have a query id $\tau \leq \tau_{latest} - 1$ and hence, $\tau_{exp} \leq \tau_{latest}$ by definition. Given this, the construction of the refunds in the proof for Lemma 4.3.1 is also valid for MonotoneRefunds and hence, it is optimal. □

We prove a stronger safety property about individual tuples that implies our original safety definition for queries.

Lemma 4.3.3. *For each tuple t, let $k \geq 1$ be the number of queries by Alice that contain t, and let r be the number of valid refund messages that request a refund for t, then, MonotoneRefunds ensures that $k - r \geq 1$ at all times.*

Proof. Let the tuple be t. We prove the safety by induction on the length of W. The base case is trivially true when the first query that includes t is executed. Note that with a single query including t, valid refund messages can not be constructed, since the two coupons in the refunds must have different query ids τ. Thus $k = 1$ and $r = 0$ and the base case is satisfied.

For the inductive case, given a message sequence W_{n-1} of length $n-1$ with $k_{n-1} = k \geq 1$ queries containing t and $r_{n-1} = r - 1$ refunds, such that $k_{n-1} - r_{n-1} \geq 1$, we consider M_n, the n^{th} message. There are four cases:

1. M_n is a query Q. If it returns the tuple t, then, $k_n = k_{n-1} + 1$, else $k_n = k_{n-1}$. Since there are no refunds, $r_n = r_{n-1}$. Thus, $k_n - r_n \geq 1$.
2. M_n is a `BEGIN REFUND` message. In this case, $k_n = k_{n-1}$ and $r_n = r_{n-1}$ and thus, $k_n - r_n \geq 1$.
3. M_n is a valid refund message $\rho = \langle c_1 = (t', \tau, h_1), c_2 = (t', \tau, h_2)\rangle$. If M_n is not a valid message for t, that is $t' \neq t$, neither k nor r change. Otherwise, by the induction hypothesis: $k - (r-1) \geq 1$. Thus, $k \geq r$.

   (1) If $k \geq r + 1$, then the ρ makes $r_n = r_{n-1} + 1 = r$ and $k_n - r_n = k - r \geq 1$ by assumption.

   (2) If $k = r$, then consider the $r - 1$ previous refunds. They must use coupons from r *distinct* query ids. This is because the first refund uses two distinct query ids (by the construction of coupons) and all $r - 1$ refunds use their second coupons from $r - 1$ different queries, since only one refund coupon for a tuple is allowed in a `BEGIN REFUND ...END REFUND` block and τ_{exp} is incremented after each valid `END REFUND`. Thus, the $r - 1$ previous refunds have used coupons from r queries. Since $k = r$, the expected query id in the refund M_n must be at least one more than the query id of the k^{th} query that contains t. But since no unused coupon for t exists, the refund M_n is not a valid coupon. This is a contradiction.

   Thus, the $k_n - r_n = k - r \geq 1$ holds.
4. M_n is a `END REFUND` message. In this case, $k_n = k_{n-1}$ and $r_n = r_{n-1}$ and thus, $k_n - r_n \geq 1$.

Thus, MonotoneRefunds is safe for tuple t. □

Lemma 4.3.3 implies Definition 4.4. Givne that $I(t, \rho)$ is an indicator variable that gives

1 if $\rho[tid] = t[tid]$ (ρ is a valid refund for tuple t) and 0 otherwise,

$$
\begin{aligned}
P(W) - R(W) &= \sum_{(t,k)\in T(W)} k * p(t[tid]) - \sum_{\rho\in W} p(\rho[tid]) && \text{(Defs 4.2, 4.3)} \\
&= \sum_{(t,k)\in T(W)} k * p(t[tid]) - \sum_{(t,n)\in T(W)} \sum_{\rho\in W} I(t,\rho) * p(t[tid]) \\
&= \sum_{(t,k)\in T(W)} k * p(t[tid]) - \sum_{(t,n)\in T(W)} p(t[tid]) \sum_{\rho\in W} I(t,\rho) \\
&= \sum_{(t,k)\in T(W)} k * p(t[tid]) - \sum_{(t,n)\in T(W)} r * p(t[tid]) \\
&= \sum_{(t,k)\in T(W)} (k-r) * p(t[tid]) \\
&\geq \sum_{(t,k)\in T(W)} p(t[tid]) && \text{(Lemma 4.3.3)}
\end{aligned}
$$

Thus, MonotoneRefunds is both optimal and safe.

4.4 Extensions and Optimizations

We now consider extensions and performance optimizations that generalize the protocols to more realistic settings.

4.4.1 Extensions

In the protocols described in the previous section, the safety and optimality proofs continue to hold as long as the tuple ids are such that different tuples have different ids and identical tuples have the same id, irrespective of the query to which the tuple belongs. This observation allows us to easily extend the protocols to support more than one user and handle updates.

Multiple Buyers If there is more than one buyer, we change the tuple identifiers to also incorporate the user id. That is, the new tuple id is (id, uid) where id is the tuple's id (as in the single-buyer protocols) and uid is a unique id assigned to each user. The coupons thus

look as follows:

$$\mathtt{c} = ((\mathtt{id}, \mathtt{uid}), \tau, \mathcal{H}(\mathtt{id} \oplus \mathtt{uid} \oplus \tau \oplus \kappa))$$

With the updated construction for the coupons, different users will be assigned different tuple ids for the same tuple, while identical tuples for a user will continue to be assigned identical tuple ids. Thus, a buyer can not use refund coupons from another buyer, but can continue to use her own coupons as in the single-buyer setting.

Updates We can also support updates by modifying the tuple ids. This is applicable when updates to a tuple are priced as if the update is a new tuple. Thus, if Alice purchases tuple t_1 in her first query, then purchases t_1 again in her second query, followed by an update to t_1, denoted now by t_2, followed by another purchase of t_2, then, she should be charged for t_1 in her first query, then refunded in the second, and eventually charged only once more for t_2.

To support updates, Bob maintains a version number, v, for each tuple that is incremented after each update. This version number is now included in the tuple id used for constructing the refund coupons:

$$\mathtt{c} = ((\mathtt{id}, \mathtt{uid}, \mathtt{v}), \tau, \mathcal{H}(\mathtt{id} \oplus \mathtt{uid} \oplus \mathtt{v} \oplus \tau \oplus \kappa))$$

Thus, only identical versions of a tuple have the same tuple id. Version numbers impose a storage overhead but they are useful for other purposes and are maintained by many systems by default. For example, the SDSS [99] adds version numbers to their data releases and SciDB [98] provides a no-overwrite storage system with versioning. So, in many applications, versions already exist.

4.4.2 Group Coupons

MonotoneRefunds, described in §4.3, only computes one coupon per tuple. This leads to a large number of refund messages, each of which is an API call to Bob, when asking for refunds.

As Figure 4.5 shows, the overhead of processing refunds can be an order of magnitude larger than the query time.

To reduce this overhead, we generalize coupons to allow Bob to group coupons that can be used to refund a group of purchased tuples with a single coupon. With group refunds, Bob sends back both the coupon for individual tuples, which are represented as groups of cardinality 1, as well as group coupons for tuple groups of his choosing.

The key idea to construct group coupons is to make a unique group id (instead of a tuple id) such that no two groups with different tuples (and with possibly different versions) have the same group id and all groups with identical tuples have the same group id. Bob must provide a way to compute such group ids and also provide a function, $contains : id, gid \rightarrow \{\texttt{true}, \texttt{false}\}$, that returns `true` if a tuple with tuple id id belongs to the group with id gid. The group coupon is constructed as follows:

$$\texttt{c} = ((\texttt{gid}, \texttt{uid}, \texttt{gv}), \tau, \mathcal{H}(\texttt{gid} \oplus \texttt{uid} \oplus \texttt{gv} \oplus \tau \oplus \kappa))$$

Here gv is the group version number and is equal to the sum of the version numbers of the tuples that belong to the group. Another interpretation of gv is that it is the total number of updates made to tuples in the group.

For group refunds, the amount Bob refunds to Alice is the total cost of the tuples in the group. Let $I(t, \rho)$ be an indicator variable with value 1 if $contains(t[tid], \rho[gid]) = \texttt{true}$ (ρ is a valid group refund for a group containing tuple t), and 0, otherwise. Then, for a workload W, the total refunds, $R(W)$, are:

$$R(W) = \sum_{\rho \in W} \sum_{(t,n) \in T(W)} I(t, \rho) * p(t[tid]) \tag{4.6}$$

To use group refunds, we modify MonotoneRefunds to GroupRefunds by changing the test to validate a refund message $\rho = \langle c_1, c_2 \rangle$ in Step 4 as:

> Apart from checking that the digest of the message is equal to the computed hash value as in BasicRefunds, Bob also checks that (a) *at most one group coupon*

contains a tuple with tuple id id, (b) the query id of the latest coupon in the refund message, that is $c_2[qid]$, is equal to τ is the $\langle$ `BEGIN REFUND` τ $\rangle$ message, and (c) $\tau \geq \tau_{exp}$.

We emphasize that Bob may compute coupons where the same tuple may belong to more than one group coupon, but in the refund protocol, Alice can only request refunds using one group coupon for each tuple. Thus, more than one group coupon that include a common tuple can not be used simultaneously in the same refund round.

Lemma 4.4.1. *GroupRefunds is both optimal and safe.*

Proof. GroupRefunds is optimal since all the refunds for individual tuples are also returned to Alice by Bob, along with group refunds with more than one tuple in the group. Thus, the construction for optimal refunds, as outlined in Lemma 4.3.2, continues to work for GroupRefunds as well.

Similarly, the only change in the safety proof for MonotoneRefunds, Lemma 4.3.3, occurs in Step 3, where instead of checking if the tuple t has the same id as the refund message, we check if the tuple belongs to the group denoted by the id in the refund message. Given that GroupRefunds is safe for individual tuples, the following holds:

$$
\begin{aligned}
P(W) - R(W) &= \sum_{(t,k)\in T(W)} k * p(t[tid]) - \sum_{\rho\in W} \sum_{(t,n)\in T(W)} I(t,\rho) * p(t[tid]) \quad \text{(Defs 4.2, 4.6)} \\
&= \sum_{(t,k)\in T(W)} k * p(t[tid]) - \sum_{(t,n)\in T(W)} p(t[tid]) \sum_{\rho\in W} I(t,\rho) \\
&= \sum_{(t,k)\in T(W)} k * p(t[tid]) - \sum_{(t,n)\in T(W)} r * p(t[tid]) \\
&= \sum_{(t,k)\in T(W)} (k-r) * p(t[tid]) \\
&\geq \sum_{(t,k)\in T(W)} p(t[tid]) \quad \text{(From the paragraph above.)}
\end{aligned}
$$

Hence, GroupRefunds is both safe and optimal. □

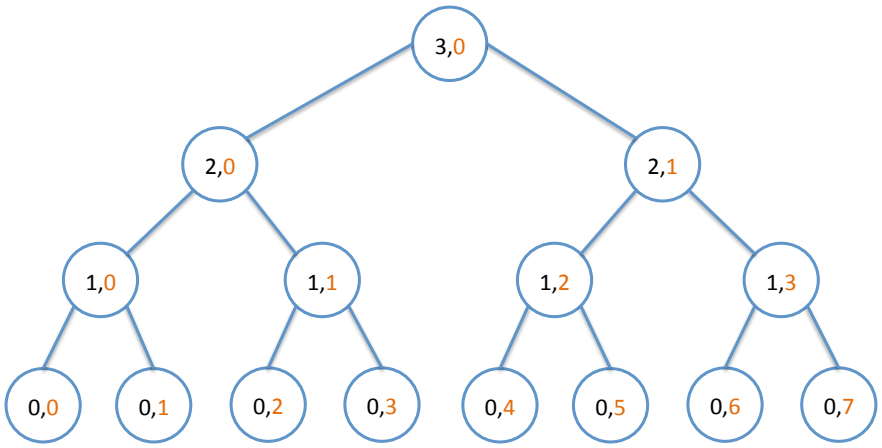


Figure 4.2: Group identifier assignment for tree-structured group coupons. The tree is built for a database with 8 tuples. It is a balanced binary tree where each node, including the leaves, is assigned an identifier `(height, id)`, where `height` is the height of the node (leaves are at height 0), and `id` is the node's order amongst the nodes at its height, where the leftmost node is assigned the id 0, the next node to the right the id 1, and so on.

There are various ways to group tuples when computing group coupons since any arbitrary subset of tuples in the answer can be a valid candidate. We now show a tree structured group coupon construction scheme for general conjunctive queries.

Tree-Structured Group Coupons

We now present a tree structured grouping of tuples. In our construction of tree-structured coupons, we require that tuple ids be integers.

Figure 4.2 illustrates how the group coupon identifiers are constructed and how the groups are formed. We construct a binary tree by treating *all* the tuples of the relation `R`, order by `id`, as leaves. The group identifier of the leaves is $(0, id)$. The next level of the tree is constructed by successively grouping nodes with `ids` $2n$ and $2n+1$ to give a group identifier $(1, n)$. Here, 1 represents the height of the node. The higher levels of the tree are constructed recursively, by combining the nodes at the lower levels. We stop combining nodes when we

only have one node, which forms the root of the tree. Note that we pad the database so that its cardinality is always a power of 2. With this construction, a group node with group id (h, n) is a group that includes all rows with ids in $\{2^h n, \ldots, 2^h(n+1)-1\}$.

Formally, the hash digest for tree coupons is computed as follows:

$$\texttt{treeHash}(\texttt{uid}, \texttt{height}, \texttt{id}, \texttt{version}, \texttt{qid}) = \mathcal{H}(\texttt{uid} \oplus \texttt{height} \oplus \texttt{id} \oplus \texttt{version} \oplus \texttt{qid} \oplus \kappa) \tag{4.7}$$

While asking for a refund, and to reduce the number of refund requests Alice makes, she asks for the largest valid group refund, that is the group refund with the maximum height such that all the tuples in that group are eligible for a refund.

There are different ways to compute tree-structured coupons and we find that the specific algorithm affects performance. We now describe two algorithms to construct the tree-structured coupons given a query.

We construct the coupon trees in two ways: StreamTree (Algorithm 6) and BlockTree (Algorithm 5). In the BlockTree algorithm, the entire set of certificates are computed before the query's answer is returned to the user; while for the StreamTree algorithm, the certificates are computed as the cursor moves forward through the query's result set.

BlockTree, outlined in Algorithm 5, works by inserting the leaves for the current query Q into the working space, table `tempTable`. It then performs a series of `group-by-having` aggregation SQL queries to construct the layer one level above, and so on. The algorithm is blocking in nature, since the ids of the query's output tuples must be first inserted into `tempTable` before the query's answer can be returned.

We can also define a modification to BlockTree that avoids computing the query twice, once at Line 2 while populating `tempTable` with the leaves and another at the end in Line 3. We call this modification `BlockTreeInt` where the query `Q` is evaluated once and stored in a relation `result` in memory. This is done before Line 1. Subsequently, references to `Q` in Lines 2 and 3 are replaced by references to `result`. This approach can be potentially useful when evaluating the query is expensive.

Algorithm 5: BlockTree Coupon Construction

Input : Query id τ, query `Q`, user id `u`.

Output: Compute the coupons and the query.

```
begin
    // tempTable has schema (height, id, version).
    // refunds has schema (uid, height, id, version, qid, digest).
    tempTable ← ∅
    refunds ← ∅
    shiftval ← 1
    height_c ← 0
1   INSERT INTO tempTable
2      SELECT height, t.id, sum(t.ver) FROM Q AS t GROUP BY height, t.id;
    while true do
        INSERT INTO tempTable
          SELECT height + 1, t.id >> shiftval
          FROM tempTable t
          WHERE t.height = height_c
          GROUP BY height + 1, t.id >> shiftval
          HAVING COUNT(*) > 1;
        // Below, in PostgreSQL, FOUND returns TRUE if the previous SQL query returns a non-empty answer.
        if NOT FOUND then
            break
        height_c ← height_c + 1
    // treeHash computes the hash as described in Equation 4.7.
    INSERT INTO refunds
       SELECT u, height, id, τ, treeHash(u, height, id, ver, τ)
       FROM tempTable t;
3   return (SELECT * FROM refunds), Q
```

The StreamTree algorithm works by ordering the results of a query by the primary key `id` and making a single pass over the data while adding an extra UDF that includes the code described in Algorithm 6. As the buyer advances the cursor, the temporary workspace, `tempTable`, is gradually populated with the tree for the query. For example, in Figure 4.2, if a query selects all the nodes, the nodes that are added to `tempTable` would be the order seen by a post-order traversal of the tree.

Algorithm 6: StreamTree Coupon Construction

Input : Query id τ, version `ver`, user id `u`, tuple id `idIn`.

Output: Updates `tempTable` to incrementally compute the coupons.

```
begin
    // tempTable has schema (height, id, version).
    height_c ← 0
    id_c ← idIn
    while true do
        INSERT INTO tempTable (height, id, version) VALUES (height_c, id_c, ver);
        if id_c % 2 == 0 then
            break
        else if EXISTS (SELECT 1 FROM tempTable WHERE height = height_c AND id = id_c - 1) then
            height_c ← height_c + 1
            id_c ← id_c >> 1
            ver ← ver + (SELECT version FROM tempTable WHERE height = height_c AND id = id_c - 1)
        else
            break
```

4.5 Evaluation

We now experimentally evaluate the performance of the various refund protocols and their implementations.

We answer the following questions:

1. How much can Alice benefit from paying only once for tuples and what performance penalty, if any, should she expect in lieu of this benefit?
2. How costly is it to compute group coupons versus computing only per-tuple coupons? Further, how much time do group coupons save when asking for refunds compared to single-tuple coupons?
3. How do the naïve approaches (§4.2) to pricing, *i.e.*, `CountBlock`, `CountStream`, and `History` perform compared to the refund-based approaches (§4.4.2), *i.e.*, `MonotoneRefunds`, `BlockTree`, `BlockTreeInt`, and `StreamTree`?

We run all experiments on a single server running PostgreSQL 9.4 over OS X 10.10.5, equipped with a 2.7 GHz Intel Core i7 processor and 16 GB DDR3 RAM. We use the SHA1

implementation of the module `pgcrypto`. The client resides on the same machine as the database.

The data setup for the experiments consists of a binary relation with two integer columns, `(tid, val)` in a table, `test`, with 524,288 (2^{19}) rows. Column `tid` is a primary key starting with a value of 0, while `val` is an integer column where the values are a random permutation of $\{0, \ldots, N-1\}$ where N is the size of `test`.

The query workload consists of queries that ask for tuples satisfying predicates within a randomly chosen range of sizes in $\{1, 8, 64, 512, 4096\}$. We consider the following classes of queries: `pkey.simple` performs a range selection on the primary key, which is the key on which the data is sorted on disk and has a clustered index; and `other.simple`, which performs a range query on the column `val` over which no indices have been constructed.

The queries are:

```
pkey.simple:
    SELECT * FROM test WHERE tid >= l AND tid <= u
other.simple:
    SELECT * FROM test WHERE val >= l AND val <= u
```

For identical values of l and u, the queries return answers with identical cardinalities.

4.5.1 Overall Results

We now investigate the benefits of optimal history-aware pricing and the associated performance penalty of approaches that can achieve such pricing.

In Figure 4.3, we simulate the amount of money a buyer must pay with different approaches. We construct a workload with 100 instances of `pkey.simple`. On the x-axis we vary the size of the ranges, that is, $u - l + 1$, while we use different distributions for selecting the value of l: *Uniform*, which chooses l uniformly at random, and *Zipf(α)*, which selects l from a heavy-tailed distribution where some values are significantly more likely than others. We experiment with $\alpha \in \{1.7, 2, 3\}$, which enables us to vary the degree of the skew. We

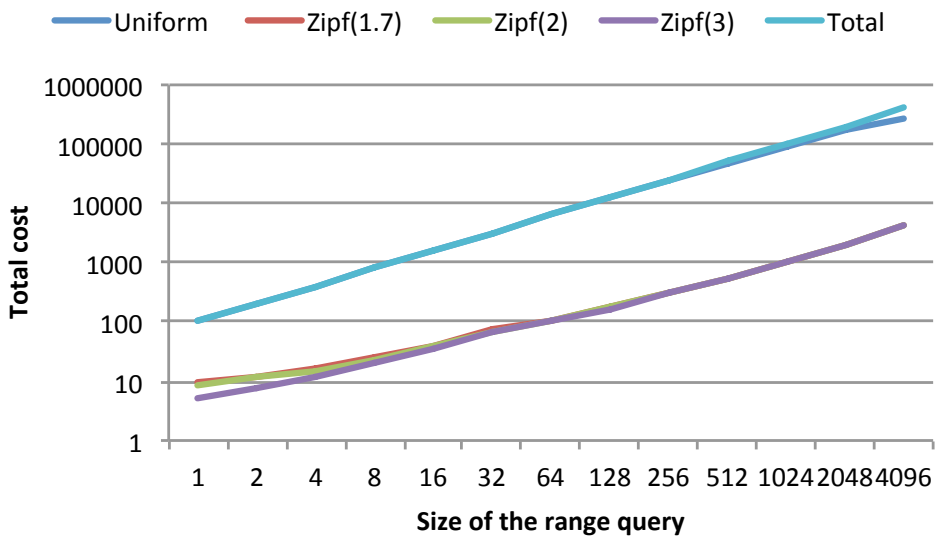


Figure 4.3: Amount paid for data with different distributions for the parameters of `pkey.simple`. "Total" is the amount paid if history-aware pricing is not used, while the other plots show the amount paid with history-aware pricing. "Uniform" denotes the case when the query's parameters are chosen uniformly at random, while "Zipf" denotes the case(s) where the query parameter l is chosen by sampling from the given Zipf distribution.

run 100 such simulations and take the average amount of money that the buyer must pay. For reference, we also plot the cost of the data if no history-aware pricing is employed such as with `CountBlock`. This is denoted by "Total."

As expected, with Uniform distribution, it is unlikely that the user may buy the same data across different queries, especially when the range is small. It is only at large ranges that noticeable differences are seen: refund-based pricing is 1.2× and 1.4× cheaper at ranges of sizes 2048 and 4096, respectively. But savings are dramatic for skewed distributions. With $\alpha = 1.7$, the history-aware pricing for point queries (range size as 1) is 10× cheaper than history-agnostic pricing, while for $\alpha = 3$, it is 19× cheaper. For large range sizes, Alice buys data that is 99× cheaper than a history-agnostic approach such as `CountBlock`.

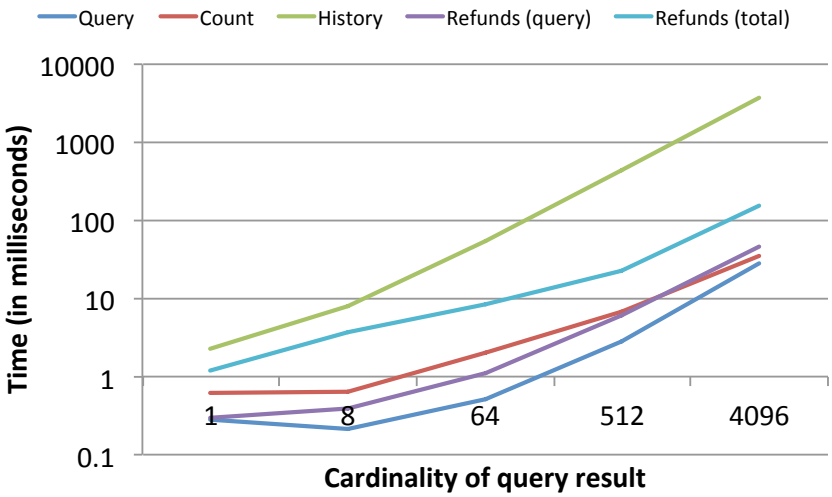


Figure 4.4: Total time, in milliseconds, taken to evaluate `pkey.simple` with randomly chosen initialization for l and a range specified on the x-axis. The time also includes the overhead of counting the cardinality of answers for `CountBlock` and `CountStream`, updating the history bit vector for `History`, and computing and verifying refunds for the refund-based approaches. In the figure, "Query" represents the time to execute just the query, "Count" represents the best time amongst `CountBlock` and `CountStream`, "Refunds (query)" represents the fastest time amongst `MonotoneRefunds`, `BlockTree`, `BlockTreeInt`, and `StreamTree` to compute the query and the refund coupons, and "Refunds (total)" is the best time, amongst the four refund-based pricing technique, including the time to ask for refunds in addition to computing them and answering the query.

We now look at the overhead of obtaining these cost savings. In Figure 4.4, we show the time taken to answer queries by the *best* naïve technique and the *best* refund-based technique. The figure shows the total time that includes (a) the time to evaluate `pkey.simple` with range lengths specified on the x-axis, and (b) the overhead of the associated pricing technique, that is, counting for `Count`, updating the history bit vector for `History`, computing coupons for `Refunds (query)`, and both computing coupon and asking for refunds for `Refunds (total)`.

As the figure shows, all techniques take more time than just executing the query, typically at least 2× more. Up until and including range lengths of 512, computing coupons is still cheaper than the history-agnostic approach adopted by `Count`. If refunds are requested though, the best refund-based technique becomes at least 4× more expensive than the best count technique. In all cases, `History` is at least an order of magnitude slower compared to counting and refund based techniques.

Thus, in the best case, refund-based pricing provides both reduced costs as well as reduced query execution time compared with the history-agnostic pricing methods, `CountBlock` and `CountStream`. This best case occurs when refunds can be requested when Bob and Alice are idle. If this is not the case, refund-based pricing still significantly reduces costs, while adding a 4× to 5× time penalty against the best history-agnostic pricing method.

4.5.2 Overhead of Refunds

In this section, we compare the time to compute group coupons in GroupRefunds to the time for computing only per-tuple coupons in MonotoneRefunds. Then, we compare the time saved in asking for refunds with GroupRefunds versus MonotoneRefunds. We measure the overhead of using MonotoneRefunds versus GroupRefunds on (a) the time to evaluate a query and return the result and the coupons, and (b) the time to ask for refunds of data previously purchased.

We assign random l and u values to `pkey.simple` to obtain queries that are executed twice. Thus, all the tuples in the second query are eligible for refunds. We compare the time it takes to execute the query and compute the coupons using MonotoneRefunds and `BlockTree`. Then, for MonotoneRefunds, we ask for refunds one tuple at a time, while for `BlockTree`, we ask for the group refunds for the largest groups (while avoiding overlaps) until all the tuples are covered.

Figure 4.5 shows the results. When comparing only the time to evaluate the query and the coupons, `MonotoneRefunds` is faster than `BlockTree` since `BlockTree` must compute additional group coupons along with the singleton group coupons. For point queries, `BlockTree`

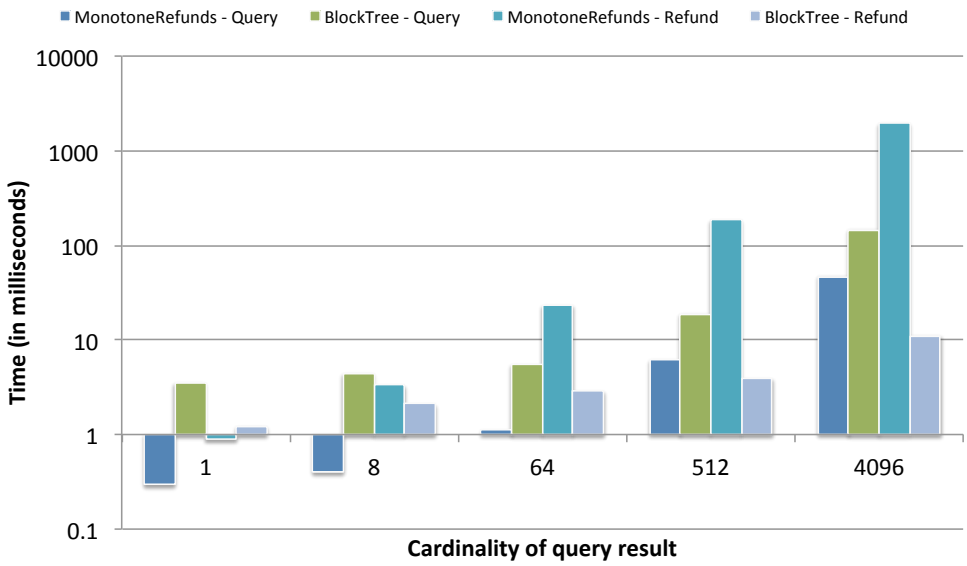


Figure 4.5: A comparison of MonotoneRefunds and BlockTree runtimes, in ms, to evaluate the query and generate the coupons (denoted by "MonotoneRefunds - Query" and "BlockTree - Query", respectively) and the time, in ms, for refund requests (denoted by "MonotoneRefunds - Query" and "BlockTree - Query", respectively). The parameterized query used for the workload is `pkey.simple` and each query is executed twice. The time is measured for the second query for which the corresponding refund protocols are exectued.

is 12× slower, while for larger ranges, such as 4096, it is 3.1× slower. This is expected since `MonotoneRefunds` needs to do just one pass over the table and computes the coupons on the fly as the cursor is advanced. `BlockTree` must make two passes of the data, once to select the leaves of the tree-structured coupons and again to evaluate the query itself. It must also suffer the additional cost of computing the tree.

But if we also take into account the overhead of asking for refunds, this advantage quickly vanishes as range sizes are increased. While `MonotoneRefunds` is 4× and 1.7× faster for ranges of sizes 1 and 8, respectively, `BlockTree` is faster by 2.9×, 8.4×, and 12.6× for ranges of sizes 64, 512, and 4096, respectively.

Thus, if a query is expected to return a small number of tuples or if it is known that group refunds can not be constructed, say when the tuples selected do not have adjacent tuple ids, `MonotoneRefunds` will outperform `GroupRefunds`.

One way to improve `BlockTree` and other tree structured coupons is to increase the fanout of the internal tree nodes. Then, fewer coupons would be computed during the query. Further, in an actual deployment, refund requests can be asked when the buyer has spare computation cycles as opposed to being asked after each query. This does not reduce the workload on the seller, though.

4.5.3 Naïve Techniques versus Refund-Based Techniques

In this series of experiments, we assign random l and u values to the test queries. These randomly parameterized queries are executed once along with the additional processing of the corresponding naïve or refund-based technique. For refund-based techniques, we do not show the overhead of asking for refunds because the overhead is sensitive to the actual overlap in the queries.

Figure 4.6 shows the average time per query for `pkey.simple`. As expected, `MonotoneRefunds` outperforms the group-based coupons (note that we do not include the time to ask for refunds). `MonotoneRefunds` also outperforms all the naïve approaches for all but the largest range size. For the largest range size, `CountBlock` performs the best and better than `CountStream`. This is because `CountBlock` only uses two SQL statements to execute the query and compute the count whereas `CountStream` must execute an `UPDATE` statement to update the running count of tuples for the query every time the cursor is advanced. This overhead becomes significant as the cardinality of the query increases.

In all cases though, `History` is the worst performing technique since the cost of modifying bits for each output tuple and writing those edits back to disk become increasingly expensive.

Figure 4.7 shows the average time per query for `other.simple`. Unlike for `pkey.simple`, `MonotoneRefunds` does not significantly outperform the other alternatives. This is because the query itself is expensive. In the absence of indices, the database does a full scan of

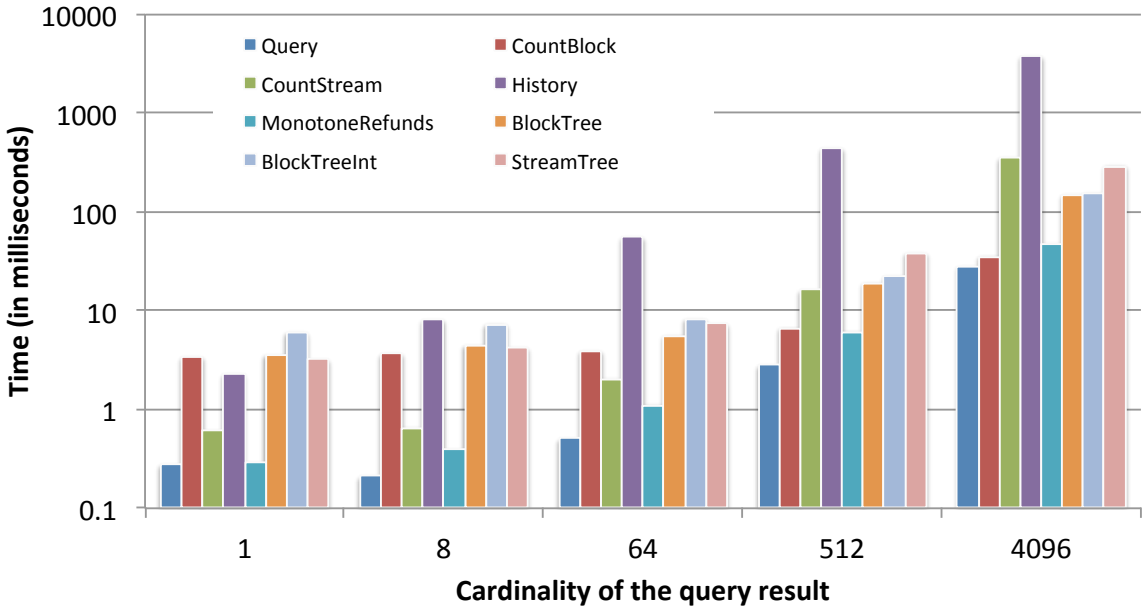


Figure 4.6: The total time, in ms, to evaluate the query and run the various pricing techniques. The parameterized query used for the workload was `pkey.simple` and numbers are averages of executing a workload of 40 queries. For refund-based techniques, only the time to run the query and generate coupons is shown.

the table to compute answers and this cost dominates the total cost. As result sizes increase though, the overheads of `CountBlock` and the refund-based techniques become more noticeable.

In the figure, we do not show measurements for `CountStream` since they were always worse than for `CountBlock`. We do not show measurements for `History` because it was significantly slower: 2×, 43×, and 273× slower than `MonotoneRefunds` for 64, 512 and 4096 sized ranges. This is because unlike for `pkey.simple`, since the column `val` is a random permutation, the indices in the history bit vector, corresponding to their tuples, are no longer clustered to adjacent bits in the history bit vector and this increases the overhead of commits.

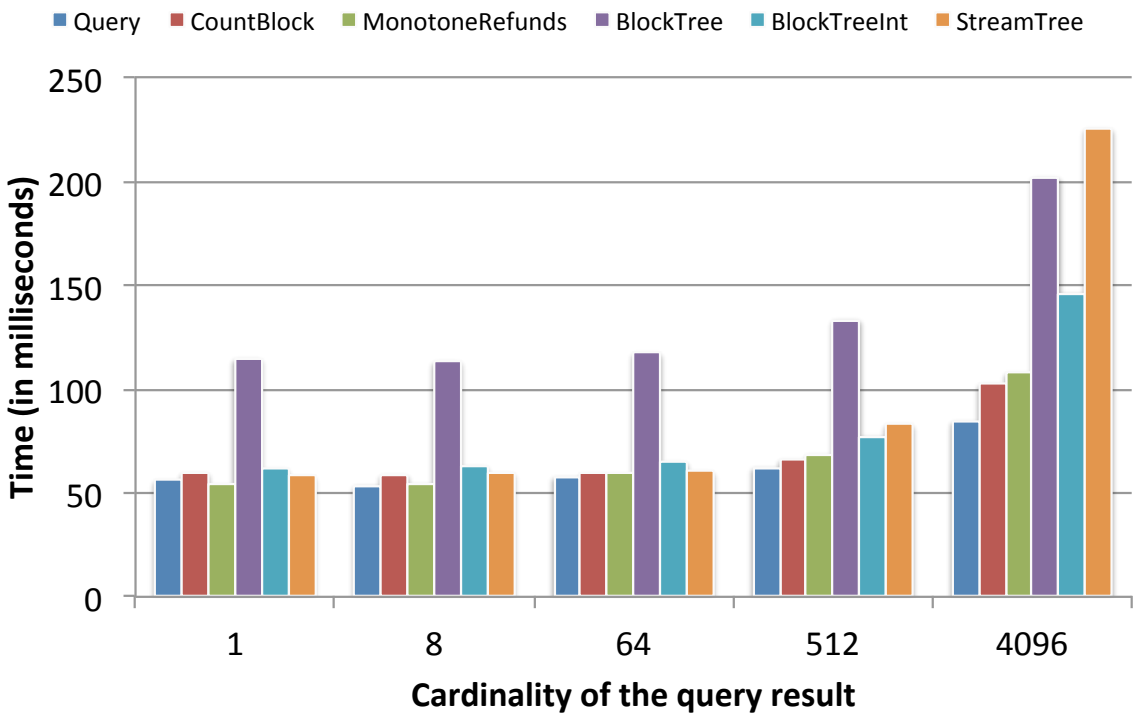


Figure 4.7: The total time, in ms, to evaluate the query and run the various pricing techniques. The parameterized query used for the workload is `other.simple` and numbers are averages of executing a workload of 20 queries. For refund-based techniques, only the time to run the query and generate coupons is shown.

Finally, `BlockTree` is approximately 2× more expensive than `CountBlock` and `MonotoneRefunds` since the query is more expensive to compute and `BlockTree` must execute the query twice.

If we only consider the time to compute coupons and the query, then `MonotoneRefunds` will always be the fastest amongst the refund-based techniques. But in applications with high overlap in the data purchased through different queries, the cost of refunds can become significant. In such cases, `GroupRefunds` might be more efficient than `MonotoneRefunds`. Unfortunately, as can be seen from Figures 4.6 and 4.7, different group refund techniques do well in different settings and there is no single technique that outperforms others every

single time. In Figure 4.6, `BlockTree` does uniformly well across many different ranges while in Figure 4.7, `BlockTreeInt` performs uniformly well. Also, for both queries, `StreamTree` outperforms other group refunds for queries with small ranges.

4.6 Conclusion

We provide a novel, lightweight and fast method to support optimal, history-aware pricing of data APIs. With our techniques, even if a buyer makes multiple API calls and ends up purchasing the same data item more than once, she is only charged once for the purchase. To enable this, we propose a framework for pricing that allows buyers to refund repeat purchases of data. We then provide a compact, secure and tamper proof protocol that enables such refunds and guarantees that if there is a repeat purchase, it is always possible to get refunds. Subsequently, we generalize the protocol to handle updates and multiple users; and provide performance improvements through the use of group refunds. We experimentally evaluated our protocol and compare it to current pricing techniques that do not provide history-aware pricing. Even with 100 API calls, we identify 10× to 99× potential reduction in data costs, while experiencing comparatively modest increases in query runtimes, by a factor of 2× to 5×.

Chapter 5

RELATED WORK

We now present related work for the three previous chapters.

5.1 Managing Data Use Agreements

DataLawyer is related to many different areas of database management research.

Data auditing. Most auditing systems [57, 4, 52, 73, 35] detect data misuses, but only after the fact. In the online case, some prior techniques such as those that rely on reordering of queries [57] are not applicable; other techniques are data instance independent [73] and only make use of the structure of the queries themselves, unlike our semantics, which are data dependent.

Privacy Mechanisms. DataLawyer's goal is not to protect privacy, its goal is to verify that queries follow a pre-defined set of usage policies.

Access control. Access control approaches [4, 40] do not handle the case when users are allowed to see individual data items but do not have permission to perform certain operations, such as joins, on these data items.

Usage Models. The $UCON_{ABC}$ model [86], is a generic framework that models 16 usage control scenarios (such as UNIX access control lists and Digital Rights Management). DataLawyer subsumes six of those, the additional complexity is due to the expressiveness of the relational model.

DBAs may also automatically enforce performance related policies using Teradata'a Active Management System [104], but they do not have support for general data usage policies.

Complex Event Processing (CEP). Theoretically, some policies may be encoded as patterns for a CEP engine [32, 17, 119, 81] that can then search them in a stream of log

increments as new queries arrive. Unlike CEP engines though, DataLawyer controls if and when to generate the log stream, which our experiments show to be a critical optimization. CEPs usually use non-deterministic finite automata to represent patterns. DataLawyer's policies are more general since arbitrary code is permitted for log-generating functions.

Multi-Query Optimization (MQO). Many techniques [5, 50, 6] for MQO identify common sub-expressions in the query and then store the intermediate results to be reused by multiple queries. SharedDB [42] also provides a complementary set of techniques to ours. Although DataLawyer can use these techniques, our main techniques exploit the boolean nature of the policies. Further, DataLawyer must also worry about regular and frequent updates to the underlying data (usage logs) that provide opportunities for significant improvements.

Triggers. Triggers [82, 89] are only executed for DML statements and not for non-DML statements unlike the policies discussed in this thesis.

Provenance Management. Provenance and annotation management [18, 19] store how data moves through databases over its life cycle. Their algorithms focus on reducing the provenance storage overhead and its querying. These techniques are orthogonal to our system, for which, provenance is just one possible log generating function.

User Interface. The interface that displays the message to the user and recommends alternative actions was demonstrated in an early prototype of our system [108].

5.2 Pricing Shared Optimizations

Today, cloud providers use two strategies for pricing optimizations. In the first, the cost of the optimization is included in the base service price. For *e.g.*, Amazon SimpleDB [12] automatically indexes user data and includes the corresponding overhead in the base-price computation (45 bytes of extra storage are added to each item, attribute, and attribute-value). Similarly, SimpleDB and SQL Azure [71] automatically replicate data and include that cost in the base service cost. The key limitation with this approach is that the cloud must decide up-front what optimizations are worth offering and it forces users to pay for these optimizations. In other cases, users choose desired optimizations and pay their exact

cost. For example, in Amazon RDS [9] a user can choose to launch and pay-for a desired number of read-replicas to speed-up her query workload. This approach, however, works well only in the absence of collaborations.

Significant recent work studies existing cloud pricing schemes, economic models, and their implications [61, 105, 115]. In contrast we develop a new pricing mechanism.

Most closely related to our work, Dash *et al.*, developed an approach for pricing data structures (indexes, materialized views, etc.) in a DBMS cloud cache [26]. In their approach, the cloud selects the structures to build based on the notion of regret and its cost is amortized to the first N queries that use it. To compute regret, the cloud relies on user supplied budget functions, that indicate their willingness to pay for various quality of service. In follow-up work Kantere *et al.* [56] tuned their approach and developed a regression-based technique to predict the extent of cost amortization. In contrast to our work, this previous approach relies on users being truthful and does not guarantee that the cost will be recovered. For example, consider a user who needs to run one, very expensive query over a private dataset. No structure will be implemented if she is truthful. Instead, she thus submits a large number of inexpensive queries over the same dataset while she expresses her willingness to pay zero for processing the extra queries, yet indicates a preference for low execution times over low costs. The regret-based approach will let her manually pick slow and cheap service for these queries. It will then compute the maximum possible regret for the missing data structure that would have enabled faster plans for these queries. When the cloud accumulates enough regret, she can run the expensive query and pay a small fraction of the total cost of the optimization.

Significant research applies economic principles to resource allocation in distributed systems [2, 16, 20, 24, 39, 93, 96, 113], collaboration promotion in peer-to-peer systems [78, 77, 111], or more recently, VM allocation in the cloud [106]. We study how to choose and price optimizations rather than allocate processing resources. The Mariposa distributed database system [101] introduced a microeconomic paradigm for optimizing distributed query evaluation and data placement. This is a problem orthogonal to ours.

We build on the Shapley Value Mechanism, which is an instance of a Moulin Mechanism [74] that have been designed for various combinatorial cost-sharing problems where the cost of servicing a set of players is determined by solving a *offline* combinatorial optimization problem defined by the set [84]. We design Moulin mechanisms in an online setting.

Online mechanisms [79, Ch. 16] consider games where valuations come one at a time. While there is work on characterizing truthful mechanisms to maximize social utility in dynamic games [79, Thm. 16.17], to the best of our knowledge, no work applies to cost-sharing in dynamic games.

5.3 Optimal History-Aware Pricing with Data APIs

Apart from pricing APIs by summing up the cost of the tuples that are returned due to an API call, other forms of pricing methods have also been proposed in the literature, though they are not as widespread as tuple-based pricing. The common idea in all the approaches is to directly price queries as opposed to pricing individual tuples. Approaches have been proposed that price data based on minimal why-provenance [102], information and determinacy [59, 60, 64], and statistical noise [62]. We know of only one approach that has explicitly looked at history-aware pricing [63, 60]. Optimal history-aware pricing in the presence of prices assigned to queries as opposed to individual tuples is NP-hard. In this approach, they explicitly store the queries purchased by user at the seller and incur significant overhead while pricing new queries.

Our solution relies on explicit support from the seller. In the absence of such support, as shown in Example 4.0.1, it may be impossible to provide optimal history-aware pricing of data. But, buyers can still reduce their costs by caching answers to queries they purchase. They can subsequently use techniques from query answering using views [51] to only acquire such that that is not present in their caches and integrate the new data with the cached data to determine the answer to their queries. Systems that provide semantic caching [25, 23, 94] and transactional caches [90, 91] are examples of such systems.

Chapter 6

CONCLUSION

Data is transforming science, business, and governance by making decisions increasingly data-driven and by enabling data-driven applications. The data used in these contexts usually has significant economic or social value. Frequently, data is purchased from a provider. The price is often linked to how the data will be used and the allowed usage is typically detailed in a license agreement. While there is significant research to help users easily express and efficiently execute complex analytics on big data, tools to manage the economic value of data (prices and licenses) are lacking. Current solutions rely on extensive and expensive support from economists, auditors and lawyers. Further, data analysis and data-driven applications increasingly rely on public clouds for their computational needs. Clouds offer scalability and the flexibility to trade-off price and performance and cloud providers know how to price for individual use. But, cloud resources are frequently shared by multiple users, especially when users analyze a common dataset. How to price such shared resources is poorly understood and when pricing ignores the shared nature of use, the cloud resources are significantly underutilized and users can not realize the full value of their data.

In this thesis, we make three contributions that deal with license agreements for data, pricing data, and pricing computation.

First, we develop DataLawyer, a middleware system to specify and enforce data use policies on relational databases. Our approach includes a SQL-based formalism to precisely define policies and novel algorithms to automatically and efficiently evaluate them. Experiments on a real dataset from the health-care domain demonstrate overhead reductions of up to 330× compared to a direct implementation of such a system on existing databases.

Next, we study how a cloud data service provider should activate and price optimizations

that benefit many users. We show how the problem can be modeled as an instance of cost-recovery mechanism design. We also show how the Shapley Value mechanism solves the problem of pricing a single optimization in an offline game. We then develop a series of mechanisms that enable the pricing of either additive or substitutive optimizations in either an offline or an online game. We prove analytically that our mechanisms are truthful and cost-recovering. Through simulations, we demonstrate that our mechanisms also yield high utility compared with a regret-based state-of-the-art approach.

Lastly, we provide a novel, lightweight and fast method to support optimal, history-aware pricing of data APIs. With our techniques, even if a buyer makes multiple API calls and ends up purchasing the same data item more than once, she is only charged once for the purchase. To enable this, we propose a framework for pricing that allows buyers to refund repeat purchases of data. We provide protocols for refunds, extensions to handle updates and multiple users, and performance optimizations to reduce the overhead of exercising refunds. Experimental evaluation identify significant potential reduction in data costs, while experiencing comparatively modest increases in query runtimes.

We now discuss immediate directions for future research that follow from the thesis.

In managing data use agreements, there are opportunities to extend the existing work in the way we model policies, to develop algorithms for more efficient policy checks, and to investigate DataLawyer's user-friendliness. Specifically,

1. For policy modeling, an important unanswered question is the feasibility of automatically generating policies for materialized views. Materialized views are a powerful method for creating custom data products as well as a way to increase system performance. Given the definition and an instance of a materialized view, it would be useful to study the possibility of generating more efficient policies for the materialized view, and thus not rely on the policies defined over the original data.
2. To verify policies, this thesis considered algorithms that rely only on query rewriting. A promising area of future work would be to consider lightweight modifications to the

query execution engine to support on-the-fly policy checks while the query is executing. Further, more efficient algorithms for policy evaluation may exist if we know more about the structure of the log generating functions, as opposed to treating them as black box functions. Lastly, adapting policy evaluation to distributed settings and understanding the implications of data distribution on the efficiency of policy checks in a shared-nothing computing cluster would be helpful in augmenting distributed, parallel data processing engines to support data use management functionalities.

3. The thesis does not explore the user experience when DataLawyer rejects a query. Further research on how to explain query rejections to a user and to automatically suggest how users should modify their queries would be very useful.
4. Lastly, to handle malicious users and to handle settings where DataLawyer can not directly observe user actions would require a rethink of both the models and the techniques for enforcing policies.

In the problem of pricing shared computation, there are two promising and useful extensions of our mechanisms that were not explored by the thesis. The first extension is related to how we can modify the mechanisms to work with value functions for optimizations that are neither additive nor substitutive, but those that exhibit sub-modularity, which can naturally model economies of scale. The second extension is about how to design mechanisms for adversarial optimizations, where implementing an optimization for one user can reduce the utility for another user. An example of such an optimization is database replication. Although the new replica may improve read performance for one user, it may reduce write throughput for another user. Adversarial optimizations are increasingly likely as users become less isolated from other users with the adoption of virtualization.

Finally, designing data APIs is a rich area for future extensions and research. The immediate direction for future work is to design an optimizer to choose the optimal refund algorithm and to explore alternate ways of computing coupons that may provide safe refunds for attacker models that are weaker than those assumed for cryptographic hashes. It is

likely that throughput can be improved with a weaker adversarial attack model. Another line of interesting research would be to theoretically understand the trade-offs between the additional overhead of computing coupons by the seller versus the overhead of communication during the refund phase. Lastly, the problem of pricing data when the price of the individual tuple is either dependent on other tuples in the answer or the prior purchase history of the user is another class of rich problems to develop models and protocols for optimal pricing.

The management of premium data is an important, impactful, and novel data management problem that companies and researchers regularly encounter. As data markets and cloud computing mature, these concerns will become increasingly more important. Developing better tools to handle these concerns provides a rich set of problems that span theory, economics, systems and security.

BIBLIOGRAPHY

[1] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Abramson, D., Buuya, R., and Giddy, J. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems 18*, 8 (Oct. 2002).

[3] Agrawal, P., Silberstein, A., Cooper, B. F., Srivastava, U., and Ramakrishnan, R. Asynchronous view maintenance for vlsd databases. In *SIGMOD Conf.* (2009), pp. 179–192.

[4] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. Hippocratic databases. In *VLDB* (2002), pp. 143–154.

[5] Agrawal, S., Chaudhuri, S., and Narasayya, V. R. Materialized view and index selection tool for microsoft sql server 2000. In *SIGMOD Conference* (2001), p. 608.

[6] Ahmad, Y., Kennedy, O., Koch, C., and Nikolic, M. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB 5*, 10 (2012), 968–979.

[7] Amazon Web Services (AWS). http://aws.amazon.com.

[8] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[9] Amazon Relational Database Service. http://aws.amazon.com/rds/.

[10] Amazon S3: Requester Pays Buckets. http://docs.amazonwebservices.com/AmazonS3/latest/dev/index.html?RequesterPaysBuckets.html.

[11] Amazon Simple Storage Service (Amazon S3). http://www.amazon.com/gp/browse.html?node=16427261.

[12] Amazon SimpleDB. http://www.amazon.com/simpledb/.

[13] Amazon Public Data Sets. https://aws.amazon.com/public-data-sets/.

[14] Windows Azure Platform. http://microsoft.com/windowsazure/.

[15] Windows Azure Storage Services REST API Ref. http://msdn.microsoft.com/en-us/library/dd179355.aspx.

[16] Balazinska, M., Balakrishnan, H., and Stonebraker, M. Contract-based load management in federated distributed systems. In *Proc. of the First NSDI Symp.* (Mar. 2004).

[17] Buchmann, A. P., and Koldehofe, B. Complex event processing. *it - Information Technology 51*, 5 (2009), 241–242.

[18] Buneman, P., Chapman, A. P., and Cheney, J. Provenance management in curated databases. In *In SIGMOD 06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), ACM, pp. 539–550.

[19] Buneman, P., Cheney, J., Tan, W. C., and Vansummeren, S. Curated databases. In *PODS* (2008), pp. 1–12.

[20] Buuya et. al. Economic models for management of resources in peer-to-peer and grid computing. In *Proc of SPIE* (Aug. 2001).

[21] Digital Folio. https://www.cartbound.com/PriceIntelligence/API.

[22] Chandra, A. K., and Merlin, P. M. Optimal implementation of conjunctive queries in relational data bases. In *STOC* (1977), pp. 77–90.

[23] Chidlovskii, B., and Borghoff, U. M. Semantic caching of web queries. *The VLDB JournalThe International Journal on Very Large Data Bases 9*, 1 (2000), 2–17.

[24] Chun, B. N. *Market-Based Cluster Resource Management*. PhD thesis, University of California at Berkeley, 2001.

[25] Dar, S., Franklin, M. J., Jonsson, B. T., Srivastava, D., Tan, M., et al. Semantic data caching and replacement. In *VLDB* (1996), vol. 96, Citeseer, pp. 330–341.

[26] Dash, D., Kantere, V., and Ailamaki, A. An economic model for self-tuned cloud caching. In *ICDE* (2009), pp. 1687–1693.

[27] Rate data.gov.uk. http://www.nationalarchives.gov.uk/doc/open-government-licence/.

[28] DataLawyer Source Code. https://github.com/prup/data-lawyer, 2014.

[29] DataMarket. https://datamarket.com.

[30] DataSift. http://datasift.com/terms/.

[31] DataSift: Pylon Facebook API. http://datasift.com/products/pylon-for-facebook-topic-data/.

[32] Demers, A. J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., and White, W. M. Cayuga: A general purpose event monitoring system. In *CIDR* (2007), pp. 412–422.

[33] Digital Folio. http://www.digitalfolio.com/Home/TermsOfService.

[34] Edelman, B. Using internet data for economic research. *Journal of Economic Perspectives 26*, 2 (2012), 189–206.

[35] Fabbri, D., LeFevre, K., and Zhu, Q. Policyreplay: Misconfiguration-response queries for data breach reporting. *PVLDB 3*, 1 (2010), 36–47.

[36] Factual. www.factual.com.

[37] Factual. http://developer.factual.com.

[38] Fan, W., Geerts, F., and Libkin, L. On scale independence for querying big data. In *PODS* (2014), pp. 51–62.

[39] Ferguson, D., Nikolaou, C., Sairamesh, J., and Yemini, Y. Economic models for allocating resources in computer systems. In *Market based Control of Distributed Systems*, S. H. Clearwater, Ed. World Scientist, Jan. 1996.

[40] Ferrari, E. *Access Control in Data Management Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[41] Foursquare terms of use. https://foursquare.com/legal/api/platformpolicy.

[42] Giannikis, G., Alonso, G., and Kossmann, D. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow. 5*, 6 (Feb. 2012), 526–537.

[43] Glavic, B., and Alonso, G. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE* (2009), pp. 174–185.

[44] Google Maps. maps.google.com.

[45] GNIP: Firehose. https://gnip.com/products/realtime/firehose/.

[46] Gonzalez et al. Google fusion tables: data management, integration and collaboration in the cloud. In *Proc. of SOCC* (2010), pp. 175–180.

[47] Google App Engine. http://code.google.com/appengine/.

[48] Google App Engine Datastore. http://code.google.com/appengine/docs/datastore/.

[49] Gray, J., and Szalay, A. Science in an exponential world. *Nature 440*, 23 (2006), 413–414.

[50] Gupta, H., and Mumick, I. S. Selection of views to materialize under a maintenance cost constraint. In *ICDT* (1999), pp. 453–470.

[51] Halevy, A. Y. Answering queries using views: A survey. *VLDB Journal 10*, 4 (2001), 270–294.

[52] Hasan, R., and Winslett, M. Efficient audit-based compliance for relational data retention. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 238–248.

[53] Fraud Analytics: Combat Credit Card Fraud with Big Data. http://www.intel.com/content/www/us/en/big-data/combat-credit-card-fraud-with-big-data-whitepaper.html, 2013.

[54] Infochimps Data Marketplace. www.infochimps.com/Marketplace.

[55] Incorporated Research Institutions for Seismology. iris.edu/.

[56] Kantere, V., Dash, D., Gratsias, G., and Ailamaki, A. Predicting cost amortization for query services. In *SIGMOD Conf.* (2011).

[57] Kaushik, R., and Ramamurthy, R. Efficient auditing for complex sql queries. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 697–708.

[58] Amazon Kindle. https://kdp.amazon.com/help?topicId=A2JGI9S4FDM39Q.

[59] Koutris, P., Upadhyaya, P., Balazinska, M., Howe, B., and Suciu, D. Query-based data pricing. In *PODS* (2012), pp. 167–178.

[60] Koutris, P., Upadhyaya, P., Balazinska, M., Howe, B., and Suciu, D. Toward practical query pricing with querymarket. In *SIGMOD Conference* (2013), pp. 613–624.

[61] Li, A., Yang, X., Kandula, S., and Zhang, M. CloudCmp: Shopping for a cloud made easy. In *Proc. of HotCloud'10* (2010).

[62] Li, C., Li, D. Y., Miklau, G., and Suciu, D. A theory of pricing private data. In *ICDT* (2013), pp. 33–44.

[63] Li, C., and Miklau, G. Pricing aggregate queries in a data marketplace. In *WebDB* (2012), pp. 19–24.

[64] Lin, B.-R., and Kifer, D. On arbitrage-free pricing for general data queries. *PVLDB 7*, 9 (2014), 757–768.

[65] S. loebman, personal communication.

[66] Large Synoptic Survey Telescope. http://www.lsst.org/.

[67] Marx, V. Biology: The big challenges of big data. *Nature 498*, 7453 (06 2013), 255–260.

[68] The big-data revolution in US health care: Accelerating value and innovation. http://www.mckinsey.com/insights/health_systems_and_services/the_big-data_revolution_in_us_health_care, 2013.

[69] Unleashing the value of advanced analytics in insurance. http://www.mckinsey.com/insights/financial_services/unleashing_the_value_of_advanced_analytics_in_insurance, 2014.

[70] Marine Geoscience Data System (MGDS). http://www.ldeo.columbia.edu/research/marine-geology-geophysics/marine-geoscience-data-system-mgds.

[71] Microsoft SQL Azure. http://microsoft.com/windowsazure/sqlazure/.

[72] MIMIC II. http://physionet.org/mimic2.

[73] Motwani, R., Nabar, S. U., and Thomas, D. Auditing sql queries. In *ICDE* (2008), pp. 287–296.

[74] Moulin, H., and Shenker, S. Strategyproof sharing of submodular costs:budget balance versus efficiency. *Economic Theory 18*, 3 (2001), 511–533.

[75] Microsoft Translator. http://datamarket.azure.com/dataset/bing/microsofttranslator.

[76] Navteq. www.navigation.com.

[77] Ng, C., Parkes, D. C., and Seltzer, M. Strategyproof computing: Systems infrastructures for self-interested parties. In *Proc. of P2PECON Workshop* (June 2003).

[78] Ngan, T.-W. J., Wallach, D. S., and Druschel, P. Enforcing fair sharing of peer-to-peer resources. In *IPTPS Workshop* (Feb. 2003).

[79] Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V. V. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[80] One Bus Away. http://onebusaway.org.

[81] Oracle Event Processing Language Reference. http://docs.oracle.com/cd/E14571_01/apirefs.1111/e14304/overview.htm#i1024819, 2013.

[82] Oracle: Fine Grained Auditing. http://www.oracle.com/technetwork/database/security/index-083815.html, 2013.

[83] Oceanic remote chemical analyzer (ORCA). http://orca.ocean.washington.edu/hoodCanal.html.

[84] Pal, M., and Tardos, E. Group strategyproof mechanisms via primal-dual algorithms. In *FOCS* (2003), pp. 584–593.

[85] Panoramic Survey Telescope & Rapid Response System. http://pann-starrs.ifa.hawaii.edu/public/.

[86] Park, J., and Sandhu, R. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur. 7*, 1 (Feb. 2004), 128–174.

[87] Parkes, D. C. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 2001.

[88] PatientsLikeMe and the FDA Sign Research Collaboration Agreement. http://news.patientslikeme.com/press-release/patientslikeme-and-fda-sign-research-collaboration-agreement, 2013.

[89] PostgreSQL: Audit Triggers. http://wiki.postgresql.org/wiki/Audit_trigger, 2013.

[90] Ports, D. R., Clements, A. T., Zhang, I., Madden, S., and Liskov, B. Transactional consistency and automatic management in an application data cache. In *OSDI* (2010), vol. 10, pp. 1–15.

[91] Ports, D. R., and Grittner, K. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1850–1861.

[92] quandl. www.quandl.com.

[93] Quiané-Ruiz, J.-A., Lamarre, P., Cazalens, S., and Valduriez, P. Managing virtual money for satisfaction and scale up in p2p systems. In *Proc. of DaMaP Workshop* (2008), pp. 67–74.

[94] Ren, Q., Dunham, M. H., and Kumar, V. Semantic caching and query processing. *Knowledge and Data Engineering, IEEE Transactions on 15*, 1 (2003), 192–210.

[95] Salesforce. http://www.salesforce.com/.

[96] Sandholm, T. An implementation of the contract net protocol based on marginal cost calculations. In *Intl. Workshop on Distributed Artificial Intelligence* (1993), pp. 295–308.

[97] Schomm, F., Stahl, F., and Vossen, G. Marketplaces for data: An initial survey. *SIGMOD Rec. 42*, 1 (May 2013), 15–26.

[98] SciDB. http://www.scidb.org/.

[99] Sloan Digital Sky Survey. http://cas.sdss.org.

[100] Socrata. http://www.socrata.com/.

[101] Stonebraker et al. Mariposa: a wide-area distributed database system. *VLDB Journal 5*, 1 (1996), 048–063.

[102] Tang, R., Wu, H., Bao, Z., Bressan, S., and Valduriez, P. The price is right - models and algorithms for pricing data. In *DEXA (2)* (2013), pp. 380–394.

[103] Tannen, V. Provenance for database transformations. In *EDBT* (2010), p. 1.

[104] Teradata Active System Management. http://www.teradata.com/article.aspx?id=1602.

[105] Teregowda, P. B., Urgaonkar, B., and Giles, C. L. Implications of moving to the cloud: A digital libraries perspective. In *Proc. of HotCloud'10* (2010).

[106] Tsakalozos, K., Kllapi, H., Sitaridi, E., Roussopoulos, M., Paparas, D., and Delis, A. Flexible use of cloud resources through profit maximization and price discrimination. In *Proc. of the 27th ICDE Conf.* (2011).

[107] Rate Limiting. https://dev.twitter.com/docs/rate-limiting/1.

[108] Upadhyaya, P., Anderson, N. R., Balazinska, M., Howe, B., Kaushik, R., Ramamurthy, R., and Suciu, D. The power of data use management in action. In *SIGMOD Conference* (2013), pp. 1117–1120.

[109] Upadhyaya, P., Balazinska, M., and Suciu, D. How to price shared optimizations in the cloud. *Proceedings of the VLDB Endowment 5*, 6 (2012), 562–573.

[110] Upadhyaya, P., Balazinska, M., and Suciu, D. Automatic enforcement of data use policies with datalawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 213–225.

[111] Vishnumurthy, V., Chandrakumar, S., and Sirer, E. G. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Proc. of P2PECON Workshop* (June 2003).

[112] Washington State Geospatial Data Archive (WAGDA). http://wagda.lib.washington.edu/.

[113] Waldspurger et al. Spawn: A distributed computational economy. *IEEE Trans. on Software Engineering SE-18*, 2 (Feb. 1992), 103–117.

[114] Wang, R. Y., and Strong, D. M. Beyond accuracy: What data quality means to data consumers. *J. Manage. Inf. Syst. 12*, 4 (Mar. 1996), 5–33.

[115] Wang et al. Distributed systems meet economics: Pricing in the cloud. In *Proc. of HotCloud'10* (2010).

[116] West, D. M. Big data for education: Data mining, data analytics, and web dashboards. *Governance Studies at Brookings* (2012), 1–10.

[117] Windows Azure Marketplace. http://datamarket.azure.com/.

[118] World Bank. https://openknowledge.worldbank.org/terms-of-use.

[119] Wu, E., Diao, Y., and Rizvi, S. High-performance complex event processing over streams. In *SIGMOD Conference* (2006), pp. 407–418.

[120] Xignite. www.xignite.com.

[121] Yelp Display Requirements. http://www.yelp.com/developers/getting_started/display_requirements.