Understanding and Improving Database-Backed Applications

Cong Yan

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

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Alvin Cheung, Chair

Magdalena Balazinska

Dan Suciu

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Understanding and Improving Database-Backed Applications

Cong Yan

Chair of the Supervisory Committee:
Alvin Cheung
Paul G. Allen School of Computer Science & Engineering

From online shopping to social media network, modern web applications are used everywhere in our daily life. These applications are often structured with three tiers: the front-end developed with HTML or JavaScript, the application server developed with object-oriented programming language like Java, Python or Ruby, and the back-end database that accepts SQL queries. Latency is critical for these applications. However, our study shows that many open-source web applications suffer from serious performance issues, with many slow pages as well as pages whose generation time scales superlinearly with the data size. Prior work has been focusing on improving the performance of each tier individually, but is often not enough to reduce the latency to meet the developer's expectation.

In this thesis, I present a different optimization strategy that enables much further improvement of database-backed applications. Rather than looking into each tier separately, I focus on optimizing one tier based on how other tiers process and consume the data. In particular, I describe five projects that implement five examples of such optimization. On the database tier, I present 1) CHESTNUT, a data layout designer that generates a customized data layout based on how the application tier consumes the query result; and 2) CONSTROPT, a query rewrite optimizer that performs query optimization by leveraging the data constraint inferred from the application code. On the application tier, I present 3) QURO, a query reorder compiler that changes the query order to reduce the database locking time when processing transactions; and 4) POWERSTATION, an IDE

plugin to fix performance anti-patterns in the application code which results in slow or redundant database queries. Besides cross-tier information, there is more we can use to improve the application. Then I propose 5) HYPERLOOP, a framework that leverages not only application knowledge but also the developer's insights to make performance tradeoffs. In each project, by looking into the interaction with other tiers and even the developers, I show that many optimizations that may not be viable or effective when optimizing each one tier alone can improve the overall application performance significantly. I show the performance gain brought by these optimizations using real-world applications, then discuss future work in this direction and conclude.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First and most important, I want to express my special thanks to my advisor Alvin Cheung for many things. Alvin offered me an opportunity to continue my Ph.D. when I was having a hard time in MIT and doubting that whether I was able to do research. He has been leading me through this journey, giving me a lot of useful advice but also giving me enough freedom to explore different topics and enough patience to learn different fields. I appreciate his enthusiasm which keeps me optimistic about my work, his straightforward comments which pushes me to think deeply about problems and his good temper that calms me down when I feel upset. I also appreciate the (bad) jokes and out-of-the-box ideas in daily life that Alvin shared with us, as well as some abrupt Chinese words during our English conversation, which give me abundant stories to share with my friends. I cannot help telling them again and again because they are really funny!

I would like to give my sincere gratitude to my collaborator and mentor Shan Lu for her support and encouragement. All of my work in this thesis cannot be done without her advice and feedback. I learned a lot from Shan on how to do software engineering research. Shan has been a role model for me as a female researcher; she shows me how to balance work and life and encourage other female researchers in the field. I would also like thank Yeye He who is my mentor during my internship in Microsoft Research. While Alvin guides me in the database system research, Yeye opens another door into data preparation and data cleaning. I had a lot of fun hacking open source code and using them to build various tools. In addition, I want to thank Sheng Wang, my mentor during my internship in Alibaba. I am deeply impressed by his passion in making his research impactful in the company, through which I obtain great motivation.

I am very much thankful to my collaborators Junwen Yang. I enjoy every moment in our collaboration, where we share everything from ideas, code, to fun TV series to watch. This thesis

certainly would not be possible without you. I am also fortunate to collaborate with Xiangyao Yu, who taught me about transactions back in MIT. We lost touch when I moved to Seattle but started to collaborate after five years, still on transaction-related project. Life is full of surprise!

In addition to Alvin, I would like to thank my "unofficial" advisors and my committee members, Dan Suciu and Magdalena Balazinska. Dan is sharp and wise but very nice at the same time. I am so happy to take his course through which I learned the beautifulness of model theory. I also constantly looking forward to attending 590Q when I can hear Dan's comments on papers, although I am not able to fully understand the content of many theory papers (which are Dan's favorites). I appreciate the feedback from Magda to my various practice talks from which I learned a lot on how to shape a talk and tell an intriguing story. I also appreciate all the great events that Magda organized which gave us rich opportunities to connect with industry and learn about real problems in the world.

Next, my labmates. I received continued help and support from Maaz Ahmad, Walter Cai, Shumo Chu, Maureen Daum, Shana Hutchison, Srini Iyer, Shrainik Jain, Brandon Haynes, Jonathan Leang, Ryan Maas, Laurel Orr, Jennifer Ortiz, Guna Prasaad, Chenglong Wang, Jingjing Wang, Remy Wang. I realized how right the decision was to transfer to University of Washington when I joined the database lab. I only wished I had shared more with you and not being so shy and nervous to begin with. The life of Ph.D. is hard, but not so hard when you guys are around.

I would like to thank many people who have given advice and feedback to my research: Phil Bernstein, Surajit Chaudhuri, Sudipto Das, Christian Konig, Feifei Li, Yue Wang and many others. I also want to thank Samuel Madden, who introduced me to my advisor Alvin and encouraged me to do database research. I am grateful to have friends who shared joy and tear along the journey: Yanling Deng, Guyu Jiang, Jieling Miao, Rongsha Li, Weinong Wang, Haowen Cao, Keyan Gao, Yimo Zhang and many others.

Finally, I would like to thank my parents, who are backing me up no matter what happens. You are the greatest parents in the world. My last but not least acknowledgement goes to Donghao Ren, my boyfriend and best friend. You have made my life joyful and bright.

Chapter 1

INTRODUCTION

From banking to social networking, we interact with database-backed applications on a daily basis. These applications are often organized into three tiers, as shown in Figure 1.1. The front-end is a client or many clients who visit the website. The application is hosted an application server that runs the application logic. The backend is often a database management system (DBMS) which persistently stores the application data. These three tiers are often developed with different languages: the front-end is usually rendering HTML/JavaScript files, the application is often developed with object-oriented programming languages like Java, Python or Ruby, and the database backend accepts SQL queries. When the application runs, the front-end issues requests to the application server. The application processes each request, during which it may issue queries to the database to fetch persistent data and update the persistent data. Then the server processes the fetched data, and organizes the data into a response like a webpage to send to the front-end. The front-end then renders the webpage on a browser. The end-to-end latency, i.e., from the time that a client issues a request to the time she sees the response (e.g., a rendered webpage), is often critical to these applications, since a slow webpage affects user's experience and thus can have an impact on the company's profit.

To improve the end-to-end performance, prior work has focused on optimizing each layer separately. For instance, decades of research has been conducted to optimize SQL queries running on databases; many compiler optimizations has been applied to improve the performance of the object-oriented language runtime; web caching and similar other techniques are used to improve the front-end rendering time. However, for these database-backed applications, the end-to-end performance is often still unsatisfying even after applying as many of these optimizations as possible.

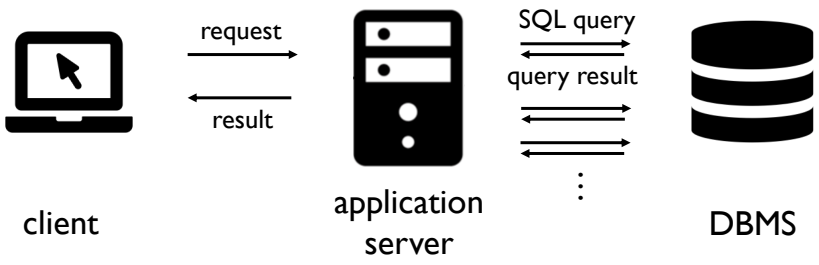


Figure 1.1: Architecture of database-backed applications.

Yet, there are still opportunities for further improvement. Instead of focusing on each tier individually, I look into optimizing one tier based on its interaction with other tiers. These optimizations may not be plausible or beneficial for a particular tier, yet they are able to improve the overall performance. For instance, instead of improving individual SQL queries at the database tier, the queries can be optimized differently based on how the application consumes the query result. Such optimization may not accelerate each query significantly but can reduce the later post-processing cost on the application tier. Another example is to change how the application issues the queries, like changing the order of queries. From the application's perspective, doing so does not reduce any computation on the application side and will not be considered by existing compilers. However, it may change how the database processes the queries and improve query latency on the database tier. Furthermore, information from other tiers may also enable more optimization opportunities. For instance, the dependencies among persistent data expressed in the application logic can be leveraged to optimize database queries as the data is processed by the application and subsequently inserted into the database. While there are many other opportunities, unfortunately, existing tools do not support analyzing all the tiers jointly, and thus unable to implement these optimizations.

In the following, I will first briefly review the architecture of a typical database-backed web application, and present our study which reveals how real-world applications perform. Then I will discuss the intuition behind optimizations leveraging information from other tiers and give a summary of my thesis contribution.

1.1 Architecture of Database-Backed Applications

As mentioned above, database-backed applications include three tiers. Many such applications are developed using object-relational mapping (ORM) framework. Each ORM framework is based on an object-oriented programming language, for instance, Ruby on Rails [55] for Ruby, Django [26] for Python and Hibernate [36] for Java. Rather than embedding SQL queries into the application code, ORM frameworks let developers manipulate persistent data as if it is in-memory objects via APIs exposed by the ORMs. When executed, such API calls are translated by the ORM into queries executed by the database and the query results are serialized into objects returned to the application. Many ORM frameworks also provide APIs to build a template for the front-end which dynamically generates the front-end code. By raising the level of abstraction, this approach allows developers to implement their entire application in a single programming language, thereby enhancing code readability and maintainability.

In this thesis, I focus on one representative type of application, i.e., web applications built using Ruby on Rails (although many optimizations in this thesis can be applied to more general applications). Ruby is among the top 3 popular languages on GitHub [131], and Rails is among the top 3 popular web application frameworks [33]. Many widely used applications are built upon Rails, such as Hulu [39], GitLab [34], Airbnb [3], etc. Compared to other popular ORM frameworks such as Django and Hibernate, Rails has $2\times$ more applications on GitHub with 400 more stars than Django and Hibernate combined. As Rails provides similar functionalities, my work can be easily applied to applications built with other frameworks as well.

Next I will introduce the setting of applications we focus in this thesis, followed by showing how these applications are developed. A detailed setting of Figure 1.1 is shown in Figure 1.2. A Rails application deploys on an application server connected to a database backend, where the database can be hosted on the same or different machine as the server[1]. Internally, the server consists of three components: *model*, *view* and *controller* [140]. Upon receiving a user's request, say to

[1] I mainly focus on a setting with a single-machine application server and a single-machine database, leaving the distributed setting to future work

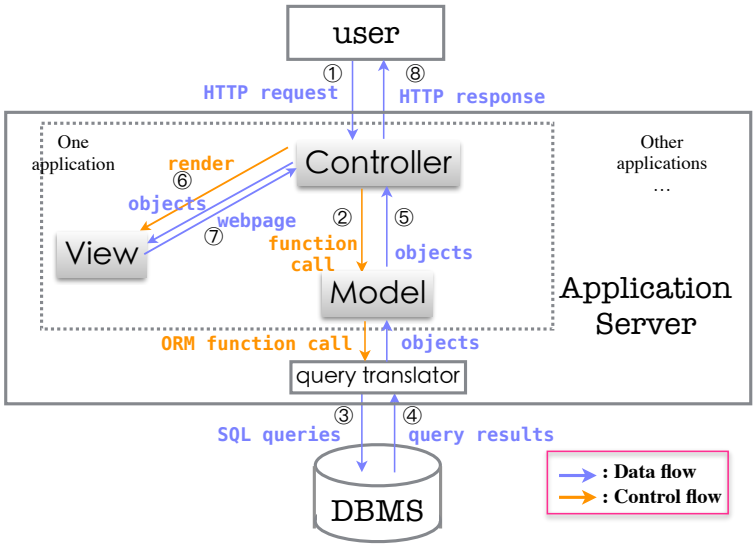


Figure 1.2: A typical architecture of a Rails application.

render a web page, the Rails server invokes the corresponding *action* that resides in the appropriate *controller* based on the routing rules provided by the application. During its execution, the *controller* interacts with the database by invoking ORM functions provided by Rails [49] ②, in which Rails translates into SQL queries ③. The query results ④ are serialized into *model* objects returned to the *controller* ⑤, and subsequently passed to the *view* ⑥ to construct a webpage ⑦ to be returned ⑧.

As an illustration, Figure 1.3 shows the code of a blogging application. While the controller and the model are written in Ruby, the view code is written in a mix of ruby and mark-up languages such as HTML. There are two actions defined in the controller: `index` and `show`. Inside the `index` action, the Rails functions `User.where` and `User.some_blogs` are translated to SQL queries at run time to retrieve blog records from the database system. The retrieved records are then passed to `render` to construct a webpage defined by the view file `blog_index.erb`. This webpage displays the excerpt and the URL link (`link_to`) of every blog. Clicking the link brings the user to a separate page, with another action (`show`) called with the corresponding blog identifier to retrieve the corresponding blog details via another query issued by `Blog.where`.

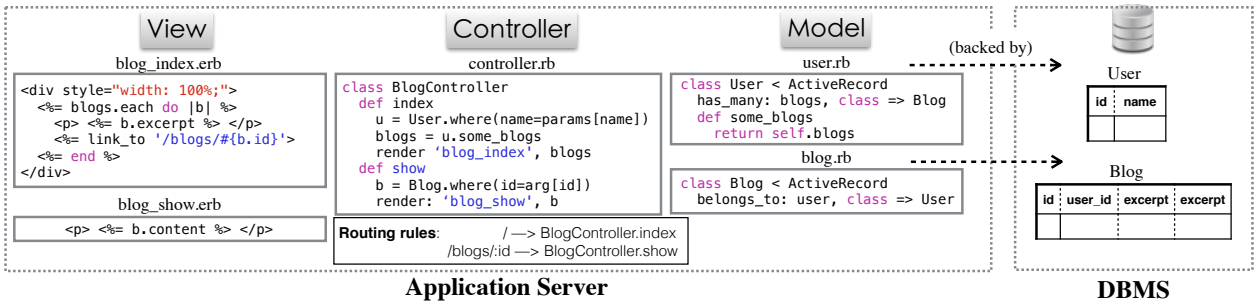


Figure 1.3: An example of blogging application built with Rails.

1.2 Real-world Application Performance

Although using ORM framework increases the development productivity, it often comes at a cost. With the details of query processing hidden, developers often do not understand how their code is translated to queries and how queries are executed. As a matter of fact, the slow performance of ORM frameworks is often complained by developers [10, 11, 12].

To measure application performance, we performed a comprehensive study on 12 popular open-source web applications built with Rails [155]. These applications, as listed in Table 1.1, come from 6 categories. These 6 categories cover almost all Rails applications with more than 100 stars on GitHub. They also cover a variety of database-usage characteristics, such as transaction-heavy (e.g., e-commerce), read-intensive (e.g., social networking), and write-intensive (e.g., forums). We study the top 2 most popular applications in each category, based on the number of "stars" on GitHub[2]. They have been developed for 5 to 12 years and are all in active use, ranging from 7K lines of code (Lobsters [41]) to 145K lines of code (Gitlab [34]).

In this study, we profiled these applications using synthetic data, which is generated according to the data distribution of the actually-hosted website. We synthesize three sets of database contents for each application that contain 200, 2000, and 20,000 records in its main database table, which is the one used in rendering homepage. Other table sizes scale up accordingly following the data

[2]We measured in Nov. 2017 and the statistics may have changed over time.

Table 1.1: Details of the applications chosen in our study

Category	Abbr.	Name	Stars	Commits	Contributors
Forum	Ds	Discourse	21238	22501	568
	Lo	Lobster	1304	1000	48
Collaboration	Gi	Gitlab	19255	49810	1276
	Re	Redmine	2399	13238	6
E-commerce	Sp	Spree	8331	17197	709
	Ro	Ror_ecommerce	1109	1727	21
Task-management	Fu	Fulcrum	1502	697	44
	Tr	Tracks	835	3512	62
Social Network	Da	Diaspora	11183	18734	335
	On	Onebody	1592	1220	6
Map	OS	Openstreetmap	664	8000	112
	FF	Fallingfruit	41	1106	7

distribution statistics discussed above. Under all settings, the database sizes for all applications are smaller than 1GB, ranging from 3MB to 982MB.

While the detailed profiling result can be found in the paper [155], we summarize the highlights as below:

- *All applications have slow and/or super-linearly scaling webpages.* 11 out of 12 applications have pages whose average end-to-end loading time exceeds 2 seconds (we say a webpage is slow if it takes over 2 seconds to load[3]); 6 out of 12 applications have pages that take more than 3 seconds to load. Furthermore, 11 applications have super-linearly scaling webpages

[3] A survey [66] shows that a website user expects a webpage to load within 2 seconds, which we use as a threshold for slow webpage.

(i.e., the end-to-end time scales super-linearly with the size of the database), with an average of 2.8 pages per application.

- *The majority of the webpage time is spent in the server and the database backend.* We break down the end-to-end loading time of the top 10 pages in each application into backend time (i.e., time spent on the application server and the database), client time (i.e., time for loading the webpage in the browser), and network time (i.e., time for data transfer between server and browser). The backend time contributes to at least 40% of the end-to-end-latency for more than half of the top 10 pages in all but one application. Furthermore, over 50% of the slow webpages spend more than 80% of the loading time on server and the database.

Note that we profiled using the most recent version of every application, where the developers have already applied many optimizations like adding indexes to the database, adding data caches, etc. Our study reveals that even after much effort to optimize, serious performance issues still widely exists in these applications.

1.3 Thesis Contributions

Traditional optimizations often optimize a single tier individually, aiming to improve the performance of a particular tier: database query optimizers optimize SQL queries to reduce the query latency; compilers and interpreters optimize the application code to reduce the computation and the memory latency of the application server; front-end libraries and web caches optimize the execution of HTML and JavaScript to reduce the rendering time of a webpage. Many of these optimizations have already been applied to existing applications, yet they still leave behind many slow webpages that greatly affect the user experience. In this thesis, I focus on the optimizations across tiers that bring much further performance gain in addition to traditional ones. Our work jointly analyze multiple tiers to understand how each tier processes the data and consumes the result from other components. We explore how to change one tier to better prepare the requests and data for other tiers for the purpose of improving the overall performance. We also look into what extra information we can obtain from other tiers in order to better carry out the optimizations for one tier. We further

ask, what other information can we leverage, other than the other tiers which are still part of the application? So we also explore how to interact with the developer to understand their insight in order to better customize the optimization for every application. These different perspectives lead to a discovery of many opportunities that are not studied by previous work.

Concretely, I will discuss four projects implementing such optimizations for the database backend and the application server, followed by a system that decides the performance tradeoff using not only the application itself but also the developer's knowledge.

- Ch. 2 introduces CHESTNUT, a data layout designer for the database tier. This work was published in VLDB 2019 [151] and SIGMOD 2020 [138]. CHESTNUT leverages how the query result is being used in the application to customize the in-memory data layout. We observe that the query result is processed in the application server as plain or nested objects, which is different from the tabular representation in the database. When the application runs, the query results are serialized from tabular representation to nested format, where the serialization cost is often the major bottleneck for many slow queries. Instead of storing data in the tabular format in memory, storing them in a nested format reduces the cost of serialization and improves the efficiency of data processing in the application. However, determining what data layout (e.g., how to nest the data and how to build index) to use is non-trivial since the application uses different data nesting for different queries and webpages. CHESTNUT solves this challenge by leveraging recent advances in program analysis, symbolic execution, and solvers to automatically generate a custom data layout and query plans on the data layout. CHESTNUT is able to accelerate the application queries by 3.6× in average (up to 42×) as compared to using the default tabular data layout of relational databases.

- Ch. 3 introduces CONSTROPT, an extension to query optimizer for the database tier. A survey of application constraints motivating CONSTROPT is published in ICSE 2020 [154] and this work is under preparation for submission. CONSTROPT leverages the data constraints defined in the application code to optimize database queries. We observe that many data constraints can be inferred from the application code, e.g., how the application processes the

data before saving it to the database, yet these constraints are not explicitly defined in the database. As a result, the database is not aware of such constraints and unable to leverage them to optimize SQL queries. In response, we propose CONSTROPT to bridge this gap. CONSTROPT infers data constraints through static analysis on the application code, rewrites the application queries and leverages the constraints to verify the correctness of query rewrites. CONSTROPT is able to discover a large number of constraints from the application code. Our evaluation shows that on average over 43 queries can be optimized by CONSTROPT per application where the query performance can be improved by up to 43×.

- Ch. 4 introduces QURO, a query-reordering compiler that works on the application code. This work was published in VLDB 2016 [150]. QURO changes how the application issues database queries to enable faster transaction processing. We observe that for transaction processing systems under locking-based concurrency control, the order of the queries in a transaction has great impact on the system throughput. By moving hotspot accesses (i.e., records that are often accessed by many concurrent transactions) to later in the transaction, the locking period of hotspots is shortened, and the transaction latency is reduced. However, the applications often issues transaction interactively, and the database only sees individual queries without knowing how they are related. So we build QURO that changes the application code to issue queries in different order. By analyzing the application semantics, it ensures that such reordering does not alter the expected transaction behavior. QURO-generated transaction code improves the system throughput by 6.53× compared to using the original code, and reducing the transaction latency by up to 85% on standard benchmarks.

- Ch. 5 introduces POWERSTATION, a compiler that fixes performance anti-patterns. This work was published in CIKM 2017 [152] and FSE 2018 [156]. A performance anti-pattern is a code pattern that leads to slow performance but can be replaced by a more efficient piece of code. In this work we focus on anti-patterns of inefficient data access involving database queries. We first conduct a study on the open-source web applications and summarize the common anti-patterns. We then build POWERSTATION to automatically detect and fix these

anti-patterns. POWERSTATION accelerates the end-to-end page time by over 5×, and 57 of the fixes are accepted by application developers.

- Ch. 6 introduces HYPERLOOP, a framework to take developer's insight and make performance tradeoffs. This work was published at CIDR 2020 [153] and ICSE 2019 [157]. HYPERLOOP relies on the developer's knowledge to decide how to optimize application components. We observe that developers are often willing to make tradeoffs to accelerate the essential part of their application, like accelerating important webpages by slowing down non-important ones, or making design changes to a webpage to make it faster. HYPERLOOP proposes an interface to let developers provide their insight using priority on the webpage element. It then uses such information to drive the optimization and tradeoffs of the entire webpage. With HYPERLOOP, developers only need to interact with the webpage visually to make the tradeoffs. While still under development, preliminary results of HYPERLOOP show that this view-driven approach can substantially improve the performance and user experience.

I will next describe each of the projects in detail in the following chapters. Then in Ch. 7 I will discuss more opportunities in cross-tier optimizations and discuss its extension to dynamic workload and runtime optimizations. Then I will discuss related work in Ch. 8 and conclude in Ch. 9.

Chapter 2

CHESTNUT: GENERATING APPLICATION-SPECIFIC DATA LAYOUTS

In this chapter, we present CHESTNUT, a data layout generator for in-memory object-oriented database applications. Given a memory budget, CHESTNUT generates *customized* in-memory layouts and query plans to answer queries written using a subset of the Rails API, a common framework for building object-oriented database applications. CHESTNUT differs from traditional query optimizers and physical designers in two ways. First, CHESTNUT automatically generates layouts that are customized for the application after analyzing their queries. Second, CHESTNUT uses a novel enumeration and verification-based algorithm to generate query plans that use such data layouts, rather than rule-based approaches as in traditional query optimizers. We evaluated CHESTNUT on four open-source Rails database applications. The result shows that it can reduce average query processing time by over 3.6× (and up to 42×), as compared to other in-memory relational database engines.

2.1 Data Representation Mismatch

As described in Sec. 1.1, database applications are increasingly written using object-oriented programming languages, and use ORM frameworks to manage the persistent data. The idea is that by utilizing relational databases, such object-oriented database applications (OODAs) can leverage relational query optimization techniques to become efficient.

In practice, however, OODAs often exhibit characteristics that make them distinct from traditional transactional processing or analytical applications. In particular:

- **Nested data model.** OODAs often come with objects containing fields of variable-length lists, making data model highly non-relational. While variable-length lists are common in classical

database applications (e.g., storing the list of ordered items in the TPC-C benchmark), OODAs, being object-oriented, makes it very easy to create deep object hierarchies including circular ones. The object hierarchy can easily reach more than 10 levels, with nearly half of the queries returning objects from multiple levels.

• **Many-way materialized joins.** Due to deep object hierarchies, simple object queries can turn into a long chain of multi-way joins when translated into relational queries. Unlike analytical applications where such joins are also common, the join results are directly returned in OODAs rather than aggregated. As an example, a single query can involve as many as 6 joins in the OODAs, and return as much as 5GB of data. This makes the data structures used to store persistent data of crucial importance as standard row or column stores are not the best fit.

• **Serialization cost.** As the database and application represent data in different formats, moving data across them incurs serialization cost. This cost is pronounced in OODAs as queries often return a long list of hierarchical objects which requires converting materialized join result into objects and nested objects. Serialization easily takes longer than retrieving data, as our experiments show.

• **Complex predicates.** OODA queries include many complex predicates, as many frameworks expose methods to filter collections of persistent objects that can be easily chained. As each chained function is translated into query predicate, the final query often contains many (potentially overlapping or redundant) predicates as a result of passing a collection through different method calls. In fact, in our OODA evaluation corpus a single query can involve as many as 40 comparison predicates.

• **Simple writes.** Similar to transactional applications, most writes in OODAs touch very few number of objects (one object for each write query in our evaluation corpus). This makes write optimization (e.g., batch updates) a secondary concern.

The above aspects make OODAs challenging to optimize using standard query processing techniques. In this section, we present an overview of CHESTNUT, our proposed solution to the data representation mismatch problem described above. CHESTNUT is an in-memory data layout[1] and

[1]We use the term "data layout" to refer to both the data being stored (in cases where only a subset of fields in an object and a subset of all objects are stored), and the type of data structure used to store data in memory.

query plan generator for OODAs. (as in object-oriented databases [75, 120]), It leverages recent advances in program analysis, symbolic execution, and solvers to automatically *generate* a custom data layout given application code.

In sum, CHESTNUT makes the following contributions:

• We design an engine that finds the best data layout for OODAs. Our engine does not rely on rule-based translation as classical optimizers. Instead, it analyzes the application code to generate a custom search space for the storage model, and uses a novel enumeration and verification-based algorithm to generate query plans.

• Given the search space, we formulate the search for the best data layout as an ILP problem. We also devise optimization techniques to reduce the problem size such that the ILP can be solved efficiently, with an average of 3 minutes of solving time when deployed on real-world applications.

• We built a prototype of CHESTNUT and evaluated it on four popular, open-source OODAs built with Ruby on Rails framework. Compared to the original deployment using an in-memory version of MySQL, CHESTNUT improves overall application performance by up to 42× (with an average of 6×). It also outperforms popular in-memory databases with an average speedup of 3.9× and 3.6× respectively. CHESTNUT compares favorably even after additional indexes and materialized views are deployed.

2.2 Chestnut *Overview*

In this section we give an overview of CHESTNUT using an example abridged from a project management application Redmine [54]. In this example, two classes are managed persistently, `Project` and `Issue`, with a *one-to-many association* from project to issues, i.e., one `Project` can contain multiple `Issue` instances, but each `Issue` belongs to at most one `Project`. Listing 2.1 shows the definition of two classes and two queries using Rails API [55] supported by CHESTNUT: `Q1` returns `Projects` created later than a parameter `p1` that contain open issues,[2] and `Q2` updates an

[2] "`exists`" is a short-hand expression introduced by CHESTNUT that filters projects based on their associated issues. The query can be written using standard Rails API as `Project.where("created>?",p).joins(issues,"status='open'").distinct`.

issue with `id` equals to `p2` and changes its status to '`close`'.

Listing 2.1: An example application

```
class Project:
  has_many: issues=>Issue # issues is a list of Issue objects, can be
    retrieved as a field of Project
  uint id
  date created
class Issue:
  has_one: project=>Project
  uint id
  string status
Q1 = Project.where(created>? AND exists(issues,status='open'), p1)
Q2 = Issue.update(p2, status:'close')
```

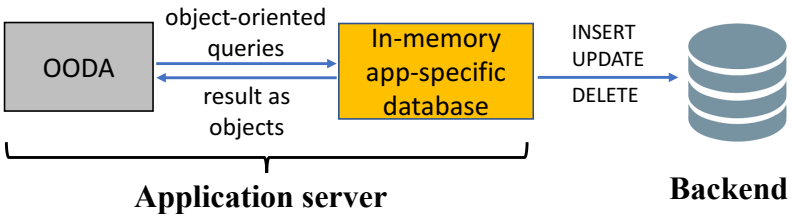


Figure 2.1: Using CHESTNUT's data layout for OODAs.

To use CHESTNUT, developer provides the input OODA code and a memory budget. CHESTNUT then generates the implementation of an in-memory data layout, consisting of data structures to hold data in memory, indexes to speed up queries, and code that leverages the data structures and indexes to answer queries embedded in the OODA. The total size of the data structures and indexes is subject to the provided memory budget. Figure 2.1 shows how the OODA can use the CHESTNUT-generated database. Like other in-memory databases, data is first loaded from persistent

storage to in-memory data layout created by CHESTNUT. Embedded queries are implemented using CHESTNUT-generated code, with the results from writes propagated to persistent storage, and read results returned back to the application as in standard relational databases.

Internally, CHESTNUT searches for different ways to organize in-memory data given the application code. Figure 2.2 shows a few data layouts that CHESTNUT considers for Q1. In (a), the `Projects` and `Issues` are stored in two separate arrays, as in standard relational databases. Given this layout, to answer Q1, we scan the `Projects` array, and repeatedly scan the `Issues` array for each `Project` to find the associated open issues. Indexes can be used to avoid repeated scans. In (b), a B-tree index is created on the `created` field of `Project` to quickly find those newly-created `Projects`, while an index on the foreign key `project_id` of `Issue` can be used to find `Issues` associated with a `Project`.

We test the data layout above with 400K projects and 8M issues (altogether 3.2GB of data). Q1 takes 15s with 3.2GB of memory, and 9s with 3.6GB of memory to finish using (a) and (b), respectively. We can do better, however. Figure 2.2 (c) and (d) show another design that CHESTNUT considers, where (c) stores a nested array of `Issues` within each `Project` object, and the `Projects` are furthermore sorted on `created`. With (c), checking whether an open issue exists in a project can be done by checking the nested objects without scanning a separate `Issues` array. Using this layout, Q1 finishes in 7.4s with 3.2GB of memory. Q1 can run even faster with layout (d), which only stores `Projects` that contain open issues, and sorted by `created`. Using (d), Q1 requires only one range scan on the `Projects` array and finishes in 0.3s using only 0.2GB of memory. Note that as (c) stores nested objects, and (d) keeps only a subset of the `Projects` in memory as determined by Q1's predicate, neither of them would be considered by relational physical designers, but are within CHESTNUT's search space of designs.

We now briefly describe how CHESTNUT finds the best data layout. CHESTNUT first analyzes the OODA code and extracts all query components written with the query APIs. It then enumerates query plans for each individual read query, with each plan using a different data layout. To do so, it first enumerates object nestings, i.e., how associated objects are stored (such as `Projects` and the embedded `Issues` in Figure 2.2). It then generates data structures for each nest level that can be

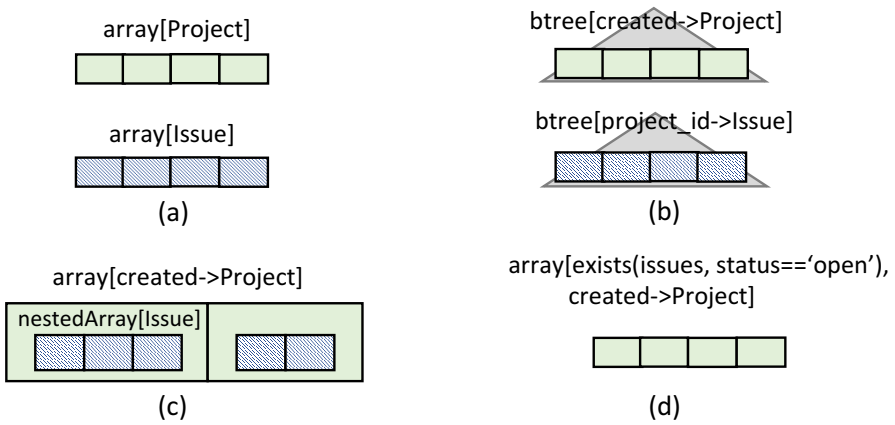


Figure 2.2: Example of data layouts for Q1. A green box shows a `Project` object and a blue shaded box shows an `Issue` object. A triangle shows an index created on the array of objects.

used to answer the query. These object nestings and data structures define the search space of data layouts. To generate query plans, CHESTNUT enumerates over the space of query plans using the data structures. Such enumeration generates many invalid plans, and CHESTNUT verifies each plan against the query and retains only the valid ones. CHESTNUT also uses a few heuristics to prune out plans that are unlikely to be the best.

All data structures enumerated above can be potentially used to answer a read query, which need to be updated properly for writes. CHESTNUT generates plans for each write query on every data structure introduced when enumerating read plans. It first generates "reconnaissance" queries[3] that fetch the objects to be updated as well as other relevant data, and searches plans for these reconnaissance queries by treating them as normal read queries. During this process, new data structures may be introduced and they also need to be updated. CHESTNUT repeats the process until no new data structure is introduced and every seen data structure can be properly updated.

CHESTNUT formulates the problem of finding the data layout that optimizes *overall* OODA query performance as an ILP problem. Solving the ILP tells us which data structure to use for the data layout and the plan chosen for every query. CHESTNUT then generates C++ code that

[3] A reconnaissance query is a read query that performs all the necessary reads to discover the set of objects to be updated given a write query, as introduced in [143].

implements an in-memory application-specific database based on this result.

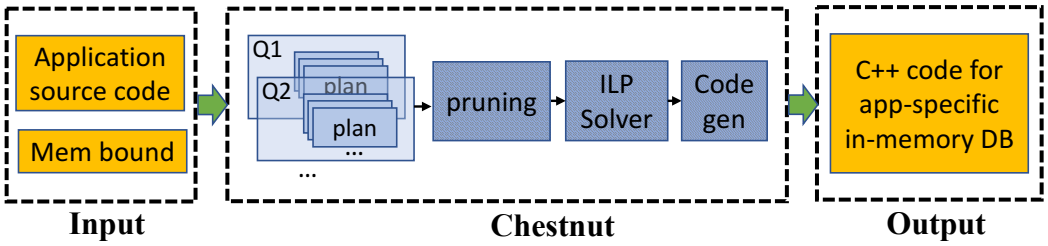


Figure 2.3: Workflow of CHESTNUT

The overall workflow of CHESTNUT is shown in Figure 2.3. We next describe each step in detail.

2.3 Data Layout and Query Plan

In this section we introduce the search space for data layouts and query plans.

2.3.1 Query interface

CHESTNUT supports queries written using a subset of the Rails API [55], including commonly used query operations such as selection, projection, aggregation, etc [2]. As a review, all queries in Rails start with a class name (we refer to it as the "base class"), which by itself returns a list of all persistent objects of that class. Functions can be applied to the returned list, e.g., `where` and `find` along with a predicate, which is similar to relational selection. Join functions (e.g., `includes`, `joins`) are used in Rails queries to retrieve objects and their embedded (i.e., associated) fields, for instance, given the `has_many` association [1] between `Projects` and `Issues`, `Project.includes(:issues)` returns all projects that and their associated issues. Further predicates can be specified to filter the join result, for instance `Q1` shown in Listing 2.1 returns only projects with open issues using the CHESTNUT-provided `exists` shorthand.

When CHESTNUT analyzes the queries in the input OODA, it first rewrites them into a stylized form for easy processing as shown in Listing 2.2. A read query always starts from the base class,

followed by filters, ordering, and aggregation, then retrieval of associated objects that are returned from subqueries. A subquery is similar to a read query except that it can only retrieve associated objects of the base class (e.g., issues that belongs to a project). For write queries, CHESTNUT rewrites them into queries that updates one object one at a time. As discussed in Sec. 2.1, OODAs often update a single object at a time, hence we leave the implementation of batch updates as future work.

Listing 2.2: Stylized CHESTNUT queries with Rails APIs in bold.

```
readQ := Class.where(...).order(..).aggr(...)
               .includes(association, subQ)
writeQ := Class.(insert|update|delete)(id,...)
```

2.3.2 Data layout search space

As mentioned in Sec. 2.2, CHESTNUT searches for the best data layout given a set of queries. A data layout describes how CHESTNUT-managed objects are stored in memory (e.g., as a simple or nested array), including indexes to be added. CHESTNUT customizes the search space for each set of queries. To make search effective, we represent the search space using the language shown below.

```
dataStructure  :=  topArray | nestedArray | index
topArray       :=  array(pred, key->value)
nestedArray    :=  nested_array(pred, key->value, nested)
index          :=  indexType(pred, key->topArray)
value          :=  obj | pointer(topArray) | aggr(obj)
indexType      :=  b-tree | hash
pred           :=  pred∧pred | pred∨pred | ¬pred
                   e==e |...| e in {e,...} | e between [e,e]|
                   exists(associatedObj, pred)
e              :=  constant | parameter | f
key            :=  {} | {f,...}
f              :=  field | (associatedObj.)+.field
```

In the above, `topArray`, `nestedArray`, and `index` are data structures that CHESTNUT enumerates during search. *Top-level arrays* store objects of a single class (e.g., `Project`), pointers to objects in another array,[4] or aggregation result. Top-level arrays can also be sorted. The predicate `pred` can involve associated objects as defined by `has_many` association using `exists` (e.g., `Q1` in Listing 2.1), or associated objects defined by `has_one` as a normal class field (e.g., `Issue.where(project.name=?)`). The field used in the predicate or order key can be the field of object stored in the array or the field of an associated object, as defined by `has_one` association. If the array stores aggregated values, it may store a single aggregated value over a list of objects if `key` is `null` (e.g., `array(status='open',null→count(issues))` stores the count of all open issues), or aggregated values for each group (e.g., `array(status→count(issues))` stores an array of (`status, count`) pair for each status and its count).

Meanwhile, *nested arrays* store values of associated objects as an array within another object. They are the same as top-level arrays except that they store only objects associated with those in another nested or top-level array.

Finally, *indexes* can be added to top-level arrays to map keys to object pointers. CHESTNUT supports B-tree and hash indexes created on all objects of one type, and also *partial indexes* on a subset of objects as determined by a predicate (e.g., all open issues).

As shown above, `topArray`, `nestedArray` and `index` are defined by three parameters: `pred` determines the set of objects to be included in the data structure, `key` is the sorted order of an array or index key, and `value` describes the objects or aggregation values stored. For instance: `topArray(exists(issues, status='open'),created→Project)` is a top-level array that stores `Project` objects containing open issues, and is sorted by the `created` field.

CHESTNUT's data layout is inspired by its query interface. First, because queries often filter objects of one class by examining their associated objects, CHESTNUT supports creating predicates and index/order keys using associated objects. Moreover, because queries often involve associated objects, CHESTNUT supports storing them as nested arrays. As we will see in Sec. 2.7, these

[4] CHESTNUT currently does not support chained pointers.

features allow CHESTNUT to greatly improve the performance of OODAs as compared to traditional physical designers.

2.3.3 Query plan description

Given a data layout, CHESTNUT enumerates query plans to answer each query in the application. As CHESTNUT's data layouts include nested data layouts that are not supported by relational databases, CHESTNUT can no longer use their query plans for query execution. We instead design an intermediate representation to represent CHESTNUT's query plans as shown below. This intermediate representation is designed to describe high-level operations on the data structures such that the cost of a plan can be easily estimated, while facilitating easy C++ code generation.

$plan$:= $(s_{coll})^*$ $(s_{sdu})^*$ $(s_{setv})^*$ `return` v

$coll$:= ds`.scan(`k_1`,`k_2`)` | ds`.lookup(`k`)`

s_{coll} := `for` v `in coll:` $(plan)$

s_{sdu} := `sort(`v`)` | `distinct(`v`)` | v'`=union(`$v_1, ...$`)`

s_{setv} := `if` $expr(v)$ $setv$

$setv$:= v=$expr$ | v`.append(`$expr$`)` |
`ds.insert(`k`,`v`)` | `ds.update(`k`,`v`)` | `ds.delete(`k`,`v`)`

$expr$:= $v + v | v \wedge v | ... | *v | v.f$ | `const`

Each read query plan starts with a `for` loop that looks up from an index or scans an array (s_{coll}, where ds is a data structure), followed by statements (s_{sdu}) that perform sort, distinct, or union, and returns a list of objects or aggregation result. The loop body conditionally appends objects into the result object list or set the value of variables like aggregation result (s_{setv}). Each s_{coll} loop may contain recursive subplans with nested loops that iterates over data structures storing associated objects.

Unlike relational databases that need to convert query results from relations to objects (i.e., deserialization), CHESTNUT's query plan returns objects and nested objects directly. Doing so reduces the overhead of time-consuming deserialization and allows CHESTNUT's query plan to be often faster than similar relational query plans that require deserilization.

Likewise, a write query plan also starts with an index lookup or array scan to find the object to update, followed by modification of each identified object. Updating a nested object can be slower than updating a relational tuple due to the overhead of locating the object. Besides, multiple copies of an object may be stored which makes a write query slower. However, since most updates in OODA are single-object updates, the overhead of slower writes are small compared to the gain from read queries, as we will see in Sec. 2.7.

2.4 Query Plan Generation

We now discuss how CHESTNUT enumerates data layouts and query plans efficiently and finds the optimal one for each read query. After determining the best data layout, we discuss finding query plans for write queries in Sec. 2.5.

2.4.1 Enumerate data layouts

CHESTNUT first enumerates data layouts that can be used to answer a read query Q. This includes enumerating object representations, i.e., whether objects of one class are stored as top-level or nested objects, as well as data structures to be used. Each design is described using the language described in Sec. 2.3.3. Unlike traditional physical designers, CHESTNUT only enumerates data structures involving objects and predicates used in Q. As we will see, doing so greatly reduces the size of the search space.

The algorithm is shown in Algorithm 1. It takes Q as input, the class `C` to generate designs for, and optionally a design ds_{upper} where `C` is to be nested within (`C` is initially set to be the base class of Q and ds_{upper} is set to null). It first enumerates the data structures to store `C` and saves them in DS_c. On line 4, it enumerates `C`'s fields used in Q's predicates as the sort or index key for the data structure to be created, e.g., `created` in Q1, and on line 5 the predicates in Q that uses `C` and does not use input parameters, e.g., `exists(issues,status='open')` in `Q1`'s predicates from Listing 2.1. The data structures are then enumerated. If `C` is not to be nested in another data structure (line 6), it then creates a top-level array using the key and predicate (line 7), along with indexes on that array (line 8-10). Otherwise, `C` is nested in ds_{upper} (line 12).

Algorithm 1 Object nesting enumeration

```
1:  procedure ENUMERATEDS(Q,C,ds_upper=null)
2:      models←[] // data layouts to be returned
3:      DS_c ←[] // data structures to store objects of C
4:      for key∈ {scalar fields of C used in Q} do
5:          for pred∈{partial pred in Q with no param} do
6:              if ds_upper is null then
7:                  array← array(pred,key,C)
8:                  DS_c.add(array)
9:                  DS_c.add(B-tree(pred,key,ptr( array ))
10:                 DS_c.add(hash(pred,key,ptr( array ))
11:             else
12:                 DS_c.add(array(pred,key,C,ds_upper))
13:     for a∈{C's associated objects involved in Q} do
14:         Q_a ← sub-query/sub-predicate on a
15:         for an ∈ EnumerateDS(Q_a,ds) do
16:             for ds∈DS_c do
17:                 ds.addNested(an))
18:             models.add(DS_c)
19:             for ds∈DS_c do
20:                 ds.addNested(an.replace(value,ptr))
21:             models.add(DS_c)
22:         for an ∈ EnumerateDS(Q_a) do
23:             models.add(an.addNested(DS_c))
24:     return models
```

After enumerating the data structures to store C, we enumerate the structures to store any associated class A of C that is used in Q. CHESTNUT considers three ways to store A, which we illustrate using the Project-Issue association in Q1:

1. Store Issues as a nested array within each Project. Issues of one project can then be retrieved by visiting these nested objects.

2. Same as 1. but store pointers to a separate array of Issues. In this case issues of a project are retrieved through pointers.

3. Store Issues as a top-level array, and keep a copy of project as a nested object within each Issue. Issues of one project p are then retrieved by visiting top-level Issues and finding those whose nested project matches p.

These three designs are generated from lines 15 to 22 in Algorithm 1. We iterate through the different data structures used to store C and combine that with data structures storing A. For every associated class A, the algorithm first extracts the subquery or the subquery's predicate from Q that uses A, e.g., the "status='open'" predicate on the associated Issues. It calls EnumerateDS using the sub-query or the predicate to recursively enumerate the data layouts for A and A's associated objects, and merges A into ds (i.e., the data structure that stores C) using one of the three ways mentioned above. The final layout is created by either nesting the associated objects (line 17), nesting pointers (line 19), or nesting the parent objects within the associated ones (line 22).

2.4.2 Enumerate query plans

Given the data layouts, CHESTNUT next enumerates query plans for them. These plans are represented using the intermediate language described in Sec. 2.3.3. Because of the huge variety of data layouts that can be generated for each OODA, CHESTNUT does not use rules to generate correct query plans as in traditional query optimizers. Instead, it enumerates as many plans as possible up to a bound, and verifies if any of them is correct, i.e., returning the desired result as the query asks for.

Suppose we have a query Q with no associations written as Class.where(pred).order(f,...) as described in Sec. 2.3.3, and are given a storage model containing a list of data structures $\{ds_0, ds_1, \ldots\}$ storing objects of Q's base class C. CHESTNUT uses a skeleton that defines the

"shape" of a query plan, and enumerates only plans whose number of statements is smaller than the number of disjunctions in Q's predicate.[5]

The skeleton is shown as below:

```
plan  :=  (op)+ (v'=union(v1,v2,...))?
          (distinct(v'))? (sort(v'))?
op    :=  for o in ds.scan/lookup(params):
          (if pred_r v.append(o))
```

The skeleton consists of several operations (op) followed by optional operations that merge and process the results from previous operations: `union`, `distinct`, or `sort`. Each op performs a scan or lookup on a data structure ds, and evaluates a predicate $pred_r$ over the input objects. The skeleton is instantiated by filling in the data structure ds, the predicate $pred_r$, and the number of operations op given the generated storage models.

If Q involves no associations, then all plans are enumerated by the above. If not, CHESTNUT enumerates sub-plans pl_a to answer the predicate on the associated objects using the same process as above, and merges the sub-plans into the loop body of pl to produce the final nested-loop query plan for Q.

2.4.3 Verify query plans

We now have two ways to represent a query: as expressed using the Rails API as described in Sec. 2.3.3, or as one of the CHESTNUT-generated query plans expressed using the intermediate language described in Sec. 2.3.3. Our goal is to check the CHESTNUT-enumerated query plans to see which ones are semantically equivalent to the input, i.e., they return the same results.

Testing can be used to check for plan equivalence: we generate random objects, pass them as inputs to the two plans, execute the plans (using Rails for the input query, or as a C++ program for the CHESTNUT-generated plan), and check if the two plans return the same results. However, this is costly given the large number of plans enumerated by CHESTNUT. CHESTNUT instead leverages

[5]The idea is that each loop generated by op answers one one clause in the disjunction, and objects from different clauses are then unioned to generate the final result. If Q is a conjunctive query, then one loop would be sufficient to retrieve all objects.

recent advances in symbolic reasoning [146, 97, 96] for verification. This is done by feeding the two plans with a fixed number (currently 4) of *symbolic* objects but with unknown contents. Executing the plans with symbolic objects will return symbolic objects, but along with a number of *constraints*. For instance, given the following CHESTNUT-generated query plan that scans through an array of projects:

```
for p in projects.scan:
  if p.created < '1/1':
    v.append(p)
```

Executing this plan with an array of `projects` consisting of two symbolic objects `projects[0] = p0` with `p0 = {created=c0}` and `projects[1] = p1` with `p1 = {created=c1}` where the creation dates `c0` and `c1` are unknown will return the following constraints:

```
if c0 < '1/1' && c1 < '1/1':
  v[0] = p0  v[1] = p1
else if c0 >= '1/1' && c1 < '1/1':
  v[0] = p1  v[1] = null
else if c0 < '1/1' && c1 >= '1/1':
  v[0] = p0  v[1] = null
else # both projects are created >= '1/1':
  v[0] = null  v[1] = null
```

While we do not know the concrete contents of `v` after execution, we know the constraints that describe what its contents should be given the input. We execute the Rails and CHESTNUT-generated plan to generate such constraints, send them to a solver [70] to check for equivalence under all possible values of `c1` and `c2`, and retain only those plans that are provably equivalent. Checking such constraints can be done efficiently: in our evaluation it takes less than 1s to check when fewer than two associations involved.

2.4.4 An illustrative example

We now use Q1 in Listing 2.1 to illustrate query plan enumeration, where it returns the `Projects` created later than a parameter `param` that contain open `Issues`.

CHESTNUT first enumerates the data structures to construct data layouts as described in Sec. 2.4.1. Figure 2.4(a) shows three data layouts. In the first one, `Issues` are nested within each `Project`. A few data structures to store `Projects` are shown from ds_1 to ds_3. The keys of a data structure are fields used in Q1 like `created` in ds_1, and the predicates are partial predicates from Q without parameters such as `exists(issue,status='open')` in ds_1. Different combinations of keys, predicates, and data structure types are enumerated. CHESTNUT similarly enumerates nested data structures to store `Issues` like ds_4 and ds_5.

Other data layouts store the association between `Projects` and `Issues` differently from layout 1. In layout 2, `Projects` store nested `Issue` pointers that point to a top-level `Issue` array ds_7, while in layout 3, top-level `Issues` store nested `Projects`. These three layouts shows the three ways to store associated class as described in Sec. 2.4.1. In layouts 2 and 3, CHESTNUT also enumerates data structures using different keys, predicate and data structure types, with some of them are shown in Figure 2.4(a).

After the data layouts are enumerated, CHESTNUT then generates query plans as described in Sec. 2.4.2. Figure 2.4(b) shows three examples of the enumerated plans. Since Q1's predicate has no disjunctions, CHESTNUT enumerates plans using the skeleton with up to one op, and fills op with different ds and $pred_r$. In (1), the skeleton is filled with ds_0 (the `Project` array that stores all `Projects`), and $pred_r$ as `p.created>param`. In (2), ds_1 and `true` are used. Both (1) and (2) are complete plans as $pred_r$s does not involve associated objects. In (3) however, the predicate `exists(issues, status='open')` involves associated issues (as shown in ①), so CHESTNUT generates a sub-plan that stores the nested `Issues` to fill a skeleton, as shown in ②. The sub-plan is then merged into the loop of ① to form a complete plan.

Each plan is then verified against Q1 as described in Sec. 2.4.3. The solver determines that plan 1 is invalid and discarded (as it does not evaluate the partial predicate `exists(issues,`

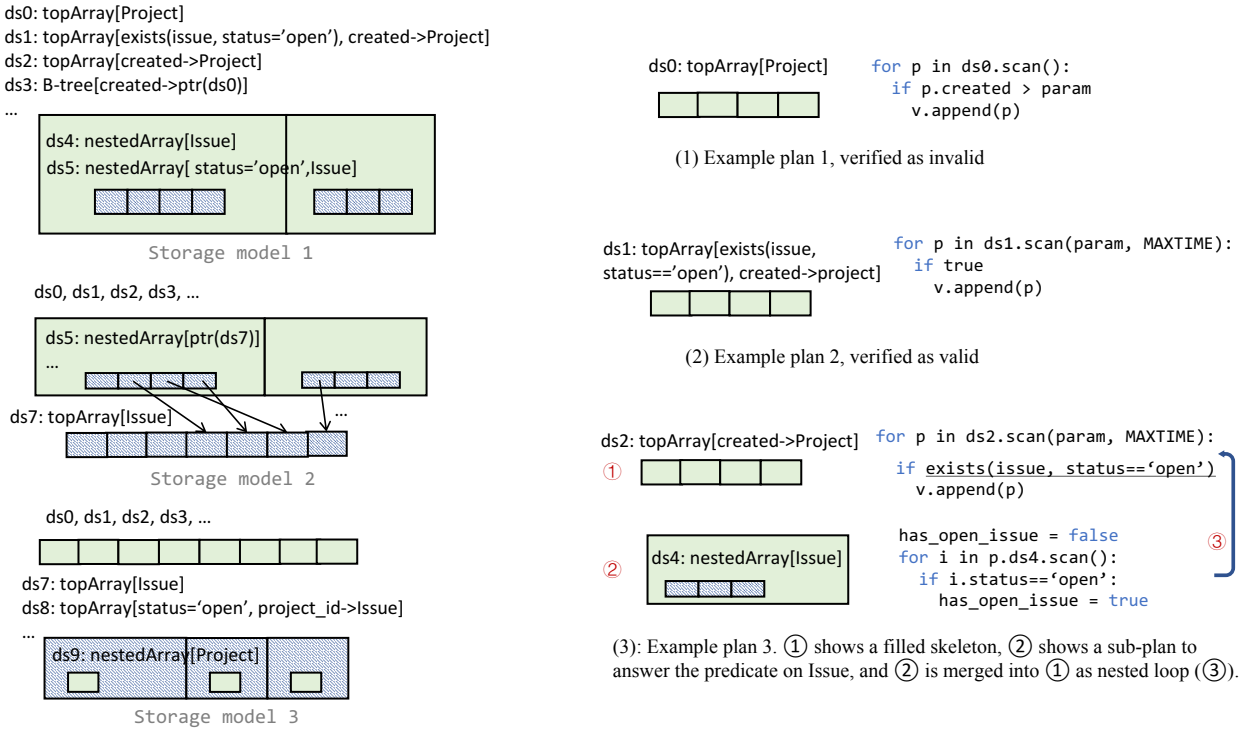


Figure 2.4: Example of how CHESTNUT enumerates data structures and plans for Q1. A green box shows a `Project` and a blue box an `Issue`.

`status='open'`) in Q1), while plan 2 and 3 are valid and retained.

2.4.5 Plan cost estimation

To find the optimal plan, CHESTNUT estimates the execution time of a plan using a cost model, and assumes that predicate selectivities are provided. The plan cost is then estimated as the number objects visited in the plan. For instance, the cost of the plan 1 shown in Figure 2.4(b) is $N_{projects}$ (i.e., number of `Projects`) as each `Project` is visited once when scanning ds_0; the cost of plan 3 is $N_{projects} * N_{issue\_per\_project}/2$ as the top-level loop visits $N_{projects}/2$ `Projects` (assuming the value of `created` is evenly distributed), and each nested loop visits $N_{issue\_per\_project}$ nested issues. Other cost models can be used with CHESTNUT as well.

2.4.6 Reducing search space size

CHESTNUT's enumeration algorithm produces a large number of query plans for each query. CHESTNUT uses three heuristics to prune away plans that are unlikely to be the optimal ones.

First, as mentioned in Sec. 2.4.1, CHESTNUT only enumerates layouts involving classes used in the queries. A layout storing an associated class `A` as top-level and `C` nested in `A` (e.g., layout 3 in Figure 2.4(a)) is discarded if the association is only used in one direction in the application, for instance, queries only retrieve `Issues` given `Projects` but not vice versa. Furthermore, if no query retrieves the associated objects directly, then any layout that stores them as top-level arrays (like layout 2 in Figure 2.4(a)) is also dropped.

Second, CHESTNUT drops plans that are subset of others. For example, if p performs the same scans on the same data structure ds twice while there exists another plan p' that only scans ds once, then p is redundant and is dropped.

Third, if the plans for one query have the same set of shared data structures, then CHESTNUT only keeps the best ones in terms of either least memory consumption or plan execution time. To do so, CHESTNUT groups the plans enumerated for one query by the set of data structures they share with other queries. For each group, CHESTNUT drops all plans other than those that has minimum execution time or uses a data layout with minimum memory cost.

These pruning heuristics can greatly reduce CHESTNUT's execution time, as we will discuss in Sec. 2.7.6.

2.5 Handling Write Queries

After enumerating plans for read queries, CHESTNUT generates plans for writes. As mentioned in Sec. 2.1, OODAs often make individual object updates, hence CHESTNUT currently supports single-object writes (i.e., insert, delete, or update), and translates multi-object writes into a sequence of single-object writes.

To generate plans for writes, CHESTNUT first executes "reconnaissance" read queries [143] to retrieve the objects that are updated. After that, CHESTNUT enumerates plans for these reconnaissance

queries, whose results are then used to generate a write plan.

2.5.1 Generating reconnaissance queries

Algorithm 2 Generate plan for Q_w to update ds

```
1:  procedure GENWRITEPLAN(Q_w, ds)
2:      plan←[]
3:      if not isAffected(ds) then
4:          return
5:      Qw1← ds.class.where(..., Q_w.id)
6:      Qw2← ds.class.where(..., obj.id)
7:      plan.add_stmt((EnumeratePlan(Qw1))
8:      plan.add_stmt((EnumeratePlan(Qw2))
9:      objs←plan.result
10:     for o ∈ objs do
11:         o.update()
12:         ds.delete(o)
13:         (k, pred, v)←(ds.key(o),ds.pred(o),ds.value(o))
14:         plan.add_stmt(if pred ds.insert(k,v))
15:     return plan
```

Given a write query Q_w on object o of class `C` and a data structure ds, ds will be updated if ds's predicate, key, or value involves `C`, or ds contains nested object of class `C`. For example, if an `Issue` is updated, `Issue` arrays and indexes, data structures whose key uses `Issue`'s field and arrays containing nested `Issue` will be updated.

CHESTNUT generates up to two reconnaissance queries for each ds. The first retrieves the affected object in ds using o's id (denoted as o_{ds}), and the second retrieves associated objects of o_{ds} to compute the new key and predicate of o_{ds}. We use Q2 updating `ds1` as shown in Sec. 2.2 to illustrate. The first query (`Qw1` below) retrieves the project in `ds1` containing the issue to be updated.

The second one (Qw2) retrieves that project's issues to recompute if that project contains any other open issues.

Listing 2.3: Read query to find the project of a deleted issue

```
Qw1=Project.where(exists(issues, id=?))
Qw2=Project.where(id=?).includes(issues)
```

2.5.2 Generate plans for write query

CHESTNUT then uses the same search procedure described in Sec. 2.4 to find plans for the read queries generated, and constructs a write plan using the read results.

The algorithm to generate write plans is shown in Algorithm 2. Lines 5-6 generate the two read queries as described above. It then finds plans for these queries using the same process shown in Sec. 2.4 (line 7-8). These read queries find all the entry objects in ds that need to be updated. For each entry object, it first deletes the entry from ds (line 12), updates the entry and recomputes the key, predicate and value (line 13), and reinserts it into ds (line 14).

CHESTNUT first generates an update plan for every ds enumerated for all input read queries. In this process it generates new reconnaissance queries, whose data layouts may contain new data structures not used by any input read query, thus do not have corresponding update plans yet. CHESTNUT iteratively produces write plans to update these new data structures until no new data structures and all write plans are added. In practice, this process converges quickly, usually within just a few rounds.

2.6 Finding Shared Data Structures

The above describes CHESTNUT's data layout and plan enumeration for all queries in the OODA. By formulating into an integer linear programming (ILP) problem, we now discuss how CHESTNUT selects the best data layout from them by trading off between query performance and memory usage.

The ILP consists of the following binary variables:

- each data structure is assigned a variable ds_i to indicate whether it is used in the chosen data layout.
- if ds_i stores an object of class C, then each of C's field f is assigned a variable $f_{ds_i[f]}$ to indicate if f is stored in ds_i (recall that a data structure may store only a subset of an object's fields).
- each plan for read query i is assigned a variable p_{ij} to indicate if plan j is used for query i.
- each plan for write query i updating a data structure ds is assigned a variable p^w_{ijd} to indicate if plan j is used to update the data structure ds.

CHESTNUT estimates the memory cost of each data structure ds_i, denoted as C^{ds}_i, as well as the execution time of each query plan C^{ds}_{ij} or C^{p}_{ijd} (for read or write plans, respectively) as described in Sec. 2.4.5. It also calculates the size of field f as $F^{ds[f]}$, and estimates the number of elements in ds_i as N^{ds}_i.

CHESTNUT then adds the following constraints to the ILP:

- Each read query uses only one plan: $\sum_j p_{ij} = 1$
- Each write query uses only one plan: $\sum_j p^w_{ijd} = 1$.
- Plan p_{ij} uses all the data structures $ds_1, \ldots, ds_N$ in its plan. Similar constraints are created for write query plans: $p_{ij} \rightarrow ds_1 \wedge \ldots \wedge ds_N$.
- Plan p_{ij} uses object fields $f_{ds[t_1]}, \ldots, f_{ds[t_N]}$ in array ds in the plan. A field is used when the plan has a s_{setv} statement that evaluates a predicate, computes an aggregation that involves that field, or adds an object to the result set where the field is projected. Similar constraints are created for write query plans: $p_{ij} \rightarrow f_{ds[t_1]} \wedge \ldots \wedge f_{ds[t_N]}$.
- If a data structure ds_k is affected by a write query Q_j, then at least one update plan should be used: $ds_k \rightarrow \wedge_j(\vee_i P^w_{ijk})$.
- The total memory cost of used data structures and object fields should be smaller than the user-provided bound M: $\sum_i D_i * C^D_i + \sum_{ij} N^D_i * F^{f_j}_{D_i} \leq M$.

The objective is then to minimize the total execution time of all queries:

$$(\sum_i \sum_j C^P_{ij} P_{ij} w_i) + (\sum_i \sum_d \sum_j D_d C^P_{ijd} P^w_{ijd} w_i)$$

where the sums are the total execution time off all read and write queries, respectively. Each query i

is associated with a weight w_i to reflect its execution frequency in the OODA, which can be provided by developers or collected by a profiler.

We use the example shown in Figure 2.4 to illustrate. Assume the application contains `Q1` and `Q2` shown in Listing 2.1. To demonstrate, we only show two plans for `Q1` (shown in Figure 2.4(b)) and a subset of data structures enumerated by CHESTNUT (ds1, ds2 and ds3). The following lists a subset of the ILP constraints, while the rest involving other plans and data structures are constructed similarly.

- Q1 uses only one plan: $p_{11} + p_{12} + ... = 1$.
- Q2 uses only one plan to update ds1: $p^w_{211} + p^w_{212} + .. = 1$.
- Q1's plan1 uses ds1: $p_{11} \rightarrow ds_1$.
- Q1's plan2 uses ds2 and ds3: $p_{12} \rightarrow ds_2 \wedge ds_3$.
- Q1's plan2 uses fields `created` in ds2 and `status` in ds3: $p_{12} \rightarrow f_{ds_2[created]} \wedge f_{ds_3[status]}$.
- ds1 is updated by Q2: $ds_1 \rightarrow p^w_{211} \vee ....$
- ds3 is updated by Q2: $ds_3 \rightarrow P^w_{231} \vee ....$
- Total memory consumed does not exceed bound: $ds_1 * C^{ds}_1 + ds_2 * C^{ds}_2 + ... + f_{ds_2[created]} * N^{ds}_2 + f_{ds_3[status]} * N^{ds}_3 ... \leq M$.
- **Goal**: $\min(p_{11} * C^p_{11} * w_1 + p_{12} * C^p_{12} * w_1 + ...)$

The ILP's solution sets a subset of variables of ds_i, $f_{ds_i[f]}$, p_{ij}, p^w_{ijd} to 1 to indicate the data structures and plans to use, along with a subset of fields to store in each data structure. CHESTNUT then generates C++ code for the chosen plans and data structures using the STL library and Stx-btree library [61]. For query plans, it translates each IR statement in the query plan into C++.

2.7 Evaluation

CHESTNUT is implemented in Python with gurobi [35] as the ILP solver, and uses the static analyzer described in [152] to collect the object queries. We evaluate CHESTNUT using open-source Rails OODAs.

2.7.1 Experiment setup

Application corpus: Similar to prior work [155], we categorize the top-100 starred Rails applications on GitHub into 6 categories based on their tags. We then pick 4 categories, project management, chatting service, forum, and web scraping, and pick one of the top-2 applications from each category as our evaluation corpus. All of the four chosen applications are professionally developed. They are selected as "trending" Rails applications [53, 63, 16], and widely used by companies, individual users [69, 68] and prior research [106, 156].

- *Kandan* [40] is an online chatting application structured with classes `User`, `Channel`, `Activity`, etc. `Users` can send messages in `Channels`. Each `Channel` lists `Activitys`, where an activity is often a message. Queries in this application contain simple predicates, but retrieve a deep hierarchy of nested objects. For example, Kandan's homepage lists channels with their activities and creators of each activity. The corresponding data is retrieved using a query that returns a list of `Channels` with nested `Activitys`, where each `Activity` contains nested `Users`.

- *Redmine* [54] is a collaboration platform like GitHub structured around `Project`, `Issue`, etc. Each project belongs to a `Module` and contains properties like issue tracking. The `Tracker` class is used to manage what can be tracked in a project, such as issue updates, and has a many-to-many relationship with `Projects`. This relationship is maintained by a mapping table with two columns, `project_id` and `tracker_id`. Such associations result in many-way materialized joins as well as complex predicates predicates to retrieve the associated objects. In addition, Redmine's queries also use disjunctions extensively. For instance, it uses nested sets [43] as trees to organize each project and its children projects. A query that retrieves projects in a subtree with ID within the range (`p1`, `p2`) is done using the predicate `left>=p1 OR right<=p2`, where `left`, `right` `Project`'s fields. Such range predicates are often combined with others and are difficult for relational databases to optimize well. We will discuss this later in the evaluation.

- *Lobsters* [41] is a forum application like Hacker News. Persistently stored classes include `User`, `Story`, `Tag`, etc. Users share URLs as stories and add tags to them. Lobsters has similar query patterns as *Redmine*, with many-to-many associations. For example, a `Story` has many `Tags`, and

the same `Tag` can be added to many `Storys`. As a result, many queries are multi-way joins.

• *Huginn* [38] is a system to build agents to perform automated web scraping tasks. It persistently stores `Agents`, each with a set of `Delayed_jobs` to automatically run in the backend to watch these `Agents`, and records `Events` when any update happens. Its queries retrieve a hierarchy of nested objects, as well as aggregations to render the current state of an `Agent` in various ways.

Table 2.1: Application statistics

App	# classes	# read query	# write query
Kandan	6	10	6
Redmine	12	24	6
Lobsters	7	26	10
Huginn	6	16	7

Table 2.2: Comparison of runtime memory cost

App	data size	PostgreSQL	CHESTNUT
Kandan	5.1GB	7.2GB	5.8GB
Redmine	5.9GB	8.3GB	7.4GB
Lobsters	9.8GB	14.5GB	13.7GB
Huginn	6.0G	9.8G	9.0G

We select the top-10 most popular pages from each application. These pages are chosen by running a crawler from prior work [155]. The crawler starts from the application's homepage and randomly clicks on links or fills forms on a current page to go to a next page. This random-click/fill process is repeated for 5 minutes, and we collect the top 10 mostly-visited pages. Table 2.1 shows the total number of classes and distinct read and write queries involved in the popular pages. We collect the queries executed when generating these pages from the SQL query log, and trace back to the object queries defined in the application. These object queries then serve as input workload to CHESTNUT.

We populate the application's database with synthetic data using the methodology described in [155]. Specifically, we collect real-world statistics of each application based from its public website or similar website to scale data properly as well as to synthesize database contents and application-specific constraints. We scale the size of the application data to be 5GB-10GB, which is close to the size of data reported by the the application deployers [17, 18].

Baseline database engines: We compare the CHESTNUT-generated database with relational in-memory databases, including MySQL, PostgreSQL and a commercial database.

• **MySQL** (version 5.7). The original applications use MySQL as the default backend database, so we use the same setting as the baseline and add the same indexes that the developers specified in the application.

• **PostgreSQL** (version 12). We use an indexing tool [22] to analyze the query log and automatically add more indexes besides the ones that come with the application. We set the buffer pool size for MySQL and PostgreSQL to be larger than the total of data and indexes (20GB) such that data stays in memory as much as possible. We use the default value for other settings.

• **System X**. We further use a commercial high-performance in-memory column-store database as the third comparison target. We add the same set of indexes we use in PostgresSQL as suggested by the automatic indexing tool.

• Chestnut. When using CHESTNUT, we measure the actual total size of tables and indexes used by PostgreSQL and set it as the memory bound for CHESTNUT. We replace the database engine and Rails' serialization code with CHESTNUT, and connect to a backend MySQL database to persist data.

We measure the time from issuing the query until the result converted into Ruby objects as the query time. Hence query time includes both **data-retrieving time**, i.e., the time to execute the queries, and **deserialization time**, i.e., the time to convert query results into ruby objects. For relational databases, Rails deserializes the query result from relational tables into (nested) Ruby objects. CHESTNUT's query plans return results as C++ (nested) objects, and uses protocol buffer [47] to convert them into Ruby objects. All evaluations are performed on a server with 128 2.8GHz processors and 1056GB memory. In our current prototype CHESTNUT does not support transactions, so we run queries sequentially and measure their latency.

2.7.2 Performance comparison

Figure 2.5 shows the performance comparison across OODAs. We focus on slow queries takes longer than 100ms to execute in the original application setting as they are often performance bottlenecks. To get a better understanding of such queries, we show the breakdown of data retrieval and deserialization time in Figure 2.5. We show only the summary, i.e., the min, max and mean

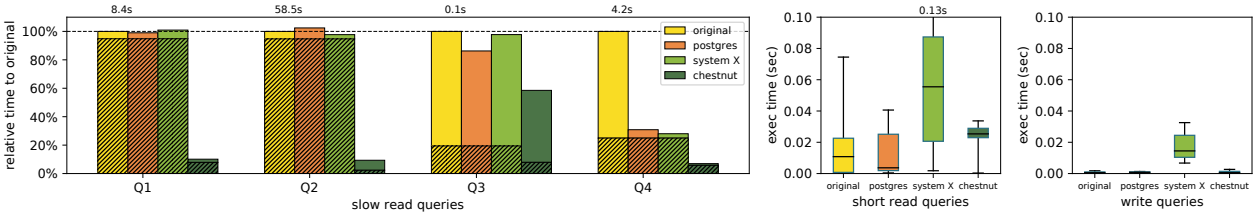
times of the remaining read and write queries as they execute quickly.

The results show that using the automatic indexing tool improves the performance for half of the slow queries by up to 163×. Interestingly, unlike OLAP queries where in-memory column stores can substantially accelerate queries, System X does not achieve similar speedup for OODA's queries. As discussed in Sec. 2.1, this is because query results in OODAs are often not aggregated but returned as lists of objects where all columns are projected. Rather than speeding up queries, columnar store instead adds the non-trivial overhead of row materialization. Compared to other relational database engines, CHESTNUT shows better performance in all slow queries, with an average speedup of 6×, 3.9×, and 3.6× against the MySQL, PostgreSQL, and System X respectively.

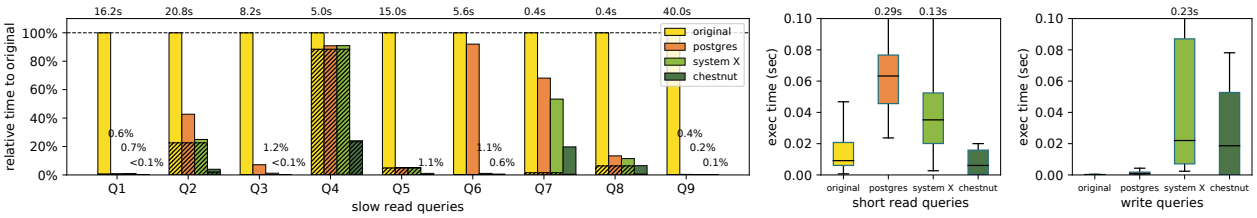
The memory consumption comparison is shown in Table 2.2. It shows the size of the application data and the maximum runtime memory used by PostgreSQL (e.g., tables, indexes and intermediate tables) and CHESTNUT's engine (e.g., data layout). Despite using smaller amount of memory than PostgreSQL, CHESTNUT still achieves significant performance gain in comparison, as shown in Figure 2.5.

We next present case studies to analyze why CHESTNUT's data layout delivers better performance on OODA's queries.

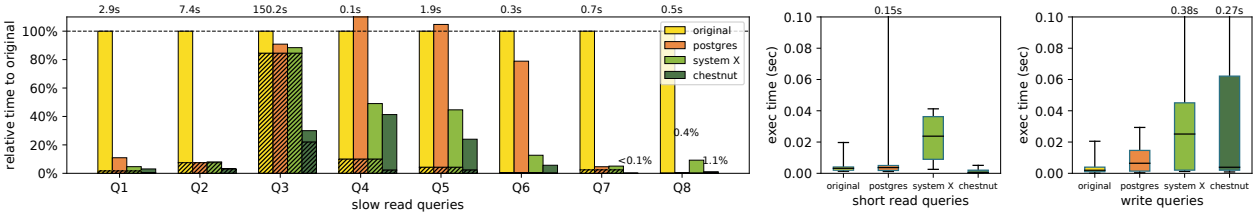
Case 1: Kandan-Q1. This query, shown in Figure 2.6(a), retrieves the first 50 `Channels` ordered by its `id`, where each `Channel` includes its associated `Activitys` and each `Activity` includes its associated `User`. The Rails-generated SQL queries are shown in Figure 2.6(b). They first select from the `channel` table, then use the foreign key `channel_id` to select from the `activity` table, followed by a third selection from the `user` table using the values of the `user_id` column in the second query result as parameter. The cost of combining the tuples from these tables to generate the final query results is prohibitively expensive. Rails would 1) create `Channel` objects from the result of the first query, 2) create `Activity` objects from the second query, 3) group `Activity` by `channel_id` and insert into `Channel` as nested object, 4) similarly create nested `User` object inside each `Activity`. In our evaluation, with one channel having 10K activities on average, deserialization alone takes 55s to finish, which is much longer than query execution time.



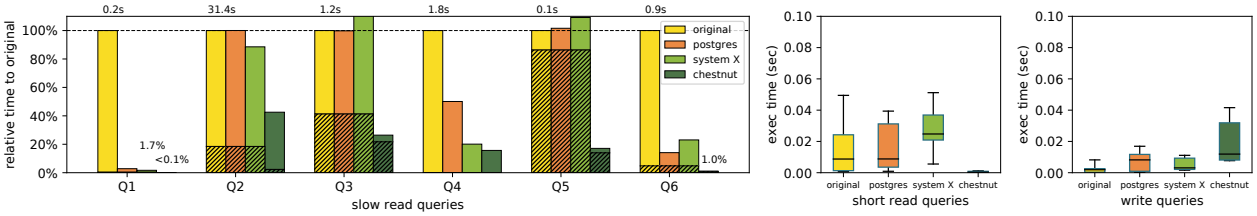
(a) *Kandan*



(b) *Redmine*



(c) *Lobsters*

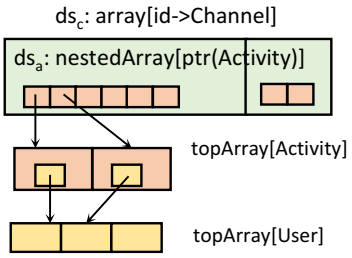


(d) *Huginn*

Figure 2.5: Query performance comparison. Figures on the left shows the relative time of slow queries compared to the original. The original query time is shown as text on the top. Each bar is divided into the top part showing the data-retrieving time and the shaded bottom part showing the deserialization time. Figures on the right shows the summary of other short read queries and write queries.

```
Channel.includes(activities, includes(user))          (a)
       .order(id).limit(50)

SELECT * FROM channel ORDER BY id LIMIT 50;
SELECT * FROM activity WHERE channel_id IN (…);       (b)
SELECT * FROM user WHERE id IN (…);
```




```
for c in dsc.scan:
  c0 = c
  for a in c.dsa.scan:                                (c)
    a0 = *a
    a0.user = *((*a).user)
    c0.activities.append(a0)
  v.append(c0)
return v
```

Figure 2.6: Case study of Kandan-Q1. (a) shows the original query. (b) shows the corresponding SQL queries. (c) shows the data layout (left) and the query plan (right) generated by CHESTNUT.

CHESTNUT instead generates a layout that stores `Channel` objects in an array sorted by `id`, a nested array of associated `Activity` pointers within each `Channel` object, and `User` pointer in each `Activity`, as shown in Figure 2.6(c). To answer this query, CHESTNUT's query plan scans the `Channel` array. Inside this scan loop, it uses a nested loop to retrieve `Activity` objects via pointers stored in each `Channel`, and then retrieves `User` objects via pointer stored in each `Activity`. Although such nested-loop query plan is slower than System X (7.5s compared to 1.7s), skipping object materialization greatly reduces deserialization time from 55s to 1.5s.

Case 2: Redmine-Q3. This query, shown in Figure 2.7(a), retrieves the `Trackers` that relate to active projects that contain issue-tracking module. Rails generates a query that contains three joins as shown in (b). The first two joins retrieve the associated `Projects` for each `Tracker` through the mapping table `project_tracker`. The third join is a semijoin that selects `Projects` within the issue-tracking module. Since the inner joins produce a denormalized table with duplicated `trackers`, the query then groups by `Tracker`'s id, followed by `DISTINCT` to generate a list of unique trackers. This many-way join takes over 8s to finish in the original application, 592ms after indexes are added in PostgreSQL, and 98ms with System X.

As this query does not have user-provided parameters, its results can be pre-computed. The

```
Tracker.where(exists(projects, status!='inactive'
    AND exists(modules, name='issue_tracking')))
```
(a)

```
SELECT DISTINCT * FROM tracker t INNER JOIN project_tracker pt
  ON pt.tracker_id = t.id
  INNER JOIN project p ON p.id = pt.project_id WHERE
  (p.status != 'inactive' AND EXISTS
    (SELECT 1 AS one FROM module m
    WHERE m.project_id = p.id AND m.name='issue_tracking'))
GROUP BY t.id;
```
(b)

ds_t: array[exists(projects, status!='inactive' AND exists(modules, name='issue_tracking')), Tracker]

```
for t in ds_t.scan:
  v.append(t)
return v
```
(c)

Figure 2.7: Case study of Redmine-Q3.

CHESTNUT-generated data layout does exactly this: it stores the `Trackers` that satisfy the query predicate into an array such that the query plan only scans this array, as shown in Figure 2.7(c). Doing so reduces the query time to only 0.2ms.

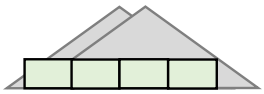
However, the data layout chosen by CHESTNUT brings extra overhead to update queries. A query that removes an issue-tracking module from a project slows down from 1ms to 76ms, since the write query plan now needs to re-examine a tracker's projects to see if that tracker contains other active, issue-tracking-enabled projects other than the one to be removed. However, the write query's

```
Project.where(status=? AND (left>=? OR right<=?))
```
(a)

```
SELECT * FROM project WHERE status=? AND
  (left>=? OR right<=?)
```
(b)

ds_{p1}: btree[(status,left)->Project]

ds_{p2}: btree[(status,right)->Project]



```
for p in ds_p1.scan:
  v_0.append(p)
init(v_1)
for p in ds_p2.scan:
  v_1.append(p)
v=distinct(union(v_0,v_1))
return v
```
(c)

Figure 2.8: Case study of Redmine-Q8.

Case 3: Redmine-Q8. The query is shown in Figure 2.8(a). It retrieves projects of a user-provided status in a subtree whose range is defined by the `left` and `right` parameters. When the selectivities of predicate "`status=? AND left>=?`" and "`status=? AND right<=?`" are small, indexing on `(status, left)` and `(status, right)` can accelerate the query compared to a full table scan. CHESTNUT-generated data layout creates these two indexes on `Projects`. The corresponding query plan performs a range scan on each index and unions the results, taking only 69ms to finish. Query optimizers in MySQL, PostgreSQL, and System X are unable to use such indexes for this query. Their query plans scan the entire `project` table even when the indexes are created, resulting in over 180ms to finish for all engines. In contrast, CHESTNUT uses custom enumeration and symbolic reasoning to find query plans rather than looking for particular query patterns to optimize. As a result, it finds better query plans as compared to the relational engines.

2.7.3 Comparison with materialized views

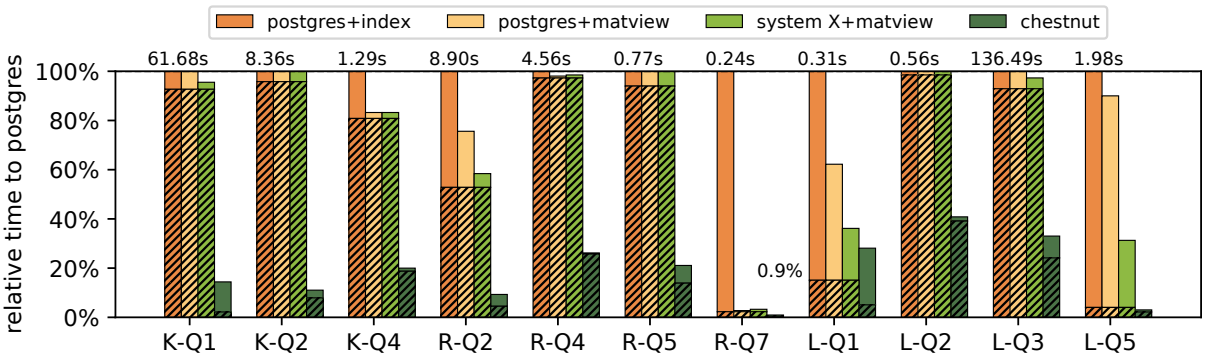


Figure 2.9: Comparison with hand-tuned materialized views on individual slow queries.

We next compare CHESTNUT's data layouts with materialized views (MVs). We use MVs to optimize the slow queries executed using relational database engines even after indexes are added. For each of these queries, we manually create different combinations of views and indexes in System X to the best of our abilities and pick those that give the best performance. Since MVs are designed to optimize a single query, it is often a union of all query results under different parameter values, indexed by the fields that are involved in the query to compare to user-provided parameters. For

example, the best MV for Q1 shown in Listing 1 is a table of all `Projects` containing open issues, with a clustered index created on field `created`. We measure the amount of memory used to create MVs for each query, and use it as the memory bound for CHESTNUT.

Both MVs and CHESTNUT selectively determine which subset of data to store, but MVs still use tabular layout and return relational query results. Instead, CHESTNUT chooses from both tabular and nested data layouts, as well as their combinations, and generate query plans that return objects. When using MVs, the bottleneck for slow queries is again object deserialization. Although MVs greatly accelerate the relational query, the overall query time is still dominated by deserialization. CHESTNUT's query plan instead returns nested objects directly without changing the data representation, significantly reducing deserialization time. The result is shown in Figure 2.9, where CHESTNUT's query plan outperforms its relational counterpart by 3.69× on average.

2.7.4 Scaling to larger data sets

In this experiment we show how CHESTNUT performs when we scale the application's data. We scale *Kandan*'s data to 50GB, and the evaluation result is shown in Figure 2.10. With larger data sets, the time spent in deserialization becomes more dominant, and CHESTNUT's ability to speed up data deserialization becomes very significant. Over the four slow read queries, CHESTNUT achieves an average speedup of 9.2× compared to MySQL, 6.6× to PostgreSQL and 6.5× to System X on the 5GB dataset, but speedup increases to 20×, 12.3×, and 12× respectively once data increases to 50GB.

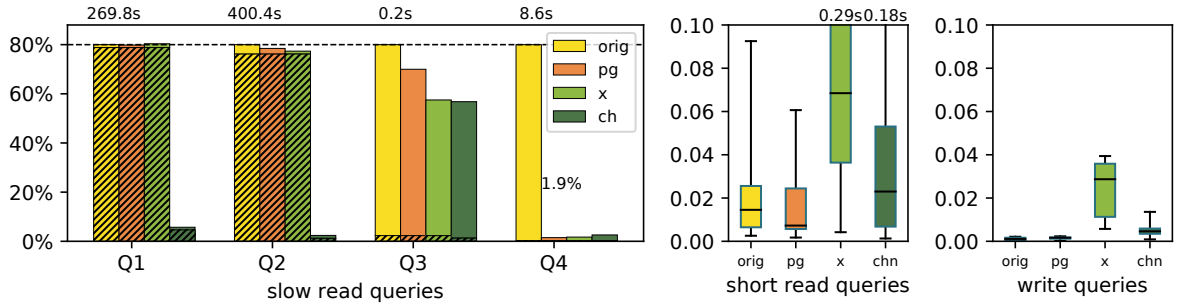


Figure 2.10: Scaling *Kandan*'s data to 50G.

2.7.5 Evaluation on TPC-H

To isolate the effect of data deserialization, we next evaluate CHESTNUT using eight analytical queries from the TPC-H [144] benchmark. Most TPC-H queries can be expressed using the Rails API in the stylized form as shown in Listing 2.2. We use CHESTNUT to find the best data layout for each query and compare the performance with System X (column store without indexes). Since these queries do not return objects, we do not include deserialization when measuring the query time, thus allowing us to study the quality of CHESTNUT-generated data layouts and plans.

The result is shown in Figure 2.11. CHESTNUT-generated database is slower in Q3 and Q6. In comparison to System X's column-oriented data store and custom machine code generation, CHESTNUT uses a slow sort from C++ STL and table scans over C++ vectors. For other queries, however, CHESTNUT is more efficient due to the better layout it finds compared to columnar tables. For example, a partial index on Q5 reduces the query time by over 90% because Q5's predicate involves many joined relations where the join predicates can be pre-computed because they do not involve user-provided parameters.

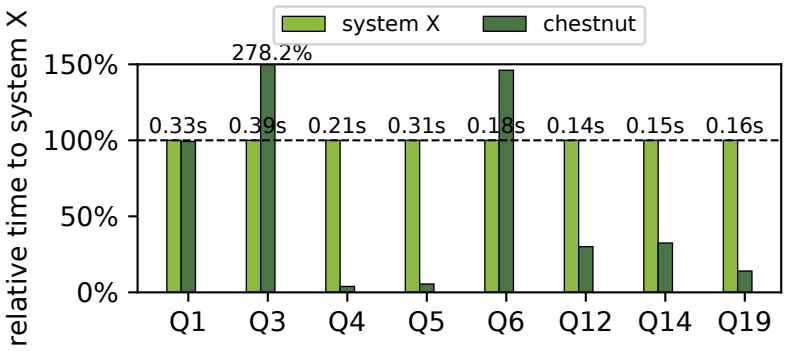


Figure 2.11: Evaluation on TPC-H.

2.7.6 Search and verification time

We run CHESTNUT on a machine with 256 cores and 1056GB memory using 32 threads, and measure the time used to find the best data layout.

Table 2.3 shows the total number of query plans enumerated by CHESTNUT (before and after pruning) and the time taken to find the best layout. The total running time includes plan enumeration and verification (described in Sec. 2.5), as well as ILP solving (described in Sec. 2.6). CHESTNUT enumerates a large number of plans even for a few queries, up to millions. With the pruning optimization described in Sec. 2.4.6, however, the number is greatly reduced by 53-96%. This reduction makes the ILP solving finishes quickly, as quick as 3min in average.

Table 2.3: The number of query plans before and after pruning, and CHESTNUT's running time.

App	# query plans		running time			unoptimized
	Orig	prune1+ prune2	plan enum	ILP solve	total	mode # query plans
Kandan	405	191	<1min	<1min	1min	34
Redmine	78K	5K	9min	1min	10min	46
Lobsters	2031K	166K	42min	12min	54min	117
Huginn	9K	880	3min	<1min	3min	41
TPC-H	143K	667	48min	<1min	48min	43

While solving ILP is fast, the majority of CHESTNUT's running time lies in plan enumeration and verification. This part varies greatly depending on the query pattern. It becomes slow when a query involves many associations. An association between classes C_1 and C_2 involved in a query predicate produces a large symbolic expression to encode the mapping. When many associations involved, the number of mapping expressions grows exponentially. Verifying complicated expression takes longer and slows down plan synthesis. For a query involving six associations, verifying one plan can take over 5min. Fortunately, verification can often be done in parallel, as discussed in Sec. 2.4.3.

Plan synthesis can also be slow when the query predicate involves many disjunctions (e.g., many `OR`s are involved). As plan synthesis enumerates all plans from small a size up to a bound determined by the number of disjunctions (described in Sec. 2.4.2), lots of disjunctions leads to a large upper bound, hence many more plans to be enumerated and verified.

Meanwhile, the reason for CHESTNUT's relatively longer running time on TPC-H and *Lobsters* is due to the two reasons mentioned above. Some queries in TPC-H involve many associations, for instance, Q5 involves six associations; some has multiple disjunctions. *Kandan* has the shortest running time because it is a small application with the fewest number of classes and associations, and the query predicates are simpler. For *Redmine*, although it also has queries with disjunctions and associations, it is fast because many sub-queries in *Redmine* shares the same predicate. CHESTNUT caches the plans synthesized for each query and sub-query, and reuses the plans when seeing the same (sub-)query without synthesizing from start, so the running time is shorter than TPC-H and *Lobsters* which barely have any shared sub-queries.

Despite pruning, CHESTNUT still takes a while to compile a few applications. To help developers view and test data layouts quickly, CHESTNUT can run in an "unoptimized" mode where the search space of data layouts is restricted to use only row-major tabular layout with foreign key indexes. For all applications, CHESTNUT enumerates much fewer plans (as shown on the last column of Table 2.3) and the total running time under this mode is less than 1 minute. We envision developers running CHESTNUT in unoptimized mode to get a preliminary design, and rerun CHESTNUT in full mode once their application is ready for deployment to obtain the best results.

2.8 Summary and Future Work

In this chapter, we presented CHESTNUT, a generator for in-memory database for OODAs. CHESTNUT searches for customized different types of data layouts to improve the performance of queries. Our experiments show that CHESTNUT can improve application's query performance by up to 42× using real-world applications.

Current version of CHESTNUT does not adapts to the workload and data distribution change; the data layout needs to be re-generated and re-loaded from scratch when the changes lead to a different optimal data layout. We leave the dynamic data layout migration to future work, as we will discuss in Ch. 7.

Chapter 3

CONSTROPT: REWRITING QUERIES BY LEVERAGING APPLICATION CONSTRAINTS

Database-backed web applications are often developed with object-oriented languages while using relational databases as the backend. In these applications, the users input data through webpages, where the data is processed by the application code before being sent to the database and persistently stored. The application data is usually associated with constraints, where a large portion of these constraints are defined in the application code (over 25% as revealed in [154]). Unlike SQL constraints, there is no standard way to define constraints in the application code and the constraints can only be inferred from how the application processes the user-input data. The database is not aware of these hidden constraints, hence not being able to leverage them to optimize storage and query performance.

To address this challenge, we build CONSTROPT to detect constraints from application code and leverage them to optimize queries. CONSTROPT infers constraints from application semantics. It then rewrites application queries to apply query optimizations utilizing such constraints. CONSTROPT differs from traditional semantic query rewrites in two ways. First, constraints detected from the application code rather than explicitly defined in the database. Second, instead of rule-based query rewrite, CONSTROPT uses verification-based rewrite. This approach is able to optimize more queries compared to rule-based rewrite approach as we have observed many distinct ways of rewrite which can not be covered by a small set of rules. We evaluate CONSTROPT on 10 real-world open-source web application. The result shows that it is able to discover over 78 constraints per application (where non of them are not already defined in the database), optimizing over 43 queries and achieving up to 43$\times$ speedup of query time.

3.1 ConstrOpt *Overview*

In this section we give an overview of CONSTROPT using an example abridged from Spree [59], a popular e-commerce web application with over 10K stars on GitHub. Spree uses a `Product` class to manage a product, and each product has multiple prices in different currencies. The prices managed by a `Price` class, with one price being the default price.

Spree stores product and price data as two tables in the database. The application manipulates the data as objects, and calls the Rails `save` function to persist the object as a tuple to the database. Lines 1-7 in Listing 3.1 shows the a function in Spree that saves a price object `p`. In this function, if the price to be saved is currently the default (i.e., the price shown on the product webpage), a query is issued to see whether another default price of the same product exist in the database (lines 3-4), and raises an error if it already exists (line 5). In the application, all prices are saved to database via the `save_price` function. As written, the function implicitly defines a data constraint that every product has at most one default price, as otherwise it would be an error.

Listing 3.1: Application code abridged from Spree showing data constraint

```
1 def save_price (p):
2   if p.is_default &&
3      Price.where(is_default = true,
4                  product_id = p.product_id).exists():
5      error('Product with default price already exists')
6   p.save()
7 end
```

This constraint can be leveraged to optimize application queries. For example, Listing 3.2 shows a query from this application that selects all products where the default price is within $10 to $100. This query has a `DISTINCT` keyword. From the constraint, we know that a product can have only one default price, hence the returned products are already distinct and the keyword can be removed. The query can be consequently accelerated by not removing duplicated tuples in the result before returned. With 100K products returned, this query can be accelerated from 278ms to 161ms, about

1.73× speedup.In this example, the query and the validation of default price are written by different developers and the developer writing the query may not be aware of the constraint and adds the `DISTINCT` keyword to ensure the returned products are unique. However, as the constraint check is done at the application level, the database is not aware of it, hence is not able to leverage them to optimize queries. Indeed, as we will show, many application constraints can be inferred from the application code but not explicitly defined in the database, and no existing tools can extract these constraints.

Listing 3.2: Application query

```
SELECT DISTINCT(Product.*) FROM Product
  INNER JOIN Price ON Price.product_id = Product.id
  WHERE Price.is_default=true
  AND Price.amount BETWEEN (10, 100) AND Price.currecy='USD'
```

CONSTROPT is built to bridge this gap. The architecture of CONSTROPT is shown in Figure 3.1. It first statically detects data constraints from the different sources like data validation before saving an object shown in Listing 3.1, and analyzes the data and control flow in order to represent the constraint with a grammar that can be parsed by a verifier (to be introduced later). It also collects the application queries by identifying the query API calls and analyzing how they are chained to construct application queries. Then it selects the queries that can potentially be optimized according to the query type and rewrites them accordingly. Because the rewrite step does not take constraint into consideration (e.g., remove `DISTINCT` keyword for all distinct queries regardless of the constraint), many rewrites may be invalid (i.e., not equivalent to the original query), and CONSTROPT next verifies every rewrite using the inferred constraints to filter out the invalid ones. To reduce runtime overhead, query detection and verification are done offline, with CONSTROPT maintaining a list of the verified pairs of original and optimized queries. During runtime, CONSTROPT intercepts all queries issued by the application and checks if it exists in the list. If so, it replaces the query with the optimized one.

We next discuss the different types of data constraints that are encoded in application code, and

how CONSTROPT detects them. Then we introduce the optimizations that leverage these constraints, followed by an evaluation showing how many data constraints can be found, how many queries can be optimized and what is the performance gain in popular, real-world applications.

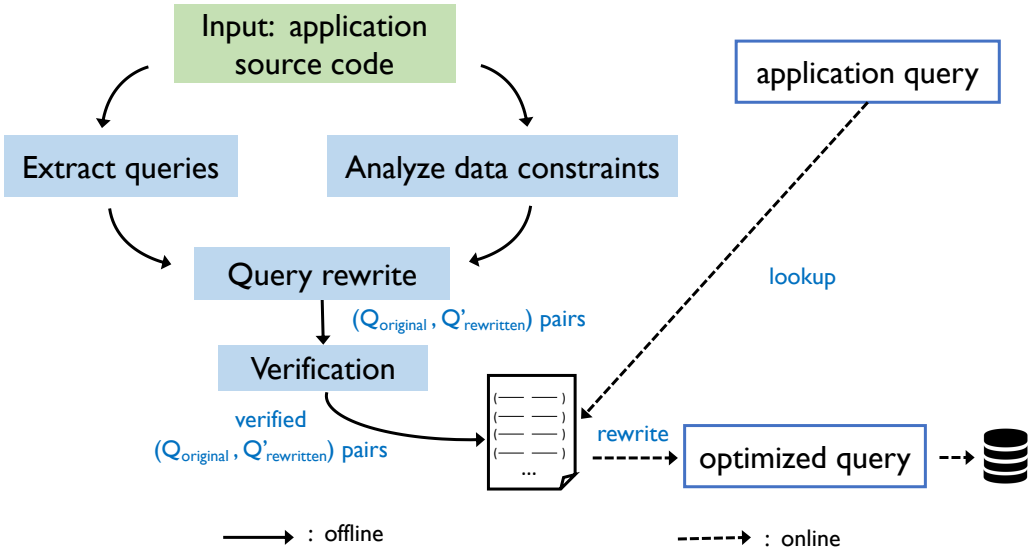


Figure 3.1: CONSTROPT architecture

3.2 Data Constraint Detection

In this section we discuss how CONSTROPT detect data constraint from application source code, and the common types of constraints that we have observed in open-source database-backed web applications.

3.2.1 Data model

Object-relational mapping (ORM) frameworks provide an object-oriented interface to manage persistent data. Usually a class or a class hierarchy maps to a database table. To connect multiple classes, these frameworks often let developers define relationship between classes, for instance, `belongs_to`, `has_one` and `has_many` relationships, where such relationship is termed as **association**. Through such association definition, relevant objects of a different can be retrieved as a field in

class. For instance, inside a `Project` class the developers defines its relationship to `Issue` class as `has_many: issues, class:'Issue'`. With this definition, given a `Project` object `p`, the issues of `p` can be simply retrieved using `p.issues` in the application, where these issues are called **associated objects** of `p`. Internally, the ORM framework will issue a query `SELECT * FROM Issue WHERE project_id=?` to the database to fetch these issues, with the parameter `?` being the primary key (`id`) of `p`. Managing classes using this association interface is a common practice for most popular ORM frameworks including Rails [1], Django [5], Hibernate [7], Entity Framework [6], DataMapper [4], SQLAlchemy [8], etc. In this work, among join and other queries that retrieve data from multiple tables (e.g., nested query), we only focus on those joining on pre-defined assoications (e.g., a join condition like `Issue.project_id=Project.id`), which we observe to cover over 90% of the queries in real-world applications we evaluate.

3.2.2 Sources of data constraints

Unlike database constraints that are explicitly defined using the SQL constraint language, there is no standard way to define constraints in the application code. Yet many data constraints can be inferred from how the application processes data. In the following we introduce four major sources where data constraints can be inferred. In this section we use applications built with Ruby on Rails as example, but the methodology can be generalized to other frameworks, which we will briefly discuss at the end of this section. We illustrate using examples abridged from four Rails applications: 1) *Redmine* [54], a project management application like GitHub; 2) *Onebody* [44], a social network management application; 3) *Discourse* [23], a discussion forum application, and 4) *Spree* [59], an e-commerce application for shopping and inventory management. Through these four types of sources many constraints can be detected from the application code, which we will present in Sec. 3.5.

Data validation. Developers often wish to validate the data before persist it to the database. The validation process checks an object to be persisted against certain properties, from simple ones like some fields should not be null, to more complicated properties that relates to other data which requires issuing queries to perform the checking (e.g., the constraint in Listing 3.1 where only one

default price exists). Such validation takes an object as input and is performed whenever an object is inserted or updated to the database, and returns an error to the user if validation fails. Many ORM frameworks provide APIs to support such validation [29, 37]. These APIs include built-in functions which check against frequently-used properties, as well as general functions that allow developers to write arbitrary validation logic.

Listing 3.3 shows an example from Redmine with Rails validation API. The application uses a `IssueRelation` class to maintain relationships between two issues. Line 2 calls a built-in validation function, `validates_uniqueness_of`, which states a data constraint that the two fields `issue_from_id` and `issue_to_id` forms a primary key (i.e., unique across the table), as illustrated on line 3. Lines 4-9 defines a validation function that returns an error when the relation is between the same issue, i.e., for every tuple in the `issue_relations` table, the value of `issue_from_id` is not equal to `issue_to_id`.

Listing 3.3: Data validation example

```
1  class IssueRelation
2    validates_uniqueness_of [issue_from_id, issue_to_id]
3    # Constraint(IssueRelation, unique([issue_from_id, issue_to_id])
4    validate validate_issue_relation
5    def validate_issue_relation
6      if issue_from_id = issue_to_id
7        error('relation on the same issue')
8      end
9    end
10   # Constraint(IssueRelation, issue_from_id != issue_to_id)
```

Association definitions. ORM frameworks often support defining relationships like `has_one`, `has_many` and `belongs_to` between classes. Such relationship enables developers to access associated object in the same way as normal object attribute (i.e., using dot method) instead of explicitly writing a join query. For instance, in Listing 3.4, with the `has_many` association defined on line 2, the associated variants of a product `p` can be retrieved using `p.variants`. ORM frameworks

often generate a foreign key constraint when an association between two classes is defined (e.g., product_id field of Variant table is a foreign key to the id field of Product).

Besides association between two classes, developers often define associations between subset of two classes or multiple (over two) classes in order to leverage the convenience of object retrieval with dot method. Such associations indicates more complex data constraints than foreign key which are not mapped to a database constraint by ORM frameworks.

Lines 3-4 in Listing 3.4 shows an example, where the developers define a has_one association between Product class and a subset of records of Variant class using a predicate (is_master is a Boolean field of Variant class). This definition indicates a data constraint on the variants table: among all tuples that is_master is true, the value of product_id field is unique. Another example is shown on line 7, where a has_one association is defined between Address and potentially many other classes. The type of the associated addressable is polymorphic instead of a concrete class, which means that the type of addressable can be different classes like Person or Company. Two fields in Address are used to maintain such association, addressable_id and addressable_type, where addressable_type is a string field storing the type of addressable. As such, polymorphic association indicates a constraint on the addressable_type field whose value can only be the class name of a class in the application.

Listing 3.4: Example of relationship definition

```
1 class Product
2   has_many variants, class_name: 'Variant'
3   has_one master_variant, class_name: 'Variant'
4     -> {where is_master = true}
5   # Constraint(Variant, unique([product_id], is_master==true))
6 class Address
7   belongs_to addressable, polymorphic: true
8   # Constraint(Address, addressable in ['Person','Company',...]
```

Field definitions. Developers often defines a class field together with a specification of how the field should be used or how the value should change. Such specification indicates a constraint on

that field. For instance, Listing 3.5 shows an example from Spree that processes payments. The `Payment` class includes a string field `status`. The application defines the change of `status` using a state machine [51] to update the value of `status` from one state represented by a string to other states upon certain event. For instance, upon event `start_processing`, the value of `status` is changed from string "checkout," "pending," or "completed" to "processing," etc. The application can only call an event to change the value of `status`. With such a definition, the value of `status` can only be the state names defined in the state machine.

Listing 3.5: Example of field definition

```
1  class Payment
2    state_machine :state do
3      event :start_processing do
4        transition from:['checkout','pending','complete'], to: '
     processing'
5      end
6      event :failure do
7        transition from:['pending', 'processing'], to: 'failed'
8      end
9      ...
10   end
11   # Constraint(Payment, state in ['checkout','pending',...]
```

Field value assignment. Data constraints can be inferred through the pattern data inserts and updates. The data to be stored to the database does not always come from user input; sometimes they are computed based on a query result. When the value of a field f is always derived from fields $g_1, \ldots, g_n$ that are stored in the database, f is functionally dependent on $g_1, \ldots, g_n$. This is a common practice in many applications to improve application performance. For example, developers often use a field to store an aggregation result of infrequently-updated data (e.g., the count of replies to a post) to accelerate aggregation queries, or use a field to store the value of an associated object's field to efficiently read that value without joining associated objects (similar as

foreign key except that the referenced value may not be a key).

Listing 3.6: Example of field value assignment

```
1 class TimeEntry
2   belongs_to issue, class: 'Issue'
3 # code to create a time entry
4 i = Issue.find_by_id(param[:issue_id])
5 time_entry = TimeEntry.new(issue=i,
6                         project_id=i.project_id)
7 time_entry.save
8 # Constraint(TimeEntry, project_id=issue.project_id)
```

Listing 3.6 shows an example from Redmine showing how objects of class `TimeEntry` are created and persisted. The `TimeEntry` class manages the progress and the amount of time spent on resolving an issue. Each `TimeEntry` has a `belongs_to` association with an `Issue` object so the corresponding issue can be accessed using `.issue` just like accessing a field, and the field `issue_id` is a foreign key to the `Issue` table. Lines 4-7 shows the code piece to create a time entry. Line 4 retrieves an issue, and line 5-6 creates a time entry object, and assigns the issue object `i` to the associated `issue` of `time_entry`, which is essentially assigning the `id` field of issue `i` to the foreign key `issue_id` of `time_entry`. On line 6 the `project_id` field is assigned as the `project_id` of the same issue `i`, and line 7 saves the `time_entry` to database. In this application, lines 4-7 is the only code to create and insert time entry, and the `issue_id`, `project_id` of `TimeEntry` and the `project_id` of `Issue` is not modified once persisted to database. Field assignment in this example hence indicates a functional dependency: the `project_id` of `TimeEntry` is equal to the `project_id` of the associated `Issue`. The detection of such constraint is explained later in Sec. 3.2.4.

3.2.3 Representing data constraints

We use the grammar shown in Table 3.1 to represent the inferred data constraints. A constraint is on a particular table where the expression `e` must be true for all the tuples in that table. The `expr`

is represented in a logical expression or three commonly-used constraints including `unique` (i.e., field(s) value is unique across the table), `match` (i.e., a string field value matches the regular expression `regex`) and `foreignkey`. The `expr` can involve data from multiple tables using association (`assoc.assoc...field` as shown in Table 3.1).

spec	:=	Constraint(table, e)	*constraint*
e	:=	!e | e ∧ e | e ∨ e | e → e	*logical op*
	|	e == e | e != e | e ≤ e | ...	*compare op*
	|	e + e | e - e |...	*arithmetic op*
	|	e ∈ e | e ∉ e | | size(e)	*set op*
	|	forall(a, e) | exists(a, e)	
	|	a | length(a)	*atomic element*
	|	unique([field,...], e)	
	|	match(field, regex)	
	|	foreignkey(field, table, field)	
a	:=	const | field	*atomic element*
	|	assoc.field | assoc.assoc..field	*associated obj*
	|	q	*query*
q	:=	Table.where(p) | assoc.where(p)	*query*
p	:=	!p | p∧p |...| a=a | a<a |...	*query predicate*

Table 3.1: Grammar of constraint

3.2.4 Constraint detection from code analysis

CONSTROPT uses an ORM-aware static analyzer for applications built using the Ruby on Rails [55] framework. For such applications, an action is the entry point of webpage generation upon receiving an HTTP request from the client. The Rails server invokes an entrance function when it starts to process the request, and which entrance function to call is based on the routing rules defined in the application. CONSTROPT produces a program flow graph (which consists of data and control flow information) for each entrance function that contains the data-flow and control-flow information

for each action. It iteratively inlines all function calls invoked by the entrance function in order to perform inter-procedural analysis. We use type inference algorithm described in previous work [104] and do not handle polymorphic functions in our current prototype.

Using the program flow graph, ConstrOpt traces the control and data flow from certain target nodes to the data sources that affects the value or execution of the target nodes. ConstrOpt identifies three types of target nodes: 1) a pre-defined set of API calls (e.g., validation APIs shown in Listing 3.4, state machine APIs as shown in Listing 3.5, etc.), 2) statement that throws an error (e.g., line 7 in Listing 3.4), and 3) statement that persist a record to the database (e.g., lines 7-8 in Listing 3.6). ConstrOpt traces the data and control flow backward from these target nodes until it reaches a data source, where the source is a node representing an *atomic element*, i.e., a constant value, a field including associated object, or a query. The specification of *atomic element* is defined in Table 3.1. Next we introduce how ConstrOpt analyzes these different target nodes to generate constrair

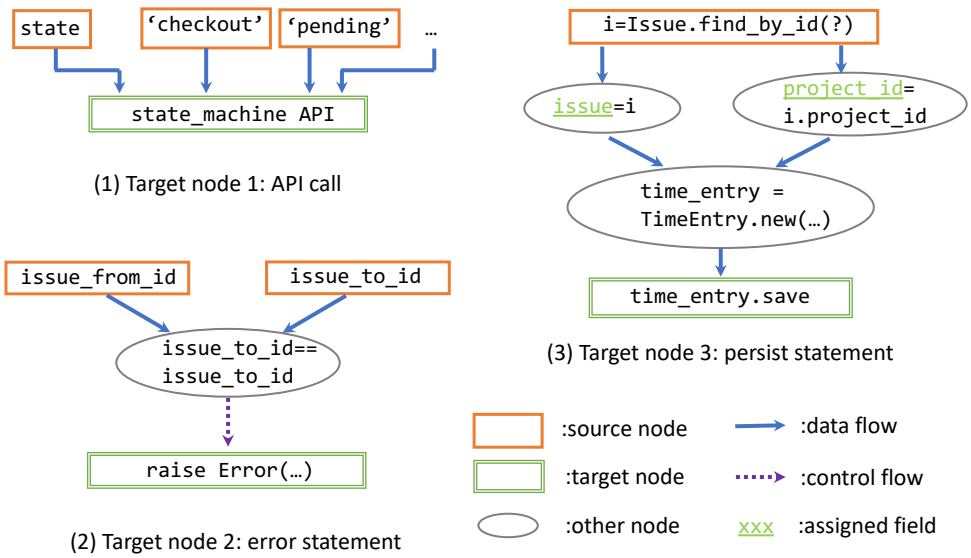


Figure 3.2: Example program flow analysis for different types of target node.

API calls. ConstrOpt analyzes the data sources of an identified API call and traces the data source of every parameter. Figure 3.2(1) shows the data flow for the state machine API call listed in List-

ing 3.5. CONSTROPT checks the source of the first parameter, `state`, to see if it is a field persistently stored in the database. Then it checks the data sources of the rest parameters (i.e., the states specified after `transition from` and `to`) to see if they are constant values. If all sources are constants, CONSTROPT then generates an inclusion constraint of the form `Constraint(t,f∈[`$P(Sr_1)$`,...])`, where Sr_i is the data source of the *i*th state and $P(Sr_i)$ is the expression that computes the *i*th state is computed from Sr_i. In Figure 3.2(1), $P(Sr_i)$ is simply Sr_i because the value of the state is the constant value itself.

Error statements. CONSTROPT currently identifies error statement in validation functions like the error statement in Listing 3.3. It first collects all the branch conditions that leads to an error statement by tracking control flow, e.g., the branch condition `issue_from_id==issue_to_id` in the program flow shown in Figure 3.2(2). Then it traces the data source of each variable in the branch condition and replace the variable with an expression computed from its data source. The negation of the conjunction of branch expressions forms the constraint, e.g., `!(issue_from_id==issue_to_id)` from the example in Figure 3.2(2), because any record passing the conjunction of branches will cause an error and will not be saved to the database.

Persist statements. CONSTROPT identifies function calls like `save` that persist an object to the database. It analyzes each assignment that changes a field value of that object, including explicit assign statements and values passed to constructor calls. Conceptually, CONSTROPT traces the data source of each assigned field and see if two assigned fields share the same data source. Then it checks whether one assignment is a direct assignment of the data source without any computation, and infers an equivalence constraint by replacing the data source in another assignment[1]. For example, Figure 3.2(3) shows the data flow for the persist statement in Listing 3.6. The assignment of associated `issue` and field `project_id` shares a same data source, i.e., issue `i`, and the assignment of `issue` on line 5 is a direct assignment from `i`. So CONSTROPT replaces the common data source, issue `i` with the field `issue` in the assignment of field `project_id` and derives an equivalence

[1]Although other patterns of persistent statement (e.g., patterns where assignment of two fields having different data sources) can infer data constraint, we only detect the pattern described above, which we observe to be the most common, while leaving the detection of other patterns as future work.

constraint `project_id=issue.project_id`. To ensure this constraint holds, CONSTROPT also checks all other persist statements in the application to see if all the fields of the class to persist on the left hand side is not assigned elsewhere in the application (e.g., `time_entry_id` and `project_id` of `TimeEntry` is not assigned in other ways than Listing 3.6), and these fields on the left hand side is re-assigned whenever any field on the right hand side (e.g., `issue` or the `project_id` field of `Issue`) is modified. CONSTROPT considers this constraint as valid only if these two criteria holds.

3.2.5 Common data constraints

We inspect the constraints detected by CONSTROPT through the analysis process presented in Sec. 3.2.4. Through this inspection, we find a small number of constraint patterns cover over 82% of all the constraints detected. In the following we present these patterns by the order of their frequency, with the detailed number of constraints under each pattern presented in Sec. 3.5.

- Categorical constraints, in the form of `Constraint(table, field∈{const1,const2,...})`. The value of a field is restricted to a limited set, for instance, the `state` value in Listing 3.5 can only be "checkout," "pending," etc., as defined in the state machine.
- Not null constraints, in the form of `Constraint(table, field!=NULL)`. The value of a field cannot be null. It is the same as not null constraint in SQL but defined only in the application code.
- Length constraints, in the form of `Constraint(table, length(field)<const)` where the comparison can be $>$, $<$, $>=$, $<=$ and $==$. The length of a string field can only be within certain range, for instance, the length of `username` cannot exceed 60 characters.
- Uniqueness constraints, in the form of `Constraint(table, unique([field1,...])`. It is the same as the SQL uniqueness constraint, but only defined in the application code.
- Format constraints, in the form of `Constraint(table, match(field, regex)`. The value of a string field must match a regular expression where the regular expression is extracted from the application code. For instance, an application validates a string field `barcode_id` to match regular expression `\A\d+\z` (i.e., those containing only numeric digits).
- Equality constraints, in the form of `Constraint(table, field1=field2)`, where `field1` and `field2` can also be field from associated object. A field's value is always equivalent to another

field, which is essentially encodes a functional dependency. An example of such constraint can be found in Listing 3.6 where a time entry's `project_id` is equivalent to its issue's `project_id`.

• Non-equality constraints, in the form of `Constraint(table, field1!=field2)`, where `field1` and `field2` can also be field from associated object. A field's value can never be equivalent to another field. An example is shown in Listing 3.3 where the two issues involved in an issue relation are not the same issue.

3.2.6 Constraint detection for other ORM frameworks.

Although CONSTROPT is built for Rails applications, the detection process can be extended to other frameworks as well. For instance, Django has the same validation mechanism as Rails [29] and also provides a set of validation APIs. It also provides APIs to specify association with partial or multiple tables [28] where inclusion or conditional unique constraints can be inferred, as seen in Listing 3.4, and multiple parameters to specify properties of a table field [27]. The techniques described to infer constraint from data validation and persist statement based on data and control flow analysis, which can be applied to applications built using other frameworks as well.

3.3 Leveraging Data Constraints to Optimize Queries

We collected optimizations leveraging constraint proposed by previous work [135, 86], and sampled query logs from running applications to manually exam and discover opportunities to optimize. We summarize the optimizations that we observe to be prevalent using manual checks and introduce them in this section.

3.3.1 Remove distinct operator

If a query contains a `DISTINCT` keyword, it can be removed if we know the result contains no duplicates given the constraints. Listing 3.7 shows an example abridged from Redmine [54]. Given the primary key constraint on fields `user_id` and `project_id` shown in line 1, the query in lines 2-4 that select users who are members of one particular project already returns distinct users, so

the DISTINCT keyword can be removed. The query before and after the optimization is shown in Listing 3.7. Evaluating using 10K users returned, the query time is reduced from 270ms to 160ms, achieving 1.69 × speed up.

Listing 3.7: Example of remove distinct

```
1 -- Constraint(Member, unique([user_id, project_id]))
2 SELECT DISTINCT(users.*) FROM User u
3   INNER JOIN Member m ON m.user_id = u.id
4   WHERE m.project_id = ?     --before
5 SELECT users.* FROM User u
6   INNER JOIN Member m ON m.user_id = u.id
7   WHERE m.project_id = ?     --after
```

3.3.2 Limit the number of results

The LIMIT 1 keyboard can be added to a query if we know that such that the query can stop as soon as it finds a satisfying tuple before completing the entire query. Listing 3.8 shows an example, where the login string (a hashed authentication string) of the User table is unique according to the constraint shown on line 1. The query (line 2) selects the user that matches a given login string. This query returns at most 1 User record under the constraint, so we can add LIMIT 1 to the end of the query. Whether this optimization can bring performance gain depends the implementation of LIMIT in the database, i.e., whether the database is able to return early after finding the desired number of results before finishing examining all tuples. With 1M users and evaluated on MySQL which returns result early under LIMIT query, the original query shown in Listing 3.8 takes 1400ms to finish. The performance of optimized query depends on the position of the matching record in the table, i.e., how many other records need to be scanned before finding the matching one. In this example, the optimized query ranges from 0.1ms to 1400ms, achieving speedup as much as 14000×.

Listing 3.8: Example of add limit 1

```
1 -- Constraint(User, unique([login]))
```

```
2 SELECT * FROM User WHERE login=?            --before
3 SELECT * FROM User WHERE login=? LIMIT 1    --after
```

3.3.3 Remove predicate

Predicates in a query can be removed if they always hold in that query given the inferred constraints. Listing 3.9 shows an example abridged from Onebody. The constraint on line 1 states that the `category` field of `Group` cannot be `NULL` or an empty string. The query on lines 2-4 contain the predicate `category IS NOT NULL AND category != ''` to check that the group belongs to some category. Under the constraint, this predicate evaluates to true for all group tuples and can thus be removed. With 100k groups, the new query takes 82ms, reducing 26% compared to the original query which finishes in 103ms.

Listing 3.9: Remove predicate optimization

```
1 -- Constraint(Group, category!=NULL && category!='')
2 SELECT category, count(*) FROM Group WHERE category
3   IS NOT NULL AND category!='' AND
4   category!='Subscription' Group by category   --before
5 SELECT category, count(*) FROM groups WHERE
6   category!='Subscription' Group by category   --after
```

3.3.4 Remove join

For queries that join on tables but not project on those tables nor filter using fields from those tables other than the join condition, the join might be removed. Listing 3.10 shows an example from Redmine. An `Issue` has one `status` where the type of `status` is class `IssueStatus`. The constraint says that for every `Issue` it always has a `status`, which means that the field `status_id` in `Issue` always references to a record in the `IssueStatus` table. The query on line 2-5 joins the `IssueStatus` table but does not filter using any field from `IssueStatus`, so this join can be

removed. The join removal reduces the query from 260ms to 180ms, speeding up the query time by 1.44×.

Listing 3.10: Example of remove join

```
1 -- Constraint(Issue, status!=nil)
2 SELECT COUNT(*) FROM Issue i INNER JOIN
3   Project p ON Project.id=Issue.project_id INNER JOIN
4   IssueStatus ON IssueStatus.id=Issue.status_id
5   WHERE (Project.status <> 'archived')          --before
6 SELECT COUNT(*) FROM Issue i INNER JOIN
7   Project p ON Project.id=Issue.project_id
8   WHERE (Project.status <> 'archived')          --after
```

3.3.5 Replace disjunctions with unions

When a query contains disjunctive predicates, each predicate in the disjunction can be converted into a single subquery where the results from the subqueries are later unioned. This rewrite has been studied by prior work [129, 25], where the idea is that by splitting disjunctive predicate into subqueries, the query optimizer can choose different access path for each subquery. The downside of this rewrite is that the union query has an extra cost of removing duplicated result when merging the results from subqueries, and this may offset the benefit of using different access path. However, if the subqueries contain no overlaping results (as determined by the constraints), then the duplicate removal step can be avoided.

Listing 3.11 shows an example from Discourse. Each `Category` has a parent category and the constraint on line 1 states that the parent cannot be itself. The query on Line 2 shows a query that selects topic that either belongs to the category specified by the variable `c`, or belongs to a category whose parent is category `c`. The original query plan scans the `topics` table, and uses a subplan which performs a sequential scan on the categories table to find the match for parent category, as shown in Figure 3.3(1). Since we know from our extracted constraints that if a topic belongs to a

category c (i.e., its field `category_id` matches the variable c), then it cannot belong to the children categories of c, so this topic does not overlap with any topic whose parent category is c. As a result, we can split the disjunctive query and use `UNION ALL` (i.e., union without removing duplicates) to combine the query results on the two sides of the disjunction. The new query using `UNION ALL` is shown on Line 3 in Listing 3.11. The new query generates a query plan that is able to perform a hash join to process the subquery, which is more efficient than doing sequential scan on both tables as in the original plan

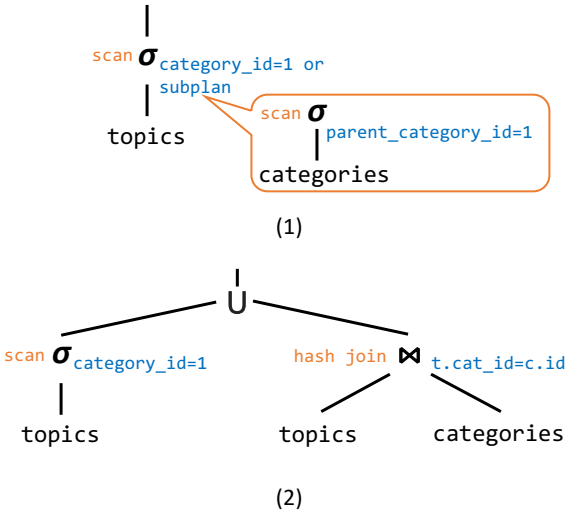


Figure 3.3: Before and after query plans of replacing disjunctions with unions. The parameter `:c` (shown in Listing 3.11 is given a concrete value 1 in the query plan.

Listing 3.11: Example of replace disjunctions with unions

```
1 --Constraint(Category, parent_category_id!=id)
2 SELECT * FROM Topic WHERE category_id = :c OR category_id IN (SELECT
    id FROM Category WHERE parent_category_id = :c --before
3 SELECT * FROM Topic WHERE category_id = :c UNION ALL (SELECT * FROM
    Topic WHERE id IN (SELECT * FROM Category WHERE parent_category_id
     = :c)) --after
```

3.3.6 Add parameter precheck

Many queries in the application filters on a field to match with a user input. If the field has a length or format constraint, a pre-check on the user input can be added to avoid running the query altogether if the input fails the pre-check. For instance, Listing 3.12 shows an example from Onebody. Line 1 shows a length and format constraint on the `barcode_id` field of the `Family` class. `barcode_id` can only contain digits and its length is between 5 to 50. One webpage in the application checks if the user inputs a valid family barcode. It issues a query (Line 3) to fetch the corresponding family. If the user inputs a barcode that does not satisfy the the length and format constraints, then no matching family tuple would be returned. We can add a pre-check as shown from line 5-9 to validate the parameter, and avoid executing the query if the pre-check fails. This brings significant performance gain the `barcode_id` is not indexed by the application, so running the query that returns no match requires scanning the entire table. For instance, with 100k records in the family table and 20% of the chance a user provides invalid barcode, a pre-check improves the query performance by 1.27×.

Listing 3.12: Example of add parameter check

```
1 # Constraint(Family, match(barcode_id, /\A\d+\z/) && length(
    barcode_id) between (5, 50))
2 # before
3 family = Family.where(barcode_id: param[:barcode]).first
4 # after
5 if param[:barcode].match(/\A\d+\z/) && param[:barcode].length.between
    ?(5, 50)
6   family=Family.where(barcode_id: param[:barcode]).first
7 else
8   family=nil
9 end
```

3.3.7 Change storage format

If a string or integer field has a categorical constraint, i.e., its value can only be a limited set of constants, we can replace this field with an enumeration type. Usually an enumeration type is stored as a byte in database [31, 30]. As a result, changing storage format not only improve the time to process predicate on that field (as byte comparison is faster than string or full integer comparison) but also the space efficiency as well.Listing 3.13 shows an example from Onebody. The gender of a person is stored as a string in the `Person` table whose value can only be `'male'` or `'female'`. Line 2 shows a query to select all females. By changing the storage format of `gender` from string to enum, with 1M records in the `Person` table, the query can be accelerated from 248ms to 206ms while the table size is reduced by 8MB. In Sec. 3.5 we will see that a large number of fields are involved in categorical constraint and changing storage format can substantially reduce the storage cost.

Listing 3.13: Example of change storage format

```
1 # Constraint(Person, gender in ['male','female'])
2 SELECT Person.* FROM Person WHERE gender = 'female'
```

3.4 Automatic Query Rewrite and Verification

We now describe how ConstrOpt automatically detects and implements the optimization described in Sec. 3.3. As shown in Figure 3.1, ConstrOpt first analyzes the application source code to collect the queries and data constraints. Then it decides whether a query can be optimized based on the query type and the constraints involved, and optimizes accordingly. It then verifies each rewrite to check if the new query is equivalent to the original query. Through this offline process, ConstrOpt maintains a list of valid query rewrites. At runtime, it intercepts the queries issued by the application, checks against the list and dynamically replace the issued queries with the optimized ones. We discuss each step in detail next.

3.4.1 Query analysis

As described in Sec. 3.2.4, CONSTROPT's analyzer identifies database queries by looking for calls to Rails' query API [50][2] Since the query functions are often chained to generated the final query, our analyzer identifies the invocation chain to collect all components of a query using program flow and inter-procedural analysis. Because Rails supports both using query API and customized SQL query string (i.e., partial SQL query passed as a string to Rails query API), CONSTROPT analyzes the query API parameter and parses SQL string in order to understand the query (e.g., the query type and the fields involved).

3.4.2 Static query rewrite

Next CONSTROPT statically applies the optimization described in Sec. 3.3 on the detected queries.

3.4.2.1 Rewrite queries

For each query Q, CONSTROPT decides whether Q is a candidate for an optimization based on the query type and the data constraints. It generates Q' from Q if Q is a candidate. The process for each optimization type is presented below. Multiple optimizations can be applied to one query and we keep all combinations of them. This step simply selects potential Qs and generates all possible Q's mostly based on the type of Q. As a result, the generated Q's may not be equivalent to Q, and the correct rewrite will be selected and verified later.

- *Remove distinct operator.* If Q contains the `DISTINCT` keyword, it is a candidate for this optimization, and Q' is generated by removing `DISTINCT` from Q.
- *Add limit 1.* Q is a candidate if 1) it does not have `LIMIT` keyword; 2) it involves only one table `T`; 3) there exist a constraint that only contains fields in `T` and also contains a `unique` expression (as shown in Table 3.1). Q' is generated by adding `LIMIT 1` at the end of Q.

[2] CONSTROPT also identifies a few APIs that process the query result where the computation can actually be pushed to the database, e.g., the `uniq` API filters duplicated elements in an array which ha the same effect as adding `DISTINCT` to the SQL query.

• *Remove predicate.* If Q contains a predicate comparing a field with a constant value (including NULL), and that predicate also appears in any constraint, then Q is a candidate for this optimization and Q' is generated by removing that predicate from Q.

• *Remove join.* If Q contains an inner join on table `t` but no there is not filtering using `t`, then Q is a candidate, and Q' is generated by removing that join from Q.

• *Replace disjunctions with unions.* If Q contains an `OR` in the query, it is a candidate. Q' is generated by splitting the predicates connected by `OR` into two separate queries, copying the rest of the original query to these two queries and merging them using `UNION ALL`.

• *Add parameter precheck.* If Q contains a predicate `t.f=?` where the parameter is computed from a user input (by checking the data flow graph), and the field `f` is involved in a format constraint or a length constraint, then Q is a candidate. While the query remains the same, the optimization adds a precheck to see if the parameter satisfies the length and format constraint, as shown in Listing 3.12.

• *Change storage format.* If Q involves a field `f` where `f` is involved in a categorical constraint, then Q is a candidate. The optimized query is the same as the original query.

3.4.2.2 Verification

As we can see, the rewrite step mainly produces Q based on the type of Q' and uses constraints only to reduce the number of potential Q's but not to generate the correct Q'. So after obtaining pairs of (Q, Q'), CONSTROPT verifies whether each Q' is a correct rewrite leveraging the constraint. It achieves this by verifying whether the two queries, Q and Q', always return the same results for all possible input relations.

CONSTROPT leverages recent advances in symbolic reasoning [146, 97] for verification process, similar to Cosette [96] and Chestnut [151]. Conceptually, this is done by populating each table in the database with a fixed number (currently 3) of symbolic tuples (i.e., a tuple whose field values are variables with unknown contents). Next, it adds a set of constraints expressed using these variables. These constraints include both the constraints specified in the database as well as the constraints detected by CONSTROPT from the application code. To do so, CONSTROPT runs the constraint expression (expressed using grammar shown in Table 3.1) on each symbolic tuple and obtains

a set of assumptions (e.g., the constraint holds on every tuple). Then it runs both Q and Q' on these symbolic tables. Note that both Q and Q' are query templates because they are collected via static analysis, so each non-constant query parameter are also represented with a symbolic variable. ConstrOpt sends both results to a solver [70] to check for equivalence under all possible values of variables under the assumptions obtained from the constraints. It retains only those rewrites that are provably equivalent under any values of the variables. If multiple rewrites are correct for one query, it only keeps the rewrite with the most optimizations.

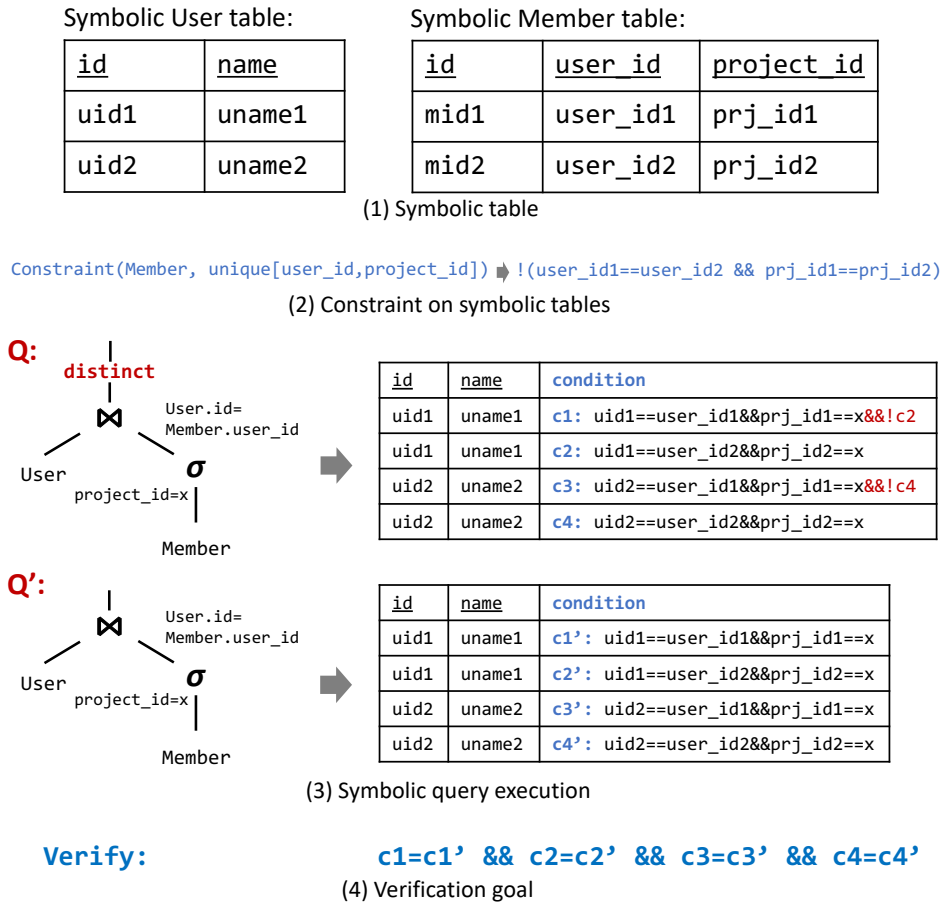


Figure 3.4: Illustration of query rewrite verification

We illustrate verification using the query rewrite in Listing 3.7, where the process is shown in Figure 3.4. Figure 3.4(1) shows two symbolic tables populated by ConstrOpt, `User` and

`Member`, where `uid1` and the rest are variables. In our implementation we use 3 symbolic tuples per table in CONSTROPT but only show 2 in Figure 3.4 to simplify the illustration. One constraint on the `Member` table states that the fields `user_id` and `project_id` forms a primary key. Applying this constraint on all tuples of the `Member` table produces an assumption as shown in Figure 3.4(2). Then CONSTROPT runs Q and Q' on the symbolic table and obtains query result shown on the right of (3). Note that unlike running query on concrete tables, whether a tuple appears in the final result depends on the value of the fields in the tuple. So a query result contains all potential tuples, in this example, the cross product of `User` and `Member` projected on `User`, while each tuple has a corresponding "condition" expressed with variables in this tuple to determine whether it will appear in the query result. As illustrated in (3), for the result tuple 1 (on the first line) which `User` with id `uid1` joins `Member` with id `mid1` (i.e., the first row in the result of Q), the corresponding condition `c1` checks whether the join condition is valid (i.e., `uid1==user_id1`) and whether the query predicate holds (i.e., `prj_id1==x` where `x` is the variable of query parameter). Furthermore, because of the `DISTINCT` keyword in Q, at most one result tuple containing `uid1` can appear in the result set. Therefore `c1` also contains a predicate `!c2` restricting that tuple 1 can be included in the final result only if tuple 2 is not included[3]. In contrast, the new query Q''s conditions do not have `DISTINCT` keyword, and the corresponding condition `c1'` does not include `!c2`. Finally CONSTROPT verifies that each result tuple appear in Q if and only if it appears in Q', which is essentially verifying the equivalence of the conditions of every tuple in the result of Q and Q', as shown in Figure 3.4(4). In this example, the verification goal evaluates to true under all possible table and parameter values under the constraint. Q' is a valid optimization if the verification passes.

3.4.3 Dynamic query optimization

The above steps to generate valid (Q, Q') pairs are performed offline. CONSTROPT maintains a hash table that maps Q to its respective Q'. Both Q and Q' are query templates. At runtime, the ORM framework constructs the query from query API calls. It first constructs the query template

[3]For DISTINCT query we assume the last matching tuple is included. Because all tuples are represented by variable, the order will not matter to the result.

from which APIs are called, then fills in the query parameters specified as parameters to API calls. CONSTROPT intercepts before the ORM framework fills in the parameter, and checks the runtime query template against the hash table. If there is a match, CONSTROPT replaces the runtime template with the optimized Q' from the hash table, and then send the Q' back to the ORM to fill in the parameters and subsequently sends to the database.

Although the whole process can be done at runtime and integrated into query optimizer, we choose to implement CONSTROPT as above due to two reasons. First, many applications are designed to be independent from a particular database implementation, and implementing the rewrite inside the ORM framework allows the application to be optimized even when the application switches to another database implementation. Second, due to a large number of candidate rewrites and a non-trivial amount of time for the verification, it may slow down other queries that cannot be optimized, especially many applications issue many small, fast queries at each page. We leave an efficient implementation of dynamic rewrite inside query optimizer as future work.

3.5 Evaluation

3.5.1 Application corpus

We select 10 real-world web applications built using the Ruby on Rails framework, covering 4 application categories: forum, collaboration, e-commerce, and social network. We select the up to 3 most popular, actively maintained applications from each category based on the number of GitHub stars, and from this pool we choose the top 10 starred applications as listed in Table 3.2. Among these applications, 6 of them have over 10K stars and 8 for them are maintained by over 100 contributors, and all of them have been developed for over 4 years. We choose the most recent versions of these applications to evaluate, which we believe have already been carefully developed and manually optimized.

Table 3.2: Details of the applications chosen in our study

Category	Abbr.	Name	Stars	Tables	Avg #fields per table
Forum	Ds	Discourse	30.8k	147	8
	Dv	Dev.to	2.1k	73	10
	Lo	Loomio	1.9k	44	9
Collaboration	Re	Redmine	3.7k	54	8
	Gi	Gitlab	22.2k	356	4
	Op	OpenProject	1.2k	54	2
E-commerce	Sp	Spree	10.4k	82	6
Social Network	Da	Diaspora	12.4k	44	6
	On	Onebody	1.4k	54	9
	Ma	Mastadon	3.6k	68	6

3.5.2 Evaluation platform

We implement CONSTROPT using a mix of Python and Ruby where the Ruby code analyzes the application code to detects queries and constraints, and the Python code rewrites queries and validates the rewrite. In evaluating query performance, we run the applications on a server with 2.4G Hz processor and 8GB memory.

3.5.3 Constraint detection

In this experiment we present how many constraints are detected by CONSTROPT from the application code. We run CONSTROPT on the applications shown in Table 3.2. For each detected constraint, we check whether the constraint is already defined in the database, and only report those that are not defined. The total number of constraints found, as well as the breakdown by their source (as

described in Sec. 3.2.2) is shown in Table 3.3, and the breakdown of constraint type (as described in Sec. 3.2.5) is shown in Table 3.4.

Table 3.3: Number of constraint detected, grouped by source.

data validation	association define	field define	data update	**SUM**
689	91	69	74	923

Table 3.4: Number of constraint detected, grouped by type.

categorical	not null	uniqueness	length	format	equal	non-equal	*other*
236	219	104	94	59	23	19	169

3.5.4 Optimization detection

In this experiment we present how many queries can be optimized using the constraints detected by CONSTROPT. We run CONSTROPT on each application to rewrite queries following the constraint detection. In this experiment, if a piece of code containing a query to be optimized can potentially be invoked by multiple actions, we only count the query once. For each query optimized, we manually check that the optimization leverages at least one constraint detected by CONSTROPT and is not defined in the database, so all the optimizations listed here are not possible no matter which database is used by the application. We present the number of queries that can be optimized in Table 3.5. Each row represents a type of optimization described in Sec. 3.3 and each column shows an application. The last column shows the sum of all applications.

The result shows that every type of optimization can be found multiple times in each application. Even though that each application contains a large number of queries, most queries are relatively very simple like finding tuple by id. Many of the described optimizations are more likely to apply to complicated queries (e.g., remove distinct, remove join, etc.), and the number of cases is significant among those queries. For example, every application has only 14 queries with `DISTINCT` keyword

and on average 1.5 of them can be optimized by removing `DISTINCT`. Besides, two optimizations are quite prevalent in all applications: *add parameter precheck* and *change storage format* On average more than 8 queries can be optimized in each application. These cases are found in the most recent version of these applications where developers have already carefully tuned the queries.

Furthermore, as all the queries accounted for in Table 3.5 are detected by static analysis, they can potentially be invoked multiple times during runtime when generating one webpage with different parameters (e.g., a query inside a loop) or be invoked under different webpages, so the number of queries that can be optimized when running the application can potentially be much more than the numbers presented in Table 3.5.

Table 3.5: Optimizations detected across 10 applications.

#App	Re	Gi	Op	Sp	Di	Da	Ma	Lo	Dv	On	**SUM**
RD	3	2	1	5	1	1	0	1	1	0	**15**
AL	2	1	7	4	1	1	1	2	1	3	**23**
RP	7	1	1	2	2	1	2	1	5	2	**24**
RJ	3	1	1	2	1	2	0	2	0	1	**13**
DU	4	4	2	1	1	0	1	0	0	2	**15**
AP	12	8	2	2	14	4	3	1	28	16	**90**
SF	19	54	20	42	0	28	7	14	56	16	**256**
SUM	50	71	34	58	20	37	13	19	91	40	436

RD: Remove distinct AL: Add limit 1

RP: Remove predicate RJ: Remove join

DU: Replace disjunctions with unions' AP: Add parameter precheck

SF: Change storage format

3.5.5 Performance evaluation

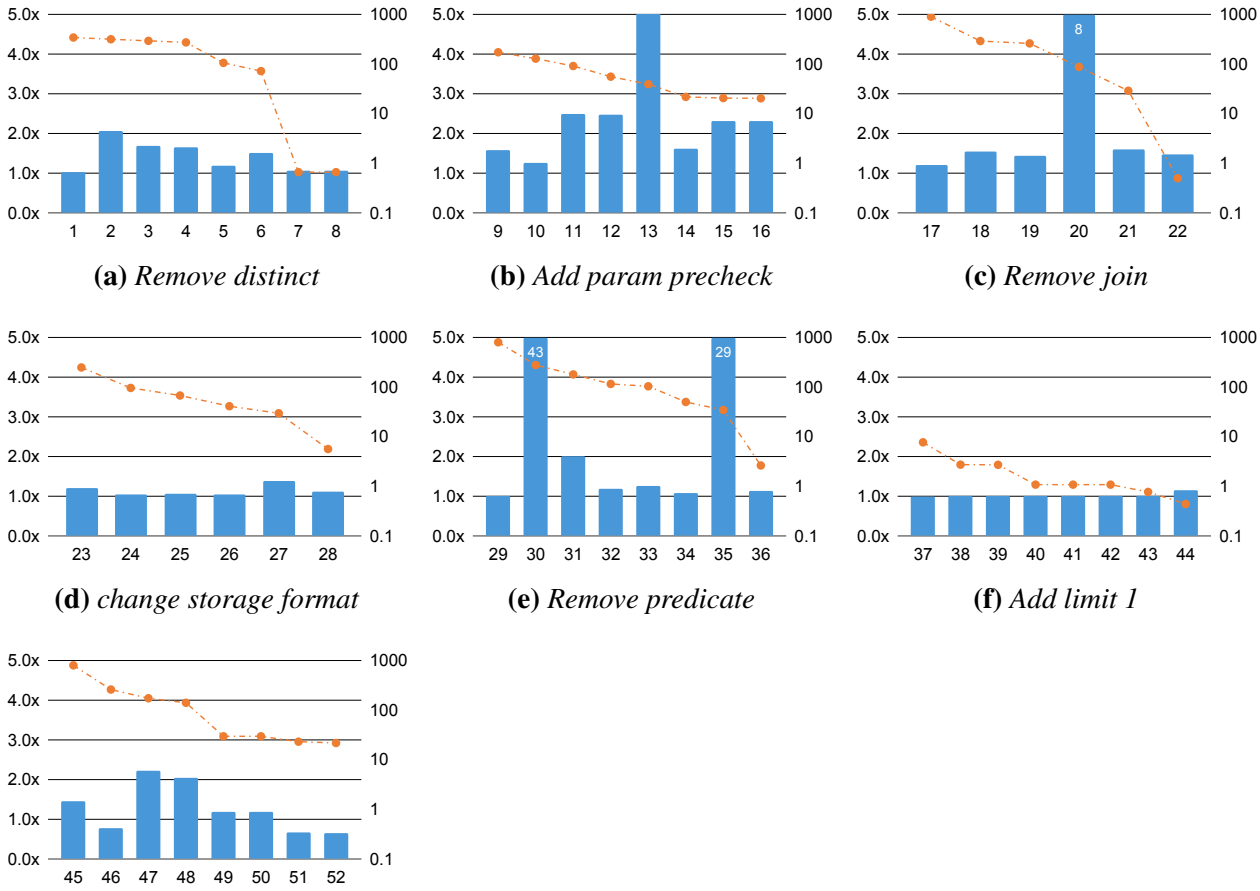
3.5.5.1 Query performance improvement

We first evaluate SQL query performance before and after optimization. In this experiment, due to the large number of queries, we select one application from each category to evaluate. The selection of applications is based on the best coverage of optimization types (e.g., we choose *Redmine* in the project management category because it has more cases in 5 out of 7 optimization types than the other applications in the same category). This selection leads to four applications selected: *Redmine* from project management, *Spree* from e-commerce, *Onebody* from social network and *Discource* from forum.

To evaluate query performance, we populate the application's database with synthetic data using the methodology described in prior work [155]. Specifically, we collect real-world statistics of each application based from its public website or similar website to scale data properly as well as to synthesize database contents and application-specific constraints. We scale the size of the application data from 5 to 10GB, which is close to the size of data reported by the the application developers [17, 18]. This results in around 10K to 1M records in the tables involved in the queries we evaluate.

We compare the query execution time of each SQL query before and after optimization grouped by optimization type. If one type contains over 10 queries under one application, we randomly sample 8 cases in total to evaluate. In this experiment we only apply one optimization to each query. If multiple optimizations can be applied on a specific query, we apply them and report the speedup separately. For three applications, Redmine, Onebody and Spree, we evaluate using two databases, MySQL and PostgreSQL. For one application, Discourse, we only evaluate with PostgreSQL since this application does not support using MySQL as backend. The evaluation with PostgreSQL is shown in Figure 3.5 and with MySQL is shown in Figure 3.6.

The results show that many types of optimization can substantially improve query performance. For example, removing joins can accelerate a query to up to over 7×, removing predicate can achieve speedup up to over 80×, removing `DISTINCT` up to 1.6×, and replacing disjunctions with



(a) *Remove distinct* **(b)** *Add param precheck* **(c)** *Remove join*

(d) *change storage format* **(e)** *Remove predicate* **(f)** *Add limit 1*

(g) *Replace disjunctions*

Figure 3.5: Query time evaluation with PosgreSQL. Each figure shows one type of optimization. Each bar shows the speedup of one query (left y axis) and the dot on line above the bar shows the actual time of the original query in millisecond (right y axis). X-axis shows the number of query.

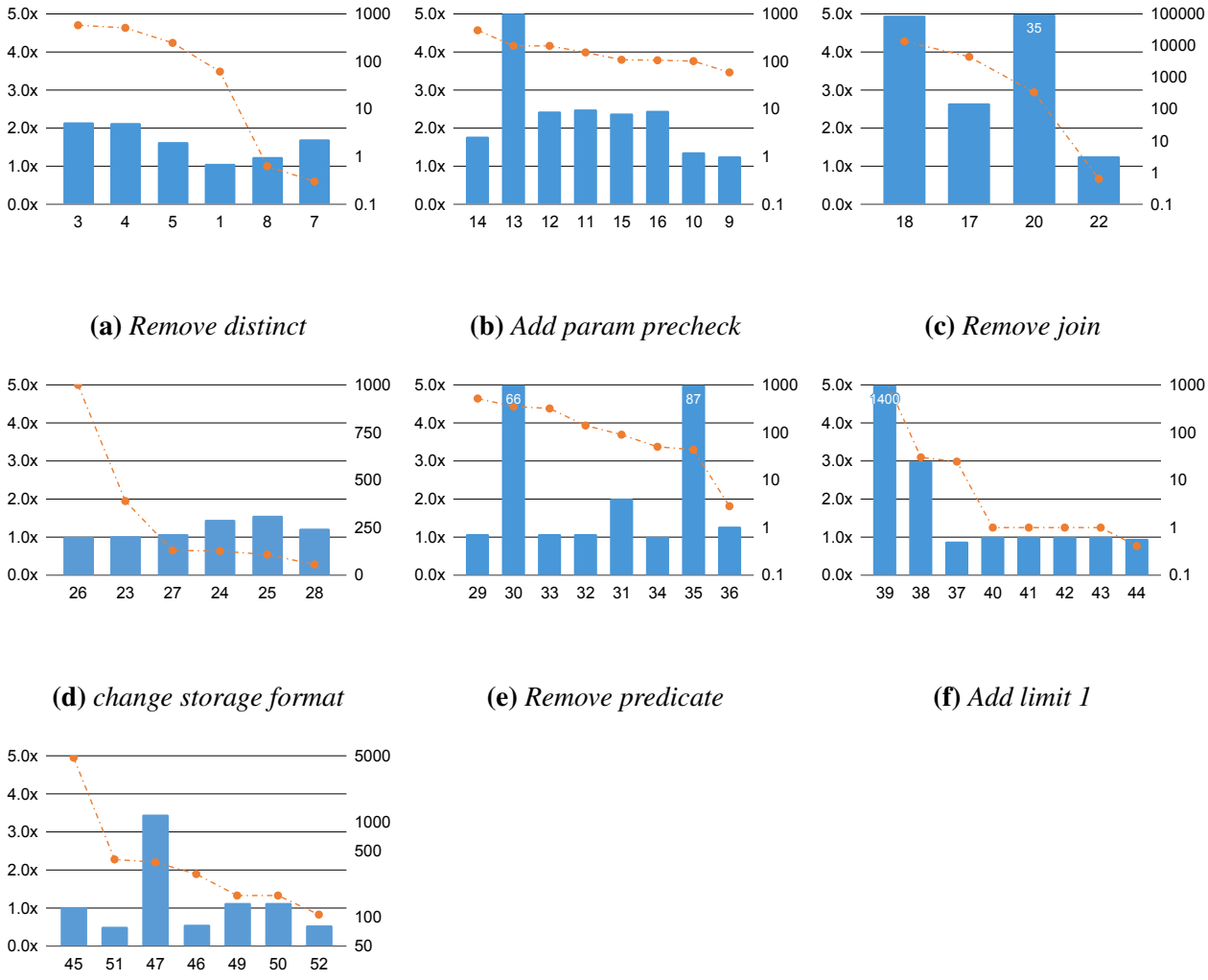
unions up to 2× in both PostgreSQL and MySQL. These speedups often come from much less computation due to a changed query plan, as we explained in Sec. 3.3. *Add limit 1* provides significant performance gain for two queries in MySQL because it avoids scanning the entire table by returning the result early. *Add parameter precheck* provides around 1.25× speedup.

In the above evaluation of `adding parameter precheck`, we assume 20% of the user-input parameter is invalid. However, we are not aware of the percentage of invalid inputs in real world application deployment. To better understand this optimization, we randomly sampled 3 queries and evaluated them under 5 different sets of invalid input percentage: 0%, 20%, 40%, 60%, 80%. The result is shown in Figure 3.7. The speedup of the three queries show very similar trend as the percentage of invalid input increases. The larger the percentage is, the more speedup it achieves. When there are 80% input is invalid, the average speed up reaches to 5×. When there are no invalid input at all, there is no speedup because all queries are necessary and no query can be avoided, but the overall performance is not hurt since the checking is fast (all within 0.1ms).

3.5.5.2 Webpage performance improvement

Although we can see that individual queries can be greatly accelerated from query evaluation, we do not know how these improvements affect overall application performance. Because application developers are mostly concerned about webpage loading time as it has direct impact on user experience, we evaluate how much the optimizations can improve end-to-end webpage loading time. In this experiment, we focus on webpages with slow queries. In particular, from all queries evaluated in Figure 3.5, we choose the top-two slowest queries (before optimization) from each optimization type, and remove queries that take fewer than 100ms as these queries are often triggered in fast webpages which developers care less about optimizing. For each selected query (10 queries in total), we trace the webpage that contains the query. This leads to 10 webpages across the 4 applications. We apply CONSTROPT on these 4 applications and evaluate the end-to-end load time (i.e., the time from web user entering the URL till the webpage is rendered on her browser) of these 10 webpages.

The result is shown in Figure 3.8. It shows that a few pages are significantly accelerated (3 pages over 2×), and the speedup is much greater for the backend (i.e., when the network and webpage



(a) *Remove distinct*

(b) *Add param precheck*

(c) *Remove join*

(d) *change storage format*

(e) *Remove predicate*

(f) *Add limit 1*

(g) *Replace disjunctions*

Figure 3.6: Query time evaluation with MySQL.

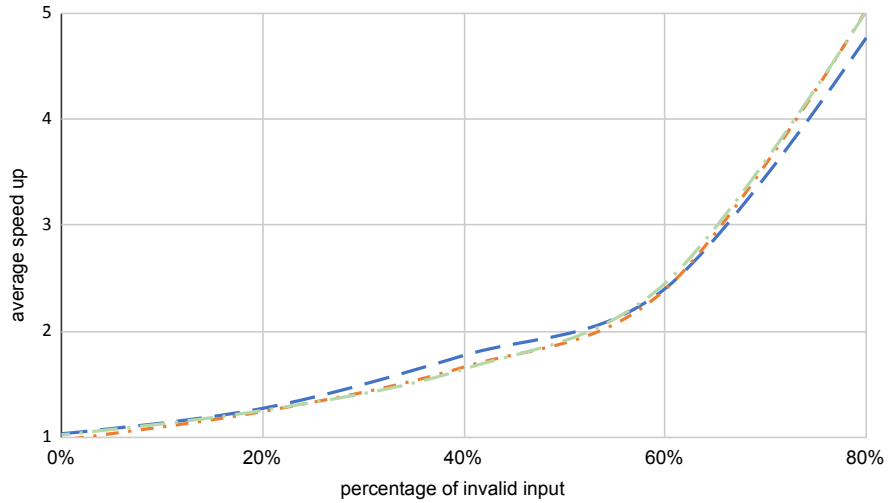


Figure 3.7: Speed up under different percentage of invalid input

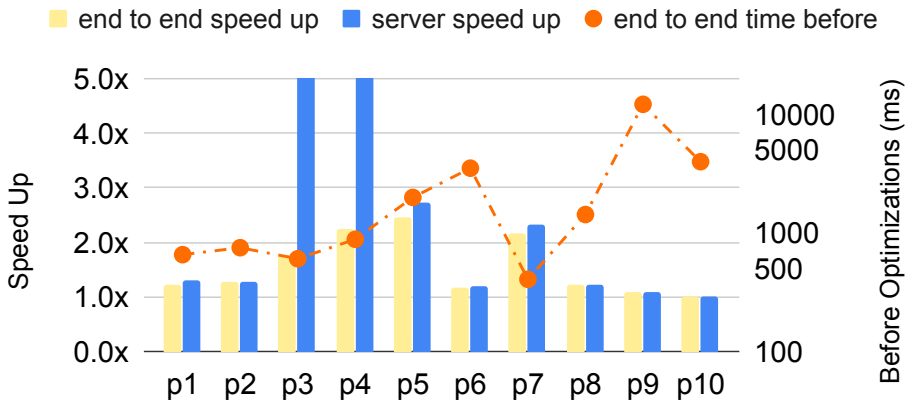


Figure 3.8: Evaluation on webpage performance. It shows the speedup of end-to-end webpage load time, the speedup of server time (the time spent on application server and the database), and the absolute time of the end-to-end page loading time.

rendering time is not considered). Interestingly, compared to the single-query speedup of the queries we selected, in 5 pages the end-to-end speedup is greater than the individual query speedup. This is because each page contains multiple queries that are optimized. The total number of optimized queries for each page is shown in Table 3.6.

Table 3.6: Page detail.

Page	App	origin query (Figure 3.5)	page description	# queries optimized	Page	App	origin query (Figure 3.5)	page description	# queries optimized
p1	spree	1	list orders	4	p6	redmine	18	update the issue	4
p2	redmine	2	list time entries	1	p7	redmine	29	list context menus	1
p3	spree	9	list users	2	p8	onebody	30	search users	2
p4	spree	10	list orders	2	p9	redmine	45	list issue metadata	5
p5	redmine	17	list issues	2	p10	spree	46		1

3.6 Summary and Future Work

In this chapter we present CONSTROPT, a tool to detect data constraints from application code and optimize queries using these constraints. CONSTROPT infers constraints by looking into how the application defines and processes application data, and develops a verification-based method for to optimize queries leveraging a wide range of constraints. Our experiments show that CONSTROPT is able to discover a large number of constraints and optimizes many application queries with speedup up to 43×.

The current prototype of CONSTROPT rewrites queries offline and replaces queries online, thus is not able to optimize queries outside the application, for instance, queries written in SQL sent by the application administrator. It remains future work to push the constraint to the database and performs the rewrite online. Besides query optimization, the constraints inferred by CONSTROPT can be leveraged to reduce data storage cost, optimize data compression, improve cardinality estimation, and many more to explore.

Chapter 4

QURO: IMPROVING TRANSACTION PERFORMANCE BY QUERY REORDERING

While much work has focused on designing efficient concurrency control mechanisms for transaction processing applications, not much has been done on understanding how the transactions issue queries and leveraging application semantics to improve application performance. In this section, we present QURO, a query-aware compiler that automatically reorders queries in transaction code to improve performance. Observing that certain queries within a transaction are more contentious than others as they require locking the same tuples as other concurrently executing transactions, QURO automatically changes the application such that contentious queries are issued as late as possible. We have evaluated QURO on various transaction benchmarks, and our results show that QURO-generated implementations can increase transaction throughput by up to 6.53×, while reduce transaction latency by up to 85%.

4.1 Query Order and Transaction Performance

In this work, we focus on the transaction processing aspect of database-backed applications. Transactions are widely used in these applications to protect the atomicity and consistency of a set of data access. For instance, an e-commerce applications use transactions for checkout and payment, social network applications use transactions for relationship updates, etc. When a large number of application users invokes the transactions on a hotspot (i.e., a record often touched by many concurrent transactions), some form of concurrency control must be implemented to ensure that all transactions get a consistent view of the persistent data. For example, when many users purchase a popular item in a short period on a sales event, the concurrency control ensures that the item is not oversale.

Two-phase locking (2PL) [79, 78] is one of the most popular concurrency control mechanisms implemented by many DBMSs. In 2PL, each data element (e.g., a tuple or a partition) stored in the database is associated with a read and a write lock, and a transaction is required to acquire the appropriate lock associated with the given database element before operating on it. For example, while multiple transactions can be holding the read lock on the same data concurrently, only one transaction can hold the write lock. When a transaction cannot acquire a lock, its pauses execution until the lock is released. To avoid deadlocks, (strict) 2PL requires that a transaction not request additional locks once it releases any lock. Thus, as each transaction executes, it goes through an expanding phase where locks are acquired and no lock is released, followed by a shrinking phase where locks are released and no locks are acquired.

Unfortunately, not all locks are created equal. While each transaction typically operates on different elements stored in the DBMS, it is often the case that certain elements are more contentious than others, i.e., they tend to be read from or written to by multiple concurrent transactions. As an illustration, imagine an application where all transactions need to update a single tuple (such as a counter) among other operations. If each transaction starts by first acquiring the write lock on the counter tuple before acquiring locks on other data elements, then essentially all but one of the transactions can make progress while the rest are blocked, even though other transactions could have made further progress if they first acquired locks on other data. As a result, each transaction takes a longer time to execute, and the overall system throughput suffers. This is exacerbated in main-memory databases. Since the transaction no longer need to access the disk, most of transaction running time is spent on executing queries, and the long lock waiting time is likely to become the predominant performance bottleneck.

One way to avoid the above problem is to reorder the queries in each transaction such that operations on the most contentious data elements are performed last. Indeed, as our results show, doing so can significantly improve application performance. Unfortunately, reordering queries in transaction code raises various challenges:

- OLTP applications are typically written in a high-level programming language such as C or Java, and compilers for these languages treat queries as black-box library calls. As such, they are

unaware of the fact that these calls are executing queries against the DBMS, let alone ordering them during compilation based on the level of contention.

• DBMS only receives queries from the application as it executes and does not understand how the queries are semantically connected. As such, it is very difficult for the DBMS to reorder queries during application execution, since the application will not issue the next query until the results from the current one have been returned.

• Queries in a transaction are usually structured based on application logic. Reordering them manually will make the code difficult to understand. Furthermore, developers need to preserve the data dependencies among different queries as they reorder them, making the process tedious and error-prone.

In this chapter we present QURO, a query-aware compiler that automatically reorders queries within transaction code based on lock contention while preserving program semantics. To do so, QURO first profiles the application to estimate the amount of contention among queries. Given the profile, QURO then formulates the reordering problem as an Integer Linear Programming (ILP) problem, and uses the solution to reorder the queries and produces an application binary by compiling the reordered code using a standard compiler.

QURO makes the following contributions:

• We observe that the order of queries in transaction code can drastically affect performance of OLTP applications, and that current general-purpose compiler frameworks and DBMSs do not take advantage of that aspect to improve application performance.

• We formulate the query reordering problem using ILP, and devise a number of optimizations to make the process scale to real-world transaction code.

• We implemented a prototype of QURO and evaluated it using popular OLTP benchmarks. When evaluated on main-memory DBMS implementations, our results show that the QURO-generated transactions can improve throughput by up to 6.53×, while reducing the average latency of individual transactions by up to 85%.

4.2 Quro *Overview*

Now we discuss query reordering in transaction code using an example and describe the architecture of QURO. To motivate, Listing 4.1 shows an excerpt from an open-source implementation [19] of the payment transaction from the TPC-C benchmark [145], which records a payment received from a customer. In the excerpt, the code first finds the warehouse to be updated with payment on line 1, and subsequently updates it on line 2. Similarly, the district table is read and updated on lines 3 and 4. After that the code updates the customer table. The customer can be selected by customer id, or customer name. If the customer has good credit, only the customer balance will be updated, otherwise the detail of this transaction will be appended to the customer record. Finally it inserts a tuple into the history table recording the change.

Listing 4.1: Original code fragment from TPC-C payment transaction. Here`select("t", v)` represents a selection query on table `t` that uses the value of program value `v` as one of its parameters, likewise for `update` and `insert`.

```
1  w_name = select("warehouse");
2  update("warehouse", w_name);
3  d_name = select("district");
4  update("district");
5  if (c_id == 0) {
6    c_id = select("customer", c_name);
7  }
8  c_credit = select("customer", c_id);
9  if (c_credit[0] == ('G')) {
10   update("customer", c_id, w_id);
11 } else {
12   c_id = "..." + w_id + c_id + "...";
13   update("customer", c_id);
14 }
15 insert("history", w_name, d_name, ...);
```

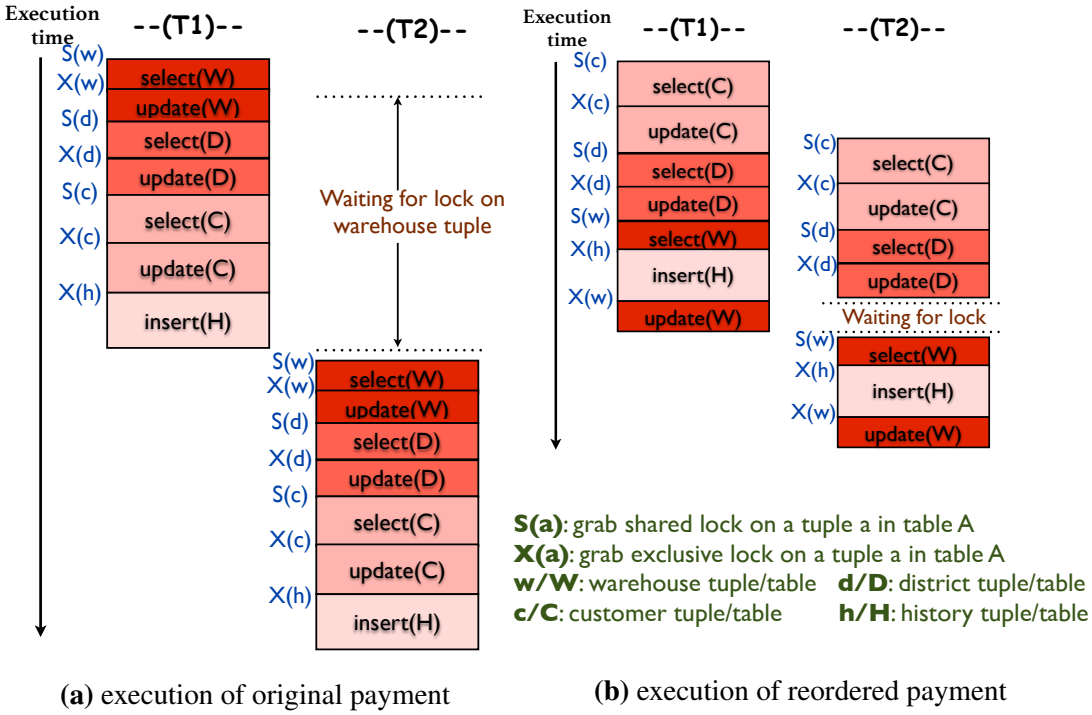
As written, the implementation shown in Listing 4.1 performs poorly due to high data contention.

In a typical TPC-C setup, the warehouse table contains the fewest number of tuples. Hence, as the number of concurrent transactions increases, the chance that multiple transactions will update the same warehouse table tuple also increases, and this will in turn increase lock contention and the amount of time spent in executing each transaction. This is illustrated pictorially in Figure 4.1a with two concurrent transactions that try to update the same tuple in the warehouse table. When executed under 2PL, each transaction attempts to acquire the exclusive lock on the same warehouse tuple before trying to update it, and will only release the lock when the transaction commits. In this case T1 acquires the lock, blocking T2 until T1 commits. Thus the total amount of time needed to process the two transactions is close to the sum of these two transactions executed serially.

Listing 4.2: Reordered code fragment from Listing 4.1

```
if (c_id == 0) {
  c_id = select("customer", c_name);
}
c_credit = select("customer", c_id);
if (c_credit[0] == ('G')) {
  update("customer", c_id, w_id);
}else{
  c_id = "..." + w_id + c_id + "...";
  update("customer", c_data);
}
d_name = select("district");
update("district");
w_name = select("warehouse");
insert("history", w_name, d_name, ...);
update("warehouse", w_name);
```

However, there is another way to implement the same transaction, as shown in Listing 4.2. Rather than updating the warehouse (i.e., the most contentious) table first, this implementation updates the customer's balance first, then updates the district and warehouse tables afterwards. This implementation has the same semantics as that shown in Listing 4.1, but with very different performance characteristics, as shown in Figure 4.1b. By performing the updates on warehouse



(a) execution of original payment

(b) execution of reordered payment

Figure 4.1: Comparison of the execution of the original and reordered implementation of the payment transaction. The darker the color, more likely the query is going to access contentious data.

table at a later time (line 15), transaction T1 delays acquiring the exclusive lock on the warehouse tuple, allowing T2 to proceed with operations on other (less contentious) tuples concurrently with T1. Comparing the two implementations, reordering increases transaction concurrency, and reduces the total amount of time needed to execute the two transactions.

While reordering the implementation shown in Listing 4.1 to Listing 4.2 seems trivial, doing so for general transaction code is not an easy task. In particular, we need to ensure that the reordered code does not violate the semantics of the original code in terms of data dependencies. For instance, the query executed on line 15 in Listing 4.1 can only be executed after the queries on lines 1 and 3, because it uses `w_name` and `d_name`, which are results of those two queries. Besides such data dependencies on program variables (as `w_name` and `d_name` mentioned above), there may also be

dependencies on database tuples. For example, the query on line 3 reads a database tuple which the query on line 4 later updates, so the two queries have a data dependency on that tuple. While such dependencies can be inferred manually, doing so for longer and more complex transactions puts excessive burden on developers. Unfortunately, typical compilers do not perform such aggressive transformations, as they treat queries as external function calls.

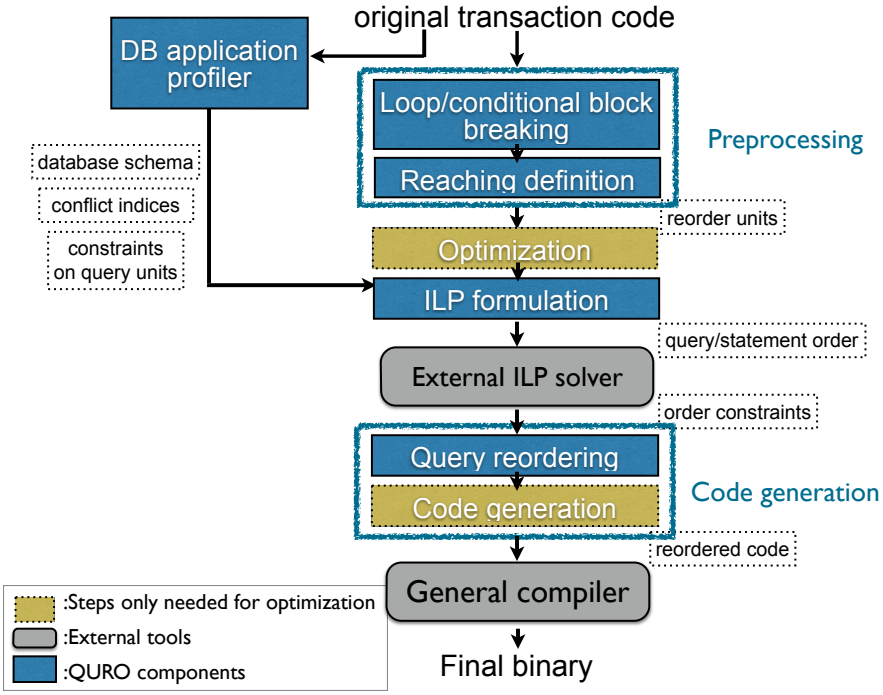


Figure 4.2: Architecture and workflow of QURO

QURO is designed to optimize transaction code by reordering query statements according to the lock contention each query incurs. To use QURO, developer first demarcates each transaction function with `BEGIN_TRANSACTION` and `END_TRANSACTION`.[1] Our current prototype is built on Clang and accepts transaction code written in C/C++, and QURO assumes that the transactions use standard APIs to issue queries to the DBMS (e.g., ODBC).

The architecture of QURO is shown in Figure 4.2. After parsing the input code, QURO first

[1] QURO currently assumes that each transaction is implemented within a single function and leave inter-procedural analysis as future work.

generates an instrumented version to profile the running time of each query and gathers information about query contention. QURO deploys the instrumented version using the same settings as the original application and runs it for a user-specified amount of time. After profiling, QURO assigns a contention index to each query to be used in subsequent reordering steps. QURO also collects information about database schema, which is used to generate order constraints.[2]

After profiling, QURO performs a number of preprocessing steps on the input code. First, it performs reaching definition analysis for each transaction function. Reaching definition analysis is used to infer data dependencies among different program variables. After that, QURO performs loop fission and breaks compound statements (e.g., conditionals with multiple statements in their bodies) into smaller statements that we refer to as *reorder units*. This is to expose more reordering opportunities, to be discussed in Sec. 4.3.

QURO next uses the results from profiling and preprocessing to discover order constraints on queries before reordering. Data dependencies among program variables or database tuples may induce order constraints. QURO first uses the results from reaching definition analysis during preprocessing to construct order constraints based on program variables. QURO then analyzes the queries with the database schemas to infer order constraints among database tuples, e.g., if two queries may update the same tuple in the same table, the order of these queries cannot be changed. Reordering is then formulated as an ILP problem based on the ordering constraints, and solving the program returns the optimal way to implement the transaction subject to the data dependencies given the contention indices. While a simple implementation is to encode each reorder unit as a variable in the ILP, solving the ILP might take a substantial amount of time, especially for transactions with many lines of code. In Sec. 4.4.4 we propose an optimization to make this tractable and more efficient. After receiving the order of queries from ILP solver, QURO restructures the program, and uses a general-purpose compiler to produce the final binary of the application.

In the next sections we discuss each step involved in the reordering process in detail.

---

[2] QURO assumes the database schema doesn't change when transactions are running.

4.3 Preprocessing

Before statement reordering, QURO parses the input transaction code into an abstract syntax tree (AST) and performs two preprocessing tasks: breaking the input code into small units to be reordered, and analyzing the data dependencies among program variables. In this section we describe the details of these two steps.

4.3.1 Breaking loop and conditional statements

The purpose of the first task is to enable more queries to be reordered. For loop and conditional statements, it is hard to change the ordering of statements within each block, as each such statement can be nested within others. Disregarding the bodies inside loop and conditional statements and treating the entire statement as one unit limits the number of possible ways that the code can be reordered. In fact, as we will demonstrate in Sec. 4.6, breaking loop and conditional statements is essential to improve the performance for many transaction benchmarks.

For loop statements, QURO applies loop fission, a well-studied code transformation technique [149], to split an individual loop nest into multiple ones. The basic idea is to split a loop with two statements S_1 and S_2 in its body into two individual loops with the same loop bounds if:

- There is no loop carry dependency. If S_1 defines a value that will be used by S_2 in later iterations, then S_1 and S_2 have to reside in the same loop.
- There is no data dependency between the two statements to be split. If S_1 defines a value that is used by S_2 in the same iteration, and S_1 will rewrite that value in some later iteration, then S_1 and S_2 cannot be split.
- The statements do not affect the loop condition. If S_1 writes some value that affects the loop condition, then all the statements within the loop cannot be split into separate loops.

Similarly, conditional statements under a single condition can also be split. In general, conditional statements S_1 and S_2 under the same Boolean condition can be split into multiple ones with the same condition if:

- Neither statement affects the condition.

• The condition does not have any side effects, for example, changing the value of a program variable that is used by any other statements in the program.

Listing 4.3: Loop fission example

```
1 for(i=0; i<n; i++){
2   var1 = select("table1");
3   update("table1", var1+1);
4   var2[i] = select("table2");
5   update("table2", var2[i]+1);
6 }
```

Listing 4.4: Code after fission

```
1 for(i=0; i<n; i++){
2   var1 = select("table1");
3   update("table1", var1+1);
4 }
5 for(i=0; i<n; i++){
6   var2[i] = select("table2");
7 }
8 for(i=0; i<n; i++){
9   update("table2", var2[i]+1);
10 }
```

Listing 4.5: Break conditional statement

```
1 if(c_credit[0] == ('G')) {
2   update("customer", c_id, w_id);
3 }
4 if(!(c_credit[0] == ('G')){
5   c_id = "..." + w_id + c_id + "...
      ";
6 }
7 if(!(c_credit[0] == ('G')){
8   update("customer", c_id);
9 }
```

As an illustration, breaking the conditional block will transform line 9 to line 14 from Listing 4.1 into Listing 4.5.

4.3.2 Analyzing reaching definitions

After breaking loop and conditional statements, QURO analyzes the data dependencies among statements by computing reaching definitions. Formally speaking, a definition of a program variable v by program statement S_1 *reaches* a subsequent (in terms of control flow) statement S_2 if there is a program path from S_1 to S_2 without any intervening definition of v. We compute reaching

definitions for each program statement using a standard data-flow algorithm [128]. The process is straightforward for most types of program statements. For function calls, however, we distinguish between those that are database calls (e.g., those that issue queries), for which we precisely model the def-use relationships among the function parameters and return value, and other functions, for which we conservatively assume that all parameters with pointer types or parameters that are passed by reference are both defined and used by the function.

4.4 Reordering Statements

The preprocessing step normalizes the input code into individual statements that can be rearranged. We call each such statement, including an assignment, a function call statement, or a loop/conditional block that cannot be further decomposed using methods as described in Sec. 4.3.1, a *reorder unit*. In this section we discuss how we formulate the statement reordering problem by making use of the information collected during preprocessing.

4.4.1 Generating order constraints

As discussed in Sec. 4.2, the goal of reordering is to change the structure of the transaction code such that the database queries are issued in an increasing order of lock contention. However, doing so is not always possible because of data dependencies among the issued queries. For instance, the result of one query might be used as a parameter in another query, or the result of one query might be passed to another query via a non-query statement. Furthermore, two queries might update the same tuple, or a query might update a field that is a foreign key to a table involved in another query. In all such cases the two queries cannot be reordered even though one query might be more contentious than the other.

Formally, we need to preserve the data dependencies 1) among the program variables, and 2) among the database tuples, when restructuring queries in transaction code.[3] The reaching definition analysis from preprocessing infers the first type of data dependency, while analyzing the queries

[3] QURO currently does not model exception flow. As such, the reordered program might have executed different number of statements when an exception is encountered as compared to the original program.

using database schema information infers the second type. These data dependencies set constraints on the order of reorder units. In the following we discuss how these constraints are derived.

Dependencies among program variables:

1. Read-after-write (RAW): Reorder unit U_i uses a variable that is defined by another unit U_j. *Formal constraint*: Reorder unit U_i should appear before U_j in the restructured code.
2. Write-after-read (WAR): U_j uses a variable that is later updated by another unit U_k. *Formal constraint*: If both U_i and U_k define the same variable v, and U_j uses v defined by U_i, then U_k cannot appear between U_i and U_j. If no such U_i exists, as in the case of v being a function parameter that is used by U_j, then U_k should appear after U_j in the restructured code.
3. Write-after-write (WAW): v is a global variable or a function parameter that is passed by reference, and both U_i and U_l define v, with U_l being the last definition in the body of the function. *Formal constraint*: U_i should appear before U_l in the restructured code. If v is a global variable, we assume that program locks are in place to prevent race conditions.

We use the code shown in Listing 4.1 to illustrate the three kinds of dependency discussed above. For instance, the insertion into the history table on Line 15 uses variable `w_name` defined on Line 1 and `d_name` defined on Line 3. Thus, there is a RAW dependency between line 15 along with lines 1 and 3. Hence, a valid reordering should always place line 15 after lines 1 and 3. Meanwhile, the update on customer table on line 13 uses variable `c_id`, which is possibly defined on line 6 or is passed in from earlier code. Furthermore, line 12 redefines this variable. Thus, there is a WAR dependency between line 12 and line 13, meaning that in a valid ordering line 13 should not appear between line 6 and line 13.

Dependencies among database tuples:

1. Operations on the same table: queries Q_i and Q_j operate on the same table, and at least one of the queries performs a write (i.e., update, insert, or delete).
2. View: query Q_i operates on table T_i which is a view of table T_j, and query Q_j operates on T_j. At least one of the queries performs a write.

3. Foreign-key constraints: Q_i performs an insert/delete on table T_i, or change in the key field C_i of T_i. Q_j operates on column C_j in table T_j, where C_j is a foreign key to T_i which includes a column that column C_i in T_i references.
4. Triggers: Q_i performs an insert or a delete on table T_i, which triggers a set of pre-defined operations that alter T_j. Query Q_j operates on table T_j.

   Formal constraint: In all of the above cases, the order of Q_i and Q_j after reordering should remain the same as in the original program.

For the code example in Listing 4.1, the queries on line 1 and line 2 operate on the same table, so they cannot be reordered or else the query on line 1 will read the wrong value.

To discover the dependencies among database tuples, QURO analyzes each database query from the application to find out which tables and columns are involved in each query. Then QURO utilizes the database schema obtained during preprocessing to discover the dependencies listed above.

4.4.2 Formulating the ILP problem

QURO formulates the reordering problem as an instance of ILP. As mentioned in Sec. 4.2, QURO first profiles the application to determine how contentious the queries are among the different concurrent transactions. Profiling gives a conflict index c to each query: the larger the value, the more likely that the query will have data conflict with other transactions. The conflict index for non-query statements is set to zero. Under this setting, the goal of reordering is to rearrange the query statements in ascending conflict index, subject to the order constraints described above.

Concretely, assume that there are n reorder units, U_1 to U_n, in a transaction, with conflict indices c_1 to c_n, respectively. We assign a positive integer variable p_i to represent the final position of each of the n reorder units. The order constraints derived from data dependencies can be expressed as the following constraints in the ILP problem:

- $p_i \leq n, i \in [1, n]$, such that each unit is assigned a valid position.
- $p_i \neq p_j, i \neq j$, such that each unit has a unique position.
- $p_i < p_j$, if there is a RAW dependency between U_i and U_j.

• $(p_k < p_i) \mid (p_k > p_j)$, if there is a WAR dependency between U_j, U_k and U_i, where U_i redefines a variable that U_j uses.

• $p_k > p_j, k \neq j$, if there is a WAR dependency between U_j and U_k, and and there is no intervening variable redefinition.

• $p_l > p_i, i \neq l$, if there is a WAW dependency between U_l and U_i, and U_l is the last definition of a global variable or a return value in the transaction.

• $p_i < p_j, i < j$, if U_i contains query Q_i, query Q_j is in reorder unit U_j, and the order of Q_i and Q_j needs to be preserved due to data dependencies among database tuples referenced by Q_i and Q_j.

Given these constraints, the objective of the ILP problem is to maximize $\sum_{i=1}^{n} p_i * c_i$. Solving the program will give us the value of $p_1, \ldots, p_n$, which indicates the position of each reorder unit in the final order.

As an example, the code shown in Listing 4.1 generates the following constraints. Here we assume that there are no view, trigger or foreign-key relationships between any two tables used in the transaction:

$p_i \leq 11, i \in \{1, 2, 3, 4, 6, 8, 10, 11, 13, 14, 16\}$

$p_i \neq p_j, i \neq j$

$p_1 < p_{15}$, RAW on variable `w_name`

$p_3 < p_{15}$, RAW on variable `d_name`

$\vdots$

$(p_{12} < p_6) \mid (p_{12} > p_8)$, WAR on variable `c_id`

$(p_{12} < p_6) \mid (p_{12} > p_{10})$, WAR on variable `c_id`

$\vdots$

$p_1 < p_2$, Query order constraint, as both query

Q_1(in U_1) and Q_2(in U_2) operate on the warehouse table

$p_3 < p_4$, Query order constraint, as both query

Q_3(in U_3) and Q_4(in U_4) operate on the district table

With the conflict indices for query-related units as shown in Table 4.1, QURO will give an

reordering of the units shown in Listing 4.2.

Table 4.1: conflict index for each reorder units in Listing 4.1

unit	1	2	3	4	6	8	10	13	15
c	500	510	100	110	50	50	60	60	10

4.4.3 Removing unnecessary dependencies

Solving ILP problems is NP-hard in general. In the following sections we describe two optimizations to reduce the number of constraints and variables in the ILP problem, and we evaluate these techniques in Sec. 4.6.9.

First, we describe a technique to reduce the number of ILP constraints. Consider the example shown in Listing 4.6.

Listing 4.6: Code example of renaming

```
1 v = select("table1");
2 update("table2", v);
3 v = select("table3", v);
```

Listing 4.7: Code example after renaming

```
1 v = select("table1");
2 v_r = v;
3 v_r = select("table3", v_r);
4 update("table2", v);
```

If the update on line 2 is more contentious than the query on line 3, then QURO's reordering algorithm would place line 3 before line 2. However, RAW and WAR lead to the following constraints:

$$p_1 < p_2;\ p_1 < p_3; \qquad \text{(RAW)}$$

$$(p_3 < p_1) \vee (p_3 > p_2); \qquad \text{(WAR)}$$

meaning that reordering will violate data dependency. However, we can remove the WAR dependency with variable renaming by creating a new variable `v_r`, assigning `v` to it before `v` is used, and replacing subsequent uses of `v` to be `v_r`. This allows us to restructure the code to that shown in Listing 4.7.

In general, WAR and WAW are *name* dependencies (in contrast to *data* dependencies, as in the case of RAW) that can be removed by renaming. Doing so reduces the number of ILP constraints, and makes more queries able to be reordered. As shown in the example above. However, if the variable v involved in a WAR or WAW dependency satisfies any of the following conditions, then removing WAR and WAW will be more complicated:

- v *is not of a primitive type (e.g., a pointer or class object).* Since renaming requires cloning the original variable, it might be impossible to do so for non-primitive types as they might contain private fields. Besides, cloning objects can slow down the program. Thus, we do not rename non-primitive variables and simply encode any WAW and WAR dependencies involving these variables in the ILP.

- v *is both used and defined in the same reorder unit.* If the same variable is both defined and used in the same reorder unit, such as `f(v)` where v is passed by reference, then replacing v with `v_r` in the statement will pass an uninitialized value to the call. To handle this issue, the value of v should be first copied to `v_r` before the call. This is done by inserting an assign statement before the statement containing the variable to be renamed: `v_r = v; f(v_r)`.

- *Multiple definitions reach the same use of* v. In this case, if any of the definitions is renamed, then all other definitions will need to be renamed as well.

Listing 4.8: Renaming example with multiple reaching definitions

```
1 v=select("table1");
2 if(cond)
3  v=select("table2");
4 update("table3", v);
```

Listing 4.9: Example after renaming

```
1 v=select("table1");
2 v_r2=v;
3 if(cond){
4  v_r1=select("table2");
5  v_r2=v_r1;}
6 update("table3", v_r2);
```

We use the example in Listing 4.8 to illustrate. The definitions of v on lines 1 and 3 both reach the update on line 4. If v needs to be renamed on line 3 due to data dependency violation (not shown in the code), then we create a new variable `v_r2` to hold the two definitions so that both `v_r1` and v

reach the use at the update query, as shown in Listing 4.9.

4.4.4 Shrinking problem size

Transactions that implement complex program logic can contain many statements, which generate many reordering units and variables in the ILP problem. This can cause the ILP solver to run for a long time. In this section we describe an optimization to reduce the number of variables required in formulating the ILP problem.

Since our goal is to reorder database query related reorder units, we could formulate the ILP problem by removing all variables associated with non-query related reorder units from the original formulation. This does not work, however, as dropping such variables will mistakenly remove data dependencies among query related reorder units as well. For example, suppose query Q_3 contained in U_3 uses as parameter the value that is computed by a reorder unit U_2 containing no query, and U_2 uses a value that is returned by query Q_1 in U_1. Dropping non-query related variables in the ILP (in this case p_2 that is associated with U_2) will also remove the constraint between p_1 and p_3, and that will lead to an incorrect order. The order will be correct, however, if we append extra constraints to the ILP problem (in this case $p_1 < p_3$) to make up for the removal of the non-query related variables. To do so, we take the original set of ILP constraints and compute transitively the relationship between all pairs of query related reorder units. First, we define an auxiliary Boolean variable x_{ij} for $i < j$, where $i, j \in [1, n]$, to indicate that $p_i < p_j$. Then, we rewrite each type of constraints in the original ILP into Boolean clauses using the auxiliary Boolean variables as follows:

- $p_i < p_j \Rightarrow x_{ij} = true$
- $(p_k < p_i) \mid (p_k > p_j) \Rightarrow \begin{cases} (x_{kj} \rightarrow x_{ki}) = true, \text{if } k < i \\ (x_{ik} \rightarrow x_{jk}) = true, \text{if } k > j \end{cases}$
- $p_k > p_j \Leftrightarrow x_{jk} = true$
- $p_l > p_i \Leftrightarrow x_{il} = true$

After that, we combine all rewritten constraints as a conjunction E. Clauses in E can include either a single literal, such as x_{ij}, or two literals, as in $x_{kl} \rightarrow x_{mn}$:

$$E = x_{ij} \wedge ... \wedge (x_{kl} \rightarrow x_{mn}) \wedge ...$$

Note that any ordering that satisfies all the constraints from the original ILP will set the values of the corresponding auxiliary Boolean variables such that E evaluates to true.

We now use the existing clauses in E to infer new clauses by applying the following inference rules:

- $(x_{ij} \rightarrow x_{kl}) \wedge (x_{kl} \rightarrow x_{uv}) \Rightarrow x_{ij} \rightarrow x_{uv}$
- $x_{ij} \wedge x_{jk} \Rightarrow x_{ik}$
- $x_{ij} \wedge (x_{ij} \rightarrow x_{kl}) \Rightarrow x_{kl}$
- $(x_{ij} \wedge x_{jk}) \Rightarrow x_{ik}$
- $((x_{ij} \wedge x_{kl}) \rightarrow x_{uv}) \wedge (x_{uv} \rightarrow x_{mn}) \Rightarrow (x_{ij} \wedge x_{kl}) \rightarrow x_{mn}$
- $((x_{ij} \wedge x_{kl}) \rightarrow x_{uv}) \wedge (x_{mn} \rightarrow x_{ij}) \Rightarrow (x_{mn} \wedge x_{kl}) \rightarrow x_{uv}$

Applying each inference rule generates a new clause, and the process continues until no new clauses can be generated. All clauses are then collected into a conjunction E' with the form:

$$E' = x_{ij} \wedge ... \wedge (x_{kl} \rightarrow x_{mn}) \wedge ... \wedge ((x_{uv} \wedge x_{wx}) \rightarrow x_{yz}) \wedge ...$$

which encodes all the dependencies across each pair of reorder units.

After this process, we convert all clauses in E' back into our ILP constraints. As we go through each clause in E', we only select those clauses with literals about query related reorder units, i.e., $\{x_{ij} : U_i$ and U_j contain queries$\}$, and convert them back into ILP constraints with the following rules:

- $x_{ij} = true \Rightarrow p_i < p_j$
- $(x_{ij} \rightarrow x_{kl}) \Rightarrow (p_i < p_j) \vee (p_k > p_l)$
- $((x_{ij} \wedge x_{kl}) \rightarrow x_{uv}) \Rightarrow (p_i > p_j) \vee (p_k > p_l) \vee (p_u < p_v)$

The ILP constraints will now only involve query related units, and solving these constraints will give us the optimal ordering.

Given a solution to the optimized ILP, there always exists an ordering of all reorder units such that all constraints are satisfied.

4.4.5 Restructuring transaction code

After QURO receives the ordering of queries from the ILP solver, it restructures the input code accordingly. If we rely on the ILP solver to find the order of all reorder units (as discussed in Sec. 4.4.2), then generating the final code would be easy. However, if we apply the optimization discussed in Sec. 4.4.4 to only solve for query related reorder units, we need to determine the ordering of all non-query related reorder units. We discuss the restructuring process in this section.

The basic idea of restructuring code is to iterate through each query according to its new order as given by the solver, try to place other reorder units that have data dependencies on the query being processed, and roll back upon violating any order constraint. As listed in Algorithm 1, we start with an empty list U_list, which is used to store the list of reordered units. We insert a unit U from the set U_s of all reordered units into the list when all other units producing values that U uses are already in the list. To do so, we define two functions: `Defs`(U_i) and `Uses`(U_i). `Defs` returns the set of reorder units that defines variables used by unit U_i, and `Uses` returns the set of reorder units that uses values defined by U_i. The values to be returned are computed during preprocessing as discussed in Sec. 4.3.2. For each query Q_i, we first insert all units in `Defs`(Q_i) into U_list (line 2 to line 8), followed by Q_i itself (line 9), and `Uses`(Q_i) (line 10 to line 16). For every reorder unit U that is inserted into U_list, we check if U violates any data dependency constraints using the function `CheckValid`. Checking is done by scanning the clauses in E' as discussed in Sec. 4.4.4 to see if the current ordering of units would make E' evaluate to false. If so, the current order violates some data dependency constraint encoded in the ILP problem, and the algorithm attempts to resolve the WAR or WAW violation using variable renaming as described in Sec. 4.4.3. If the variable cannot be renamed, then the algorithm backtracks to reprocess all reorder units starting from the first reorder unit that falsifies E'. For each query Q_i, we keep a reject list ($Rej[Q_i]$) to record all reorderings that have been attempted but failed and led to a rollback. The process continues until a satisfying reordering is found, and Sec. 4.4.4 showed that there always exists a valid order.

Algorithm 3 Algorithm For Restructuring Transaction Code

1: **for** $Q_i \in Q\_list$ **do**
2: **for** $U_j \in U_s$ **and** $U_j \notin U\_list$ **and** $U_j \notin Rej[Q_i]$ **and** `Defs(`U_j`)`$\in U\_list$ **and** $Q_i \in$ `Uses(`U_j`)` **do**
3: **if** `CheckValid(`U_j`)` **then**
4: U_list.insert (U_j);
5: **else**
6: break;
7: U_list.insert (Q_i);
8: **for** $U_k \in U_s$ **and** $U_k \notin U\_list$ **and** $U_k \notin Rej[Q_i]$ **and** `Defs(`U_k`)`$\in U\_list$ **and** $Q_i \in$ `Defs(`U_k`)` **do**
9: **if** `CheckValid(`U_k`)` **then**
10: U_list.insert (U_k);
11: **else**
12: break;
13: **function** CheckValid(U_i)
14: ... // check all clauses in E' (details not shown)
15: **if** E' evaluates to true **then**
16: return 1;
17: **else**
18: **if** Variable **v** in U_i can be renamed **then**
19: Rename **v** in U_i and `Uses(`U_i`)`;
20: **else**
21: $temp$ = clear U_list to the failing point U_f;
22: reinsert $temp$ into U_list;
23: **for** query units $Q_f \in temp$ **do**
24: $Rej[Q_f]$.insert(U_f);
25: **end function**

4.5 Profiling

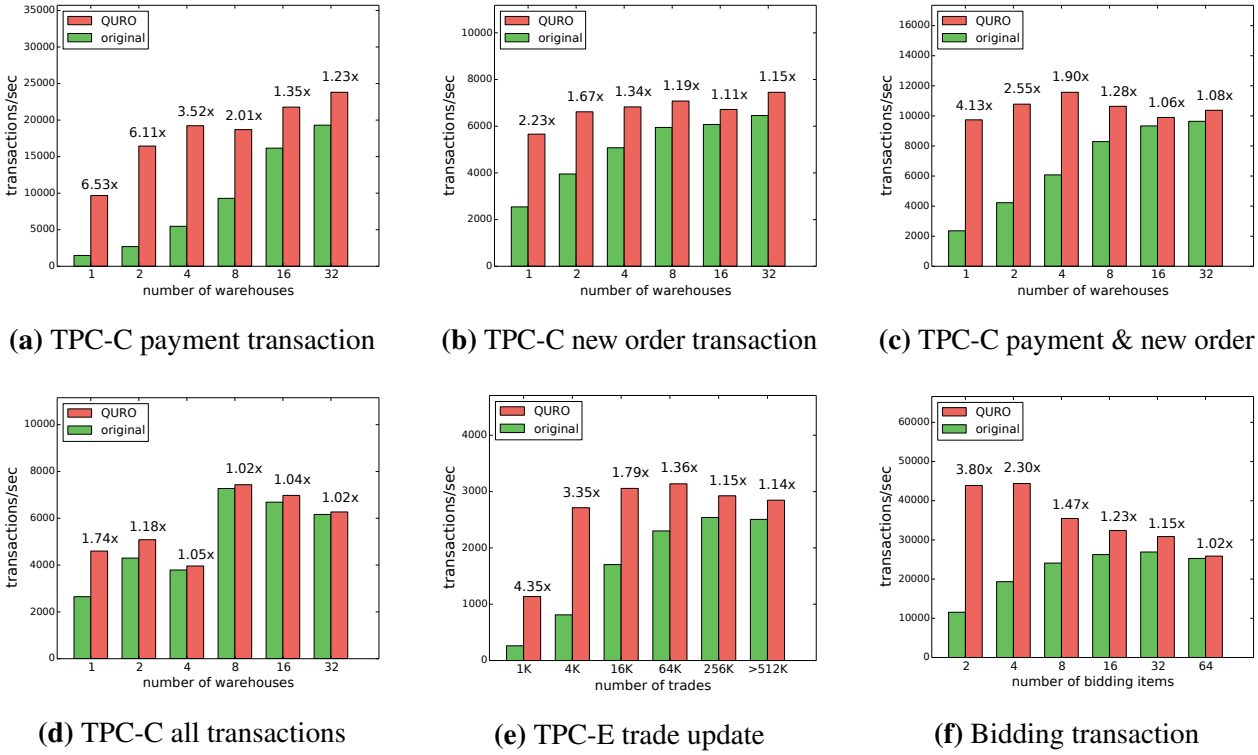
As mentioned in Sec. 4.2, QURO profiles the transaction code by running an instrumented version of the application to estimate the amount of lock contention for each query. There has been prior work that studies how to estimate locking contention. Johnson et al. [116] use Sun's profiling tools to calculate time breakdown of database transactions, analyzing the time spent on useful work and lock waiting to identify contention level in the lock manager. Syncchar [136] runs a representative sample workload to calculate the conflict density and infer lock contention. QURO can use such techniques, but chose a simpler method which examines the running time of each query and computes its standard deviation. In our current prototype, most of the transaction time is spent on lock waiting. If the query accesses contentious data, then the lock waiting time will vary greatly from one execution to another. Hence the larger the deviation, the greater the possibility of data conflict.

To collect query running time, QURO adds instrumentation code before and after each query, and computes the standard deviation after profiling is completed. In the current prototype, we assume that the profiler runs with the same machine settings as the actual deployment of the application.

4.6 Evaluation

We have implemented a prototype of QURO using Clang [13] to process transaction code written in C/C++, and gurobi [35] as the external ILP solver. In this section we report our experiment results under different settings where we compare the performance of the original implementation and the one generated by QURO.

We first study the performance of transaction code generated by QURO by isolating the effects of disk operations (e.g., fetching and writing committed data back to the disk), which can dominate the amount of time spent in processing each transaction. To do so, we disabled flushing data to the disk at commit time. The machine we use has memory large enough to hold the entire data set of any application used in the evaluation. All of the following experiments were performed on MySQL 5.5 server hosted on a machine with 128 2.8GHz processors and 1056GB memory.



(a) TPC-C payment transaction **(b)** TPC-C new order transaction **(c)** TPC-C payment & new order

(d) TPC-C all transactions **(e)** TPC-E trade update **(f)** Bidding transaction

Figure 4.3: Performance comparison: varying data contention

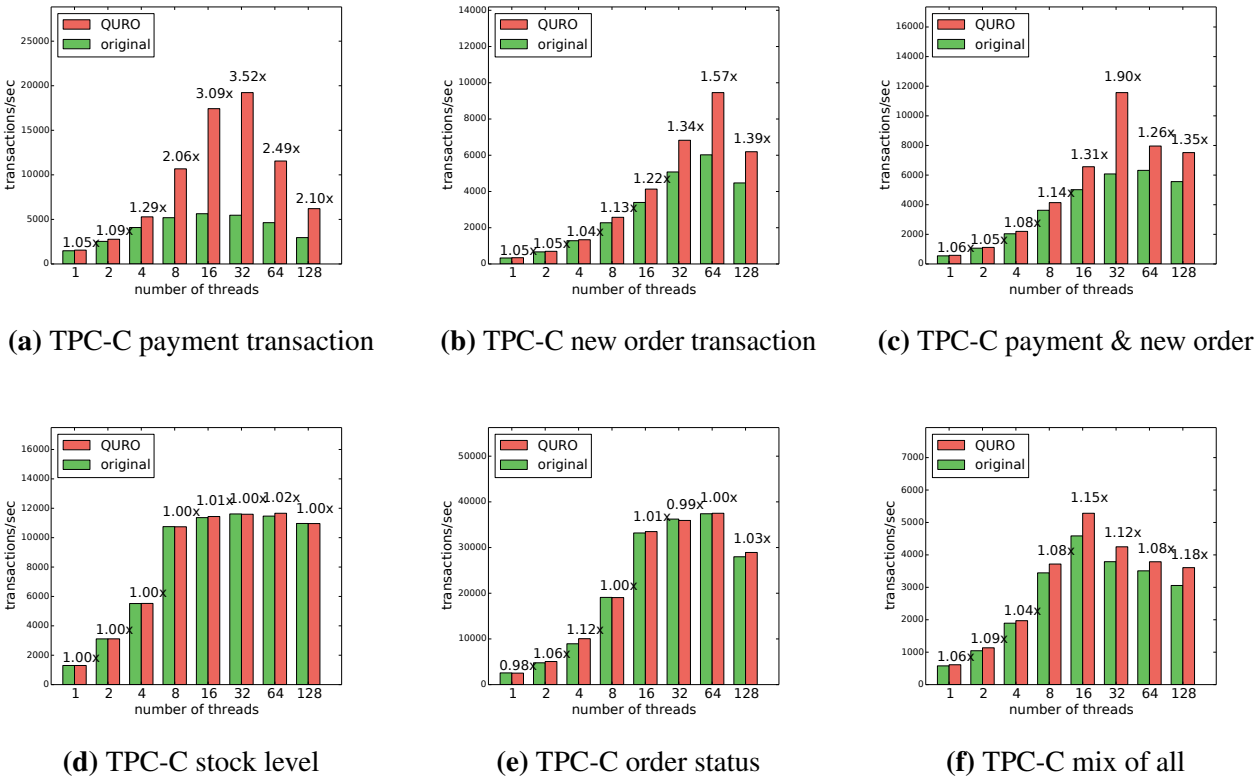
4.6.1 Benchmarks

We used the following OLTP applications for our experiments:

- The TPC-C benchmark. We used an open source implementation [19] of the benchmark as input source code, and performed experiments by running each type of transactions individually and different mixes of transactions.

- The trade related transactions from the TPC-E benchmark. The TPC-E benchmark models a financial brokerage house using three components: customers, brokerage house, and stock exchange. We used an open source implementation [20] as the input. The transactions we evaluated includes trade update, order, result and status transactions.

- Transaction from the bidding benchmark. We use an open source implementation of this benchmark [9]. This benchmark simulates the activity of users placing bids on items. There is only



(a) TPC-C payment transaction **(b)** TPC-C new order transaction **(c)** TPC-C payment & new order

(d) TPC-C stock level **(e)** TPC-C order status **(f)** TPC-C mix of all

Figure 4.4: Performance comparison: varying number of threads for TPC-C benchmark

one type of transaction in this benchmark: it reads the current bid, updates the user and bidding item information accordingly, and inserts a record into the bidding history table.

We ran the applications for 20 minutes in the profiling phase to collect the running time of queries. In each experiment, we omitted the first 5 minutes to allow threads to start and fill up the buffer pool, and the measured the throughput for 15 minutes. We ran each application for 3 times and report the average throughput. We also omitted the thinking time on the customer side, and we assume that there are enough users issuing transactions to keep the system saturated.

4.6.2 Varying data contention

In the first experiment, we compared the performance of the original code and the reordered code generated by QURO by varying contention rates while fixing the number of concurrently running database threads to 32. Varying data contention is done by changing the data set size a transaction accesses. For the TPC-C benchmark, we change the data size by adjusting the number of warehouses, from 1 to 32. With 1 warehouse, every transaction either reads or writes the same warehouse tuple. With 32 warehouses, concurrent transactions are likely to access different warehouses as they have little data contention. For the TPC-E benchmark, we adjust the number of trades each transaction accesses. As the trade update transaction is designed to emulate the process of making minor corrections to a set of trades, changing the number of trades changes the amount of data the transaction accesses. We varied the number of trades from 1K to the size of entire trade table, 576K. When the number of trades being updated is small, multiple concurrent transactions will likely modify the same trade tuple. In contrast, when transactions randomly access any trade tuple in the trade table, they will likely modify different trades and have little data contention. We did not evaluate the other transactions since no contentious data is accessed in these transactions, so varying data size won't significantly change the locking situation, and performance will not be much affected by locking. For the bidding benchmark, we adjusted the number of bidding items. The bidder giving a higher bidding price will change the current price on that bid item. We set the percentage of bidder giving higher bidding price to be 75%, which means that 75% of the transactions will write the item tuple.

Figure 4.3a shows the results of TPC-C running only the payment transaction (an excerpt is shown in Listing 4.1). Reordered implementation generated by QURO achieves up to 6.53× speedup as compared to the original implementation. Figure 4.3b shows the results of TPC-C benchmark running only new order transactions. In this transaction, the flexibility of reordering is restricted by the many data dependencies among program variables. Despite this limitation, QURO still achieves up to 2.23× speedup as a result of reordering. Figure 4.3c shows the results of the TPC-C benchmark comprising 50% new order and 50% payment transactions. Under high data contention,

the speedup of reordering is 4.13×. Figure 4.3d shows the results of TPC-C with standard mix of five types of transactions according to the specification. Increasing the types of transactions makes more data contentious, as some tables are only read by one type of transaction, hence there are no data contentions on those tables when only that type of transactions are executed. But with transaction mixes, there might be other transaction types that would write to the same table, thus causing contentions. However, with a mix of five types of transactions, reordering still increases the overall throughput by up to 1.74×.

For TPC-E, Figure 4.3e shows the results of trade update transaction. This benchmark has a while loop that on average runs for 20 iterations. There are multiple read queries within each iteration, but only one update query. QURO discovered the optimal reordering by breaking the loop and putting all the write operations from different iterations towards the end of the transaction, resulting in a speedup of up to 4.35× as compared to the original implementation. Even when there is little data contention, reordering still outperforms the original implementation by 1.14×.

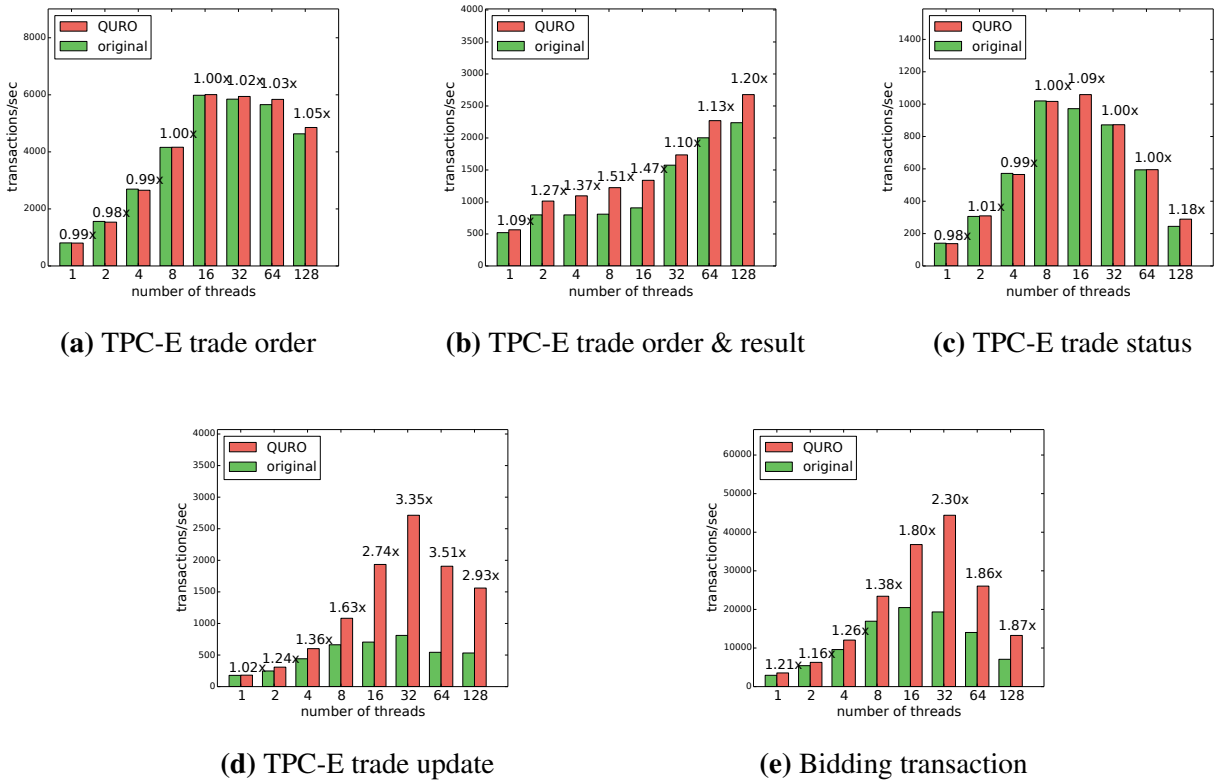
Finally, Figure 4.3f shows the results of the bidding benchmark. The bidding transaction is short and contains only five queries. In the reordered implementation, the bidding item is read as late as possible, followed by the item update being the last operation in the transaction. This resulted in a speedup of up to 3.80×.

The results show that as the data size decreases, the contention rate increases, which in return increases the chance of improving performance by reordering.

4.6.3 Varying number of database threads

In the next set of experiments, we ran the benchmarks on the same data, but varied the number of database threads from 1 to 128. With the same database size, running more threads increases the amount of contention. We would like to study how the Quro-generated implementations behave as the number of threads increases.

Figure 4.4a-d shows the results of running transactions from TPC-C. In this experiment we fixed the number of warehouse to be 4. For the payment transaction, QURO speeds up the application by up to 3.52×. For the new order transaction, the amount of speedup due to reordering increases as



(a) TPC-E trade order **(b)** TPC-E trade order & result **(c)** TPC-E trade status

(d) TPC-E trade update **(e)** Bidding transaction

Figure 4.5: Performance comparison: varying number of threads

thread number increases, reaching the maximum of 1.57× when using 64 threads. Stock level and order status transactions are read-only transactions. In these transactions, no query needs to wait on lock when running standalone, and QURO-generated implementation has the same performance as the original implementation. We did not analyze the delivery transaction in TPC-C, since this transaction performs both read and write on a very small set of data. When it is running standalone, most transactions will abort.

Figure 4.5d shows the results of the TPC-E trade update transaction, where we fixed the number of trades to be 4K. The throughput of original implementation falls greatly as the number of threads exceeds 16, while the reordered implementation only decreases slightly, and still speeds up the application by up to 3.35×.

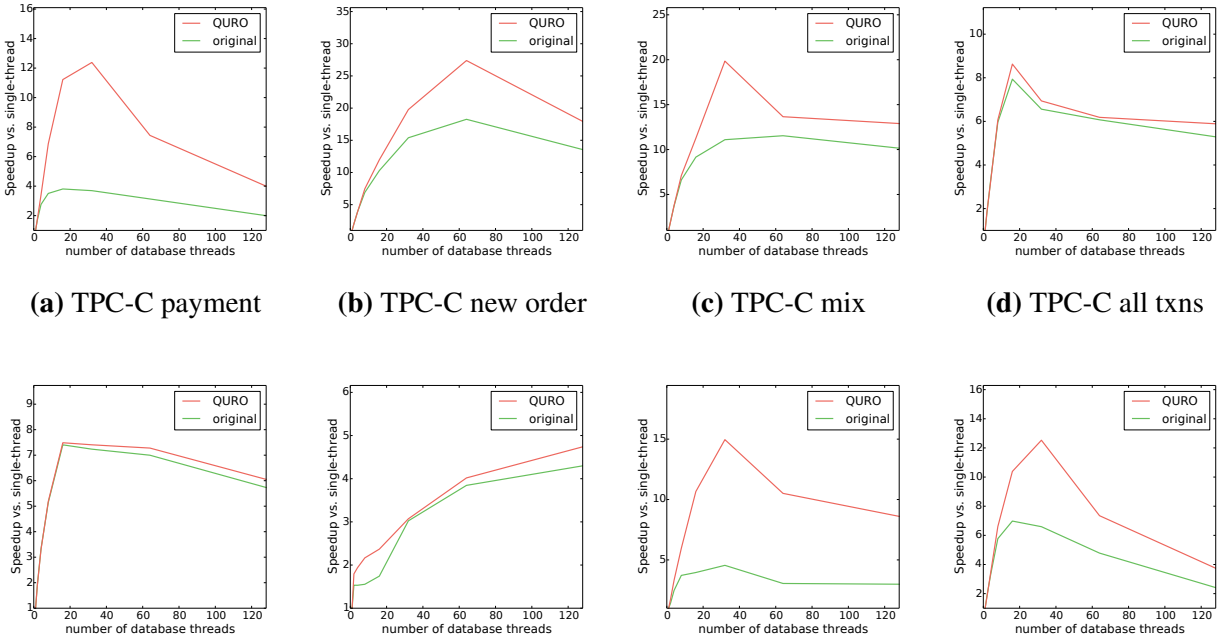
Unlike TPC-C new order and payment transactions which have significantly contentious queries, queries in trade order and trade result transaction access data with small contention. For trade order transaction, 19 out of 21 queries are read queries, and the remaining 3 write queries are insert operations. These write queries are executed after all the read queries in the transaction in the original implementation. QURO only changes the order of the last three insert operations, which does not greatly affect the overall performance. Figure 4.5a shows that QURO-generated implementation has nearly the same performance as the original implementation. In the best case QURO improves the application throughput by 1.05×, and in worst case decreases throughput by only 2%. Since the input of trade result transaction is dependent on the trade order transaction, we only evaluated a mix of these two types of transactions. Figure 4.5b shows that reordering increases the throughput by up to 1.57×. For the trade status transaction which is a read-only transaction, QURO-generated implementation has the same performance as the original one as shown in Figure 4.5c.

Finally, Figure 4.5e shows the results of the bidding transaction. We fixed the number of bidding items to be 4. As shown in the figure, reordering increases application throughput by up to 2.30×.

We also compared how each benchmark scales as the number of threads increases by evaluating self-relative speedup. The baseline for both implementations is the throughput of single-thread execution with the original implementation, and Figure 4.6 shows how throughput changes as the number of threads increases. As shown in the figure, the reordered implementation has larger self-relative speedup than the original implementation as the number of threads exceeds 8. When running on 32 threads on the payment transaction, QURO-generated implementation has 12.4× self-relative speedup while 3.8× for the original. By reordering queries, QURO allows applications to scale up to a larger number of concurrent database connections as compared to the original implementation.

4.6.4 Analyzing performance gain

We did another set of experiments to gain insights on the sources of performance gain due to reordering.



(a) TPC-C payment **(b)** TPC-C new order **(c)** TPC-C mix **(d)** TPC-C all txns

(e) TPC-E trade order **(f)** TPC-E trade mix **(g)** TPC-E tradeupdate **(h)** Bidding

Figure 4.6: Self-relative speedup comparison

Query execution time We profiled the amount of time spent in executing each query. We show the results for the TPC-C payment transaction in Table 4.2. The table lists the aggregate time breakdown of 10K transactions, with 32 database threads running transactions on a single warehouse. In this case reordering sped up the benchmark by 6.53×, as shown in Figure 4.3a.

The single-thread results in the third column indicates the running time of queries without locking overhead as each transaction is executed serially. By comparing the values to the single-thread query time on each row, we can infer amount of time spent in waiting for locks of that query in both implementations.

For the original implementation, the results show that most of the execution time was spent on query 2. As shown in Listing 4.1, this query performs an update on a tuple in the warehouse table, and every transaction updates the same tuple. By reordering, the running time of this query is

	original	reordered	single-thread	ratio
query 1	0.97	1.09	0.72	1.12
query 2	210.27	6.06	0.73	0.03
query 3	0.97	1.12	0.68	1.15
query 4	0.80	17.18	0.62	21.48
query 5	1.38	1.59	0.70	1.15
query 6	1.86	1.31	0.98	0.70
query 7	1.09	1.52	0.81	1.39
query 8	1.16	1.60	0.17	1.38
query 9	0.79	0.96	0.70	1.22
total	219.29	32.43	6.11	0.15

Table 4.2: Query running time of the payment transaction, where ratio=reordered/original. The reordered implementation reduces latency by 85%.

	original	reordered	single-thread	ratio
query 1	0.73	0.76	1.91	1.04
query 2	89.77	16.16	0.93	0.18
query 3	0.92	1.01	1.02	1.09
query 4	0.81	0.88	0.76	1.09
query 5	0.63	0.69	0.71	1.08
query 6	0.74	0.97	0.78	1.30
query 7	0.66	0.76	0.62	1.16
query 8	0.69	0.80	0.78	1.15
query 9	0.79	1.06	0.69	1.34
query 10	0.67	0.82	0.62	1.23
total	114.99	46.66	27.01	0.41

Table 4.3: Query running time of the new order transaction. The reordered implementation reduces latency by 59%.

significantly shortened: the query time in the original implementation is reduced by 97%. However, the execution time for all other queries increases after reordering. This is due to the increased chances of other queries getting blocked during lock acquisition. In the original implementation, the query accessing the most contentious data effectively serializes all transactions as each thread needs to acquire the most contentious lock in the beginning. As a result, the execution time of all subsequent queries is nearly the same as the time running on a single thread (i.e., without locking overhead). But reordering makes these queries run concurrently as they need to compete for locks, and this increases the running time.

We also profiled the new order transaction. In this transaction, the opportunities for reordering is limited by the data dependencies among queries. In particular, query 2 reads a tuple from the district table and reserves it for update later in the transaction. This query is the most contentious one. However, query 2 cannot be reordered to the end of transaction since there are many other queries that depend on the result of this query. This limits the amount of query time reduction, as shown in Table 4.3.

The above analysis indicates a trade-off with reordering. Reordering decreases the time spent on lock waiting on conflicting queries, but increases the time spent on less-conflicting queries. In most cases, the decrease in lock waiting time for contentious data usually outweighs the increase in lock waiting time for queries that access non-contentious data, and reordering reduces query latency despite some queries now takes slightly longer time to execute.

#of trades	original	reordered	ratio	speedup
1K	53.89%	39.12%	0.73	4.35x
4K	20.08%	1.78%	0.09	3.35x
16K	2.39%	0.35%	0.15	1.79x
64K	0.55%	0.08%	0.15	1.36x
256K	0.14%	0.02%	0.14	1.15x
>512K	0.00%	0.00%	1.00	1.14x

Table 4.4: Abort rate comparison of the trade update transaction, where ratio=reordered/original. With 4K trades, reordering can reduce the abort rate by up to 91%.

Abort rate For the TPC-E trade update transaction, reordering improves performance by reducing abort rate. On average one trade update transaction randomly samples 20 trades and modifies the trade tuples selected. When the number of trades is small, concurrent transactions are likely to access the same set of trades, but in a different order, which causes deadlock. Reducing the locking time not only makes transaction faster, but also reduces the abort rate as shown in Table 4.4.

Figure 4.7 shows how reordering reduces abort rate. In the trade update transaction, only one query modifies the trade table in every loop, while there are other queries performing read on other tables that are not being modified. Assuming transaction T1 starts first, there is a time window within which if T2 starts, then T1 and T2 will likely deadlock each other. We refer this time range as the deadlock window, as shown in Figure 4.7. After reordering, the deadlock window is shortened, so there is less chance that the transaction would abort, and thus throughput increases.

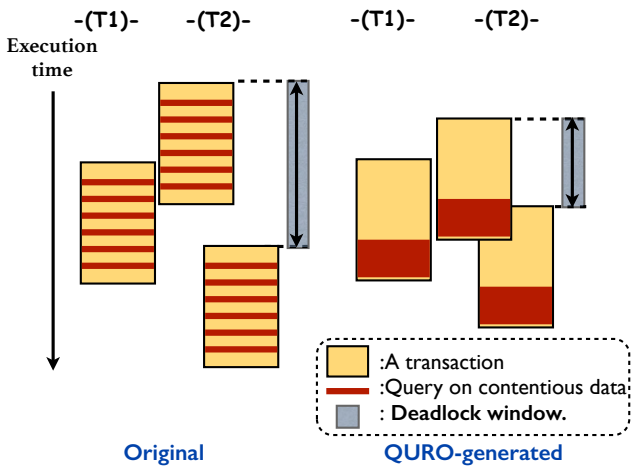


Figure 4.7: Execution of trade update transactions T1 and T2 before and after reordering. Deadlock window is the time range where if T2 starts within the range, it is likely to deadlock with T1.

4.6.5 Worst-case implementations

To further validate our observation that the order of queries affects transaction performance, we manually implement a "worst-case" implementation where the most contentious queries are issued *first* within each transaction. We then compared the performance of those implementations against the original and QURO generated implementations. The result for the TPC-C payment transaction is shown in Figure 4.8. This experiment shows that the order of queries can have great impact on performance. While QURO relies on profiling to estimate the lock contention, if the workload changes, transactions need to be re-profiled and reordered accordingly. We leave dynamic profiling and regenerating transaction code as future work.

4.6.6 Disk-based DBMS

Next we consider the performance of QURO-generated code for disk-based DBMSs. We measured application performance while changing the amount of time spent on disk operations during transaction execution. This is done by varying buffer pool sizes. When the buffer pool is smaller than the database size, dirty data needs to be flushed to disk, making disk operations a significant

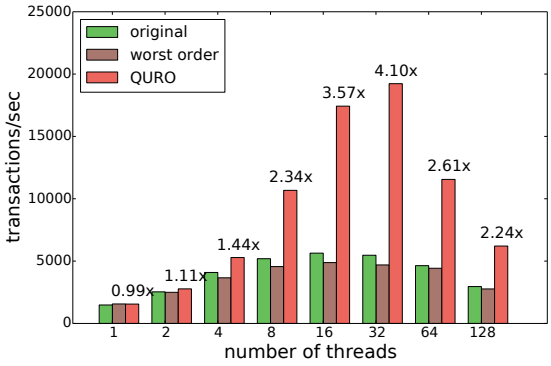


Figure 4.8: TPC-C worst-case

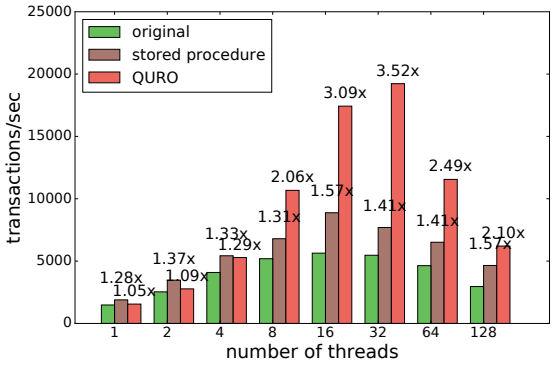
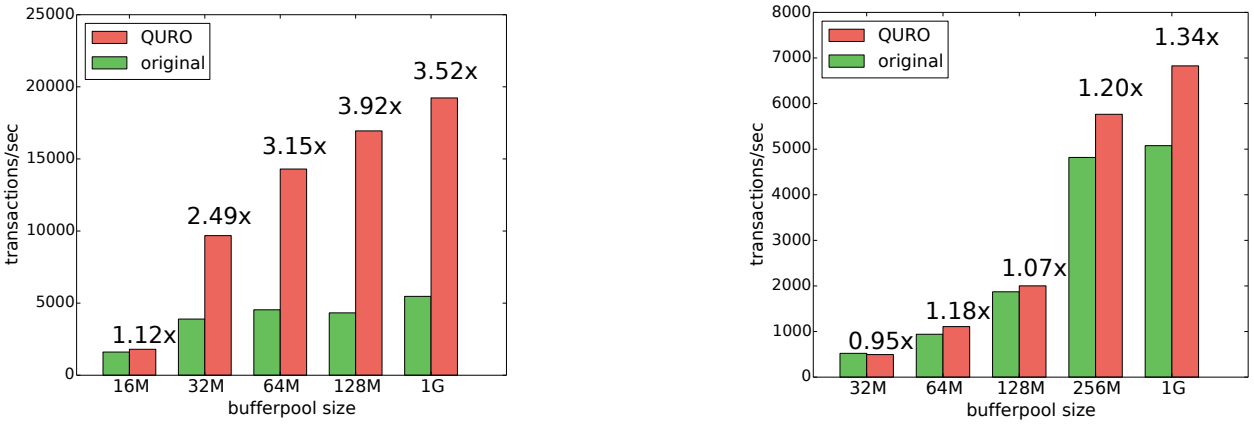


Figure 4.9: Comparison with stored procedure.

portion of transaction execution time. Figure 4.10 shows the performance comparison of the TPC-C transactions. When the buffer pool is small, disk operations dominate the transaction processing time, thus the performance gained by reordering becomes trivial. Increasing buffer pool size decreases the time spent on disk and, as expected, that increases throughput as a result of reordering. This experiment shows that even when the database is not completely in-memory, reordering can still improve application throughput.

4.6.7 Performance comparison with other concurrency control schemes

We also run experiments to test the performance of 2PL reordered implementation of TPC-C transactions against other concurrency control schemes like optimistic concurrency control (OCC) and multi-version concurrency control (MVCC). In this experiment, we use the main memory DBMS provided by Yu et al. [158], which supports different schemes including several variants of 2PL, OCC and MVCC. Since this DBMS doesn't use standard query interface, we manually ordered the transaction according to the order generated by QURO (the same order as in previous experiments). We evaluated the throughput of the TPC-C payment, new order, and a mix of these two transactions with original and reordered implementation under two versions of 2PL: 2PL with deadlock detection (DL_DETECT) and 2PL with non-waiting deadlock prevention (NO_WAIT), as well as the original implementation under OCC and MVCC. We set the number of warehouses to be



(a) TPC-C payment

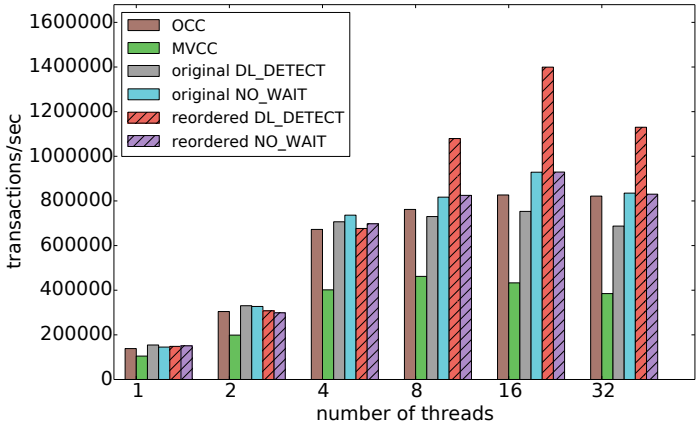
(b) TPC-C neworder

Figure 4.10: Throughput comparison by varying buffer pool size

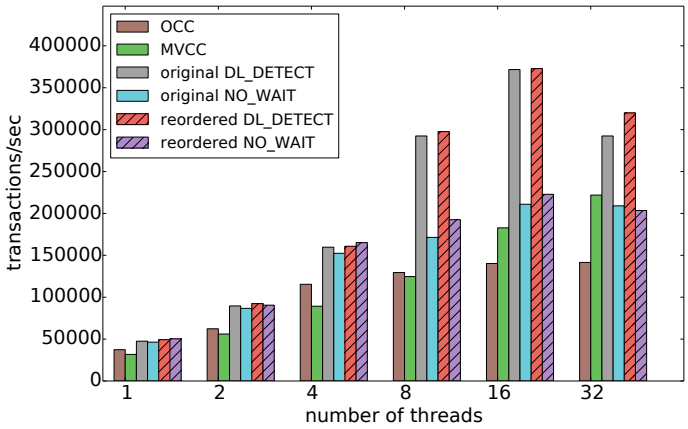
4 and varied the number of threads from 1 to 64, to test how different schemes scale.

Figure 4.11a-c show the results of this experiment. Shaded bars show the throughput of reordered implementation while other bars show the original implementation under different concurrency control schemes. As the number of threads increases, data contention increases, and the performance gain of reordered transaction under 2PL comparing to the original transaction under OCC and MVCC becomes larger. As shown in Figure 4.11c, for a mix of payment and new order transaction with 16 threads, reordered under 2PL with deadlock detection(DL_DETECT) outperforms OCC by 3.04×, and MVCC by 2.74×.

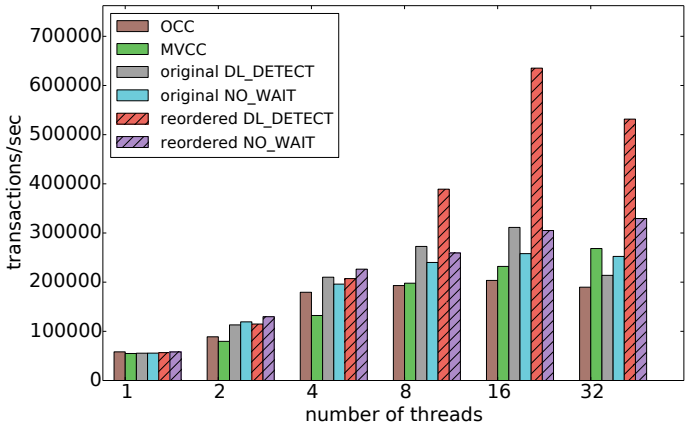
Figure 4.12 explains why reordered transaction under 2PL outperforms original transaction under OCC and MVCC. Assume two transactions T1 and T2 both read and write on a contentious tuple, and T1 starts first. Under each concurrency control scheme, there are time ranges within which if T2 starts, then it will end up being aborted. We refer this time range as abort window. Since OCC only writes to database at validation phase, any read on the contentious tuple before the validation phase will cause T2 to abort. Figure 4.12 shows that the original implementation under 2PL and OCC has much longer abort window than MVCC and reordered implementation under 2PL. Since MVCC requires complicated version management and the concurrency control overhead



(a) TPC-C payment transaction



(b) TPC-C new order transaction



(c) TPC-C payment & new order

Figure 4.11: Performance comparison of TPCC transactions among original implementation under OCC, MVCC, and variants of 2PL.

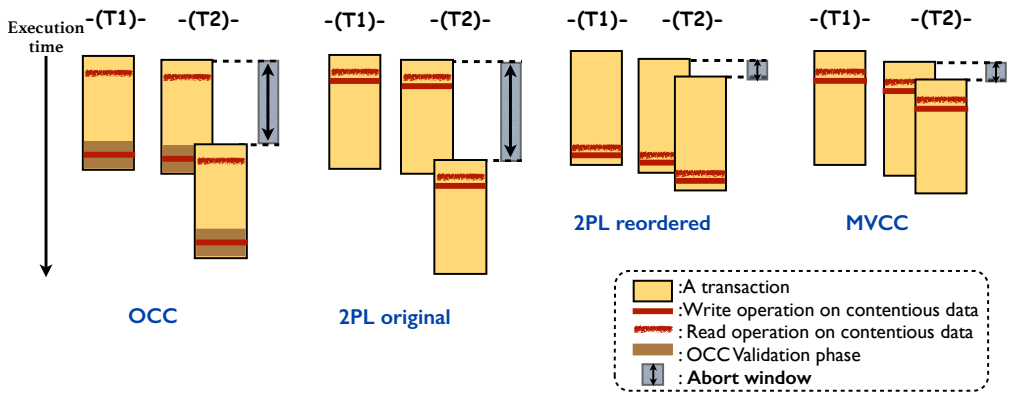


Figure 4.12: Execution of transaction T1 and T2 under different concurrency control schemes.

is larger than 2PL, reordered 2PL has higher throughput than MVCC.

4.6.8 Performance comparison with stored procedures

In the previous experiments, we ran our database server and client program on the same machine to minimize the effect of network round trips as a result of issuing queries. In this experiment we added another open source implementation that uses stored procedures [19]. Using stored procedures, the client issues a single query to invoke the stored procedure that implements the entire transaction. As a result, the amount of network communication between the client and the database is minimized.

Figure 4.9 shows the results of TPC-C payment transaction in this experiment. When there is little data contention, the stored procedure implementation outperforms the others due to shorter round trip time. However, as the amount of data contention increases, the time spent on locking far exceeds the time spent on the network communication, and the reordered implementation has higher performance improvement (3.52×) over the original, as compared to the improvement of the stored procedure implementation (1.41×).

4.6.9 ILP optimization

In the final experiment, we quantified the optimization presented in Sec. 4.4.4 in reducing the amount of ILP solving time. For the experiment, we chose two transactions: new order transaction with 40 statements and 9 queries, and trade order transaction with 189 statements and 21 queries. After preprocessing, the two transactions were split into 27 and 121 reordering units respectively. We used QURO to formulate each transaction into an ILP problem, using both the simple formulation discussed in Sec. 4.4.2 and the optimized formulation as discussed in Sec. 4.4.4. We then used two popular open source ILP solvers, lpsolve [42] and gurobi [35], to solve the generated programs with a timeout of 2 hours. The algorithm for restructuring the transaction code is described in Sec. 4.4.5, which is only needed when optimization is used, and runs for 1 second in both cases.

	Transaction 1		Transaction 2	
# statements	40		189	
# queries	7		25	
# variables	27		121	
# constraints	266		3595	
# constraints post-opt	6		64	
	lpsolve	gurobi	lpsolve	gurobi
Original solving time	>2hrs	1s	>2hrs	>2hrs
Optimized solving time	1s	1s	>2hrs	1s

Table 4.5: ILP experiment results. The number of variables equals to the number of reordering units. Together with the number of constraints, these two numbers indicate the ILP problem size.

The results are shown in Table 4.5. Under the original problem formulation where each reorder unit is represented by a variable in the ILP, both solvers did not finish before timeout for transaction 2. In contrast, the optimized formulation reduces the number of variables by 79% and the number of constraints by 98%. The problem also solves much faster, and hence allowing QURO to process larger transaction code inputs.

4.7 Summary and Future Work

In this chapter, we presented QURO, a new tool for compiling transaction code. QURO improves performance of OLTP applications by leveraging information about query contention to automatically reorder transaction code. QURO formulates the reordering problem as an ILP problem, and uses different optimization techniques to effectively reduce the solving time required. Our experiments show that QURO can find orderings that can reduce latency up to 85% along with an up to 6.53× improvement in throughput as compared to open source implementations.

The limitation of QURO lies in the flexibility of reordering: it can only achieve good performance gain when the hotspot access does not have data dependency with later queries and can be moved to the bottom of transaction. When such hotspot access cannot be moved, the locking period can still be reduced by releasing the lock early before commit time, but at the cost of potential cascading abort if multiple hotspots exist. We are designing a new protocol that enables early lock release which can compensate QURO's limitation.

Chapter 5

POWERSTATION: UNDERSTANDING AND FIXING PERFORMANCE ANTI-PATTERNS

Database-backed applications often use ORM frameworks to manage persistent data. While ORM hides the relational query details from developers for the ease of development, it is also vulnerable to performance issues because developers may write code that results in efficient queries. In this work, we summarized the common code patterns from the popular open-source web applications that leads to slow performance, which we call "anti-patterns". Then we describe PowerStation, a RubyMine IDE plugin to automatically detect and fix some of these anti-patterns. Our evaluation using 12 real-world applications shows that PowerStation can automatically detects 1221 performance issues across them.

5.1 Understanding Performance Anti-Patterns

We performed a comprehensive study on open source web applications, where the application corpus and some of the study results were mentioned in Sec. 1.2. In this section we describe it in more detail. This study includes two parts. One one hand, we propose a few anti-patterns that we summarized by looking into the query log of the application, and then use static code analysis on a large body of open source applications to check whether these anti-patterns indeed widely exist. One the other hand, we look into the issue reports from the applications' bug tracking systems and summarizes the common code anti-patterns that developers complain and file issues. In this thesis we will mainly introduce the first part, and the detail of the issue-report study can be found in [155].

5.1.1 Static analysis methodology

We build an ORM-aware static analyzer for Rails applications in order to find and confirm the proposed anti-patterns. The analyzer produces an *Action Flow Graph* (AFG) that contains the data-flow and control-flow information for each action, along with ORM-specific information inside and across different actions. For instance, the graph labels query function nodes that we identify based on Rails semantics, such as the `where` function shown in Figure 1.2. To support inter-procedural analysis, we start building AFGs from the top-level function that runs an action and iteratively inline all function calls.[1] In addition to regular function calls, we also inline filters and validations [32], which are functions that are automatically executed before an action and before any function that modifies database, respectively.

The AFG contains *next action* edges between pairs of actions (c_1, c_2) if c_2 can be invoked from c_1 as a result of a user interaction. To identify such interactions, we identify actions that render URLs (e.g., `link_to` in Figure 1.3) or forms in their output webpages, and determine the subsequent actions that may be triggered as a result of an user interaction (e.g., clicking an URL or submitting a form) based on the application routing rules, such as those listed in Figure 1.3.

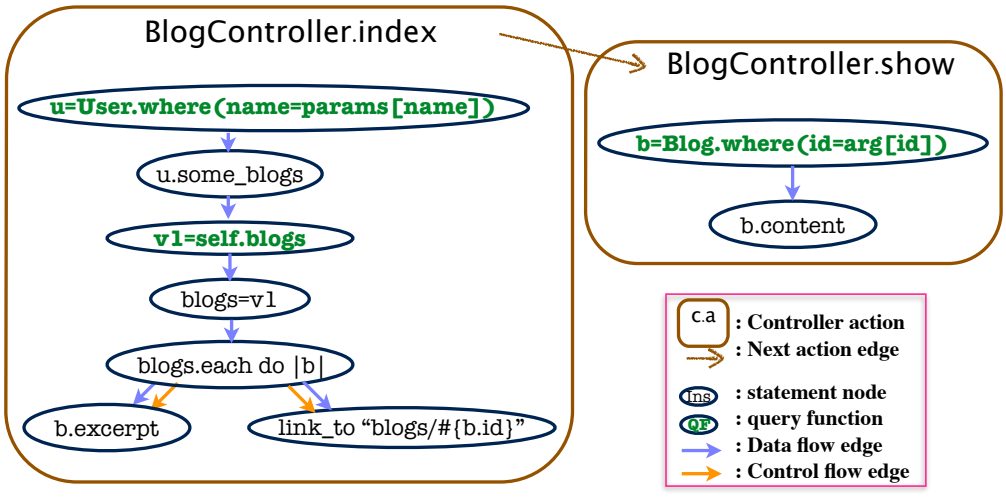


Figure 5.1: Action Flow Graph (AFG) example

[1] We assume all recursive calls terminate with call depth 1.

App	Loc	App	Loc	App	Loc	App	Loc	App	Loc	App	Loc
forum		**social networking**		**collaboration**		**task management**		**resource sharing**		**e-Commerce**	
forem	5957	kandan	1694	redmine	27589	kanban	2027	boxroom	2614	piggybak	2325
lobsters	7127	onebody	32467	rucksack	8388	fulcrum	4663	brevidy	13672	shoppe	5904
linuxfr	11231	commEng	34486	railscollab	12743	tracks	23129	wallgig	11189	amahi	8412
sugar	11738	diaspora	47474	jobsworth	15419	calagator	1328	enki	5275	sharetribe	67169
				gitlab	145351			publify	16269		

Table 5.1: Application categories and lines of source code (all applications are hosted on github). All applications listed are studied with static analysis. Among them, gitlab, kandan, lobsters, publify, redmine, sugar, tracks are profiled as well.

We apply this analyzer on more applications than the 12 applications listed in Table 1.1, with a sum of 27 applications, as shown in Table 5.1. The wider range of applications supports the validity of our findings more strongly.

Next we will introduce the anti-patterns that leads to slow queries. For each anti-pattern, we will introduce how the analyzer detects it, its distribution in the applications we studied, and the potential fix. We group the optimization by two large categories: single-action query optimization and multi-action caching.

5.1.2 Anti-patterns in single action

Many ORM frameworks allow users to construct queries by chaining multiple function calls (e.g., `where`, `join`), with each chain translated into a SQL query. We look into the query log generated by running the applications as described in Sec. 1.2 and manually check the query log and trace the application code to understand how these queries are generated. We find that current query translation scheme often generates inefficient queries. However, they can be optimized by understanding the query generation process and how the results are subsequently used. In the following we introduce common types of inefficient queries that we summarize through a manual exam of the query log and source code. Then we propose ways to identify and optimize them, and manually implement a subset of the optimizations to demonstrate potential performance gain.

5.1.2.1 *Caching common subexpressions*

By studying the query log, we find that many queries share common subexpressions that cause repetitive computation. An example is shown in Listing 5.1. First, `subPj` is created on `subprojSelect` from `projects` by issuing a query (`Q1` in Figure 5.2) to retrieve the first level nested `projects` of the given `projects`. Then, `projSelect` creates `descPj` also from `projects`, issuing another query (`Q2` in Figure 5.2) to get all descendants of the given `projects`. These two queries share the same selection predicate and the results are both ordered by project `id`, i.e., they share the same common subexpression.

Listing 5.1: Example queries that partially share predicate, abridged from redmine [54].

```
subPj = projects.children.visible.order('id ASC')
descPj = projects.where('lft > ? AND rgt < ?').visible.order('id ASC'
    )
```

	Original queries	Query time
Q1:	SELECT projects.* FROM projects WHERE projects.parent_id=? AND (projects.status <> 9) ORDER BY projects.id ASC	0.70 sec
Q2:	SELECT projects.* FROM projects WHERE (projects.lft > ? AND projects.rgt < ?) AND (projects.status <> 9) ORDER BY projects.id ASC	2.91 sec
	Simulating intermediate-query-result cache	
Q3:	CREATE VIEW pj AS SELECT * FROM projects WHERE (projects.status <> 9) ORDER BY projects.id ASC	0.04 sec
Q4:	SELECT * FROM pj WHERE pj.parent_id = ?	0.69 sec
Q5:	SELECT * FROM pj WHERE (pj.lft > ? AND pj.rgt < ?)	0.47 sec

Figure 5.2: Performance gain by caching of query results.

To systematically measure how prevalent such common subexpressions are, we generate the AFG of each action and compare the query call chains using static analysis: if the same query function is used in two different query function chains, the corresponding queries will share a common subexpression. Using this analysis, we count the number of queries that share subexpressions with a previous query issued within the same action.

Our static analyzer shows that **17% queries share subexpressions with other queries with**

most of them performance critical.

To study the effect of caching the intermediate results in Listing 5.1, we create a view (`Q3` in Figure 5.2) to store the results of the common subexpression (i.e., the ordered projects with certain status) and change the queries to use the view instead (`Q4` and `Q5`). By using the cached results, the total query execution time of `Q1` and `Q2` is significantly reduced from 3.6 to 1.2 seconds (67%).

There has been prior work on identifying shared subexpressions in the context of multi-query optimization by batching and analyzing queries online [103, 111, 105]. This imposes a performance penalty on all queries due to the detection overhead of common subexpression at runtime. Instead, using static analysis offline incurs no runtime overhead and is still able to find many queries that potentially share subexpressions. Static analysis alone may result in false positives: if a query shares subexpressions with queries in different branches, such analysis may propose a strategy to cache all subexpressions but at runtime only one branch is taken and one subexpression will be useful. While this brings extra caching overhead, our manual check on all applications finds that very few such cases arise.

5.1.2.2 Fusing queries

Checking the query logs reveals that the results of queries are often only used to issue subsequent queries. Listing 5.2 shows an example of such queries. `Q1` in the original implementation returns all members from group 1, with the results (`m`) only used to issue a subsequent query `Q2` to retrieve the corresponding issues. Each query incurs a network round trip between the DBMS and application server and application server will serialize query results into objects after the results returned. Combining such queries can reduce the amount of data to be transferred, reducing the time spent on network and serializing data.

Listing 5.2: Original queries (abridge from redmine [54]) listing issues by members from group 1.

```
1 Q1: m = SELECT * FROM members WHERE group_id = 1;
```

```
2 Q2: SELECT * FROM issues WHERE creator_id IN (m.id) AND is_public =
    1;
```

We use static analysis on the AFG to identify queries whose results are only used to issue subsequent queries, with the goal to fuse them such that their results do not need to return to the application. To understand how query results are used, we trace the dataflow from each query node in the AFG until we reach either a query function node, or a node having no outgoing dataflow edge. Such read query sinks can be classified as: (1) query parameters used in subsequent queries; (2) results rendered in the view; (3) values used in branch conditions; or (4) values assigned to global variables. After analyzing the sinks, we count the number of queries that only have sinks belonging to (1), i.e., they are queries that can be fused.

Our static analyzer shows that **33% of queries return results that are only used to issue subsequent queries.**

Fusing queries can lead to significant performance gain. In the previous example, `Q1` and `Q2` can be combined into a single query as shown in Listing 5.3. This reduces execution time from 1.46 and 1.3 seconds for executing `Q1` and `Q2` respectively to 1.02 seconds for executing the fused query (reduction of 62.3%). Moreover, fusing `Q1` and `Q2` avoids returning the result of `Q1` (20K records, 340KB in size in our experiments) to the application server, which brings further performance gain due to less data transfer over network and reduced serialization effort.

Listing 5.3: Combining Q1 and Q2 in Listing 2

```
SELECT * FROM issues INNER JOIN members ON members.group_id = 1 AND
issues.is_public = 1 issues.creator_id = members.id
```

However, such optimization can also lead to a few issues. First, if a query's result is used as a parameter to more than one subsequent query, then query fusion will lead to repeated query execution. Fortunately, using common subexpression optimization can avoid repeated work. Secondly, the performance of combined queries is dependent on the query optimizer, so combined query may not be always faster than the original separated queries.

5.1.2.3 Eliminating redundant data retrieval

We find that many queries issued by the applications retrieve fields that are not used in subsequent computation. By default, query functions provided by ORMs fetch entire rows (i.e., `SELECT *`) from the database, unless programmers use explicit functions to project specific columns from the table (e.g., using `select` in Rails). Unfortunately, such project functions are rarely used as programmers who write model functions to retrieve objects are usually unaware of how their functions will be subsequently used (possibly due to structuring the application using Model-View-Control [140]). As such, automatically identifying and avoiding unnecessary data retrieval can reduce both query execution time and amount of data transferred.

To do so, we use the AFG to identify the fields retrieved by each query along with their subsequent uses. Next, we calculate the amount of unused data. For each fixed-size field like integers, we use the data size stored in the database; for unbounded size field (e.g., `varchar`) we use 2000 bytes.[2]

Our static analyzer shows that **more than 63% of all retrieved data is not used in the application after retrieval.**

After identifying such fields, queries can be rewritten such that only used columns are retrieved using projection as mentioned above. Prior work has studied the unnecessary column retrieval problem [91] but did not propose an effective way to automatically identify them. In particular, it only evaluates the performance impact by analyzing the program and the query log obtained from dynamic profiling. Our method shows that using only static analysis on AFGs can effectively detect both retrieved and used columns, and rewrite queries automatically to avoid redundant data retrieval.

5.1.2.4 Query with unbounded results

We observe that loops are usually the cause of performance inefficiencies in the processing of query results. By analyzing loops in the AFG, we find that 99% of them iterate over arrays or maps; 49%

[2]Earlier studies [24, 14] have concluded that the most popular length of a comment is around 200 words and in average each word is 10 bytes.

process results from queries issued in the current action, while the remaining 51% iterate over user inputs or query results from previous actions. For example, when a user labels all messages as read on a webpage and clicks the "submit" button, the list of messages (which are query results from a previous action) are sent to the current action to be processed iteratively.

This observation suggests that if a query returns a large number of tuples, then the controller or page renderer will likely be slow due to individual (rather than batched) tuple processing. Such queries also bring scalability issues: as the database size increases, the time spent on processing or rendering the query result may increase linearly or even superlinearly, making the application unable to scale well with its data. To quantify this, we analyze the result size of each query. Specifically, we check whether each query returns an increasing number of tuples with increasing database size.

To do so, we first need to estimate the size of the query results. In Rails, a query can return a large amount of results when the database contains more tuples (we call it an "unbounded result") in all but the following cases: (1) the query always returns a single value (e.g., a `COUNT` query); (2) the query always returns a single record (e.g., retrieving using a unique identifier); (3) the query uses a `LIMIT` keyword bounding the number of returned records. Our static analyzer examines all queries in each application and determines whether a query returns bounded or unbounded result based on the query type discussed above. We then count the average number of both queries.

Our static analyzer shows that **36% of queries return unbounded numbers of records.**

Turning queries from returning unbounded to bounded results often requires changing the application logic. Pagination and incremental loading are common techniques to bound the amount of data shown on a webpage. For instance, developers can change an application to to display messages over a number of pages rather than a single one. This allows the messages to be incrementally loaded as the user scrolls down the page. We manually apply pagination to three popular-visited pages from three different applications, where these pages are the most rendering-time consuming pages in their respective application. We evaluate the rendering time before and after pagination, with the results shown in Figure 5.5.

5.1.2.5 *Evaluation of anti-pattern fix*

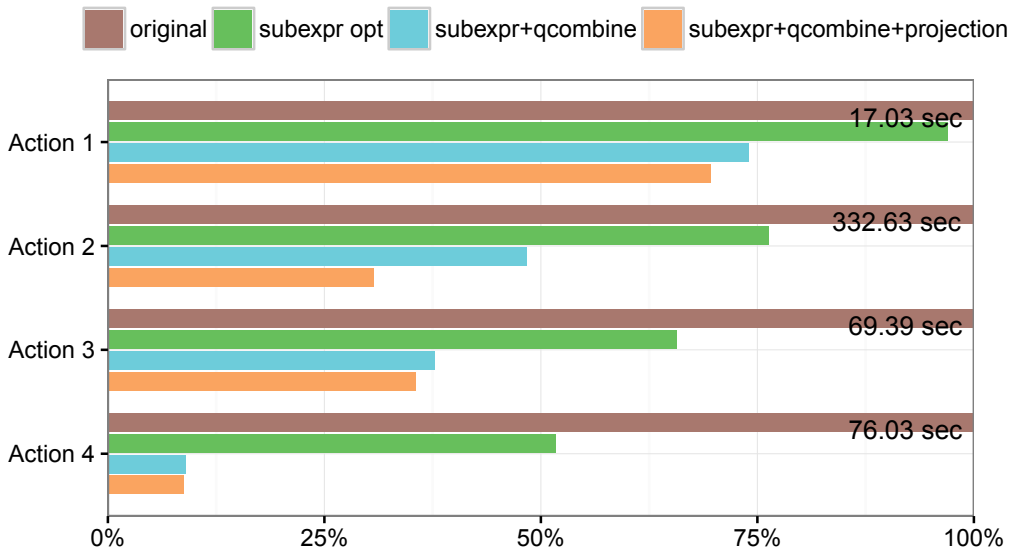


Figure 5.3: Performance gain after applying all optimizations

We choose the four most query intensive actions in the redmine [54] application to evaluate the fix of the first three optimizations mentioned above. For each action, we evaluate both the query time and the size of data transferred from DBMS. Figure 5.3 and Figure 5.4 show the results. Despite the overhead described in Sec. 5.1.2.1 and Sec. 5.1.2.2, each optimization still improves the overall performance. Adding up all optimizations significantly reduces the query time, up to 91%. For transfer size, the most reduction comes from fusing queries and eliminating redundant data retrieval. The amount of data transferred in the three actions is reduced by more than 60%. These results show the significance of the inefficiencies that we have identified and the potential performance gain that can be be obtained.

We also evaluate the fix of changing queries returning unbounded result to bounded with pagination. Our evaluation shows that pagination provides impressive performance gains, reducing rendering time by 85%. As such, building tools that can identify such queries and suggest possible code changes will be an interesting area for future research.

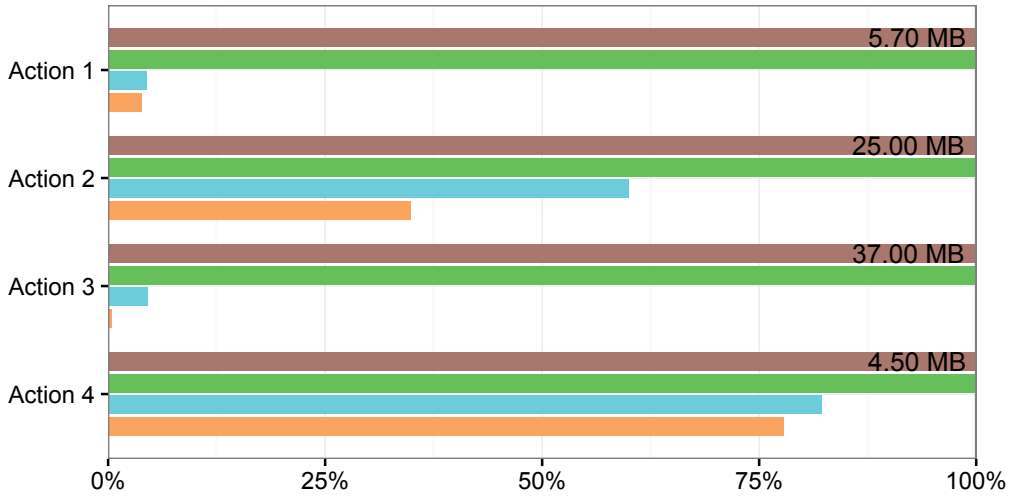


Figure 5.4: Transfer size reduction

5.1.3 Anti-pattern across actions

In this section we present our findings on performance issues and potential optimizations beyond individual actions.

By default, Rails does not maintain state after an action returns (i.e., the resulting webpage has been generated). Although Rails provide APIs for sharing states across actions (e.g., caching APIs that store fragmented pages or query results), using them complicates program logic and introduces user-defined caches that need to be maintained. Unfortunately, not caching query results often leads to redundant computation being performed across multiple actions.

We analyze our chosen applications and find two query patterns that can benefit from cross-action caching. Below we first introduce these two patterns, and then discuss how AFGs can be used to automatically identify and quantitatively measure how common these patterns are. Finally, we manually implement caches to evaluate the potential performance benefits of cross-action caching.

5.1.3.1 Shared queries across consecutive actions

Syntactically equivalent queries. We find that common practices in Rails applications can cause

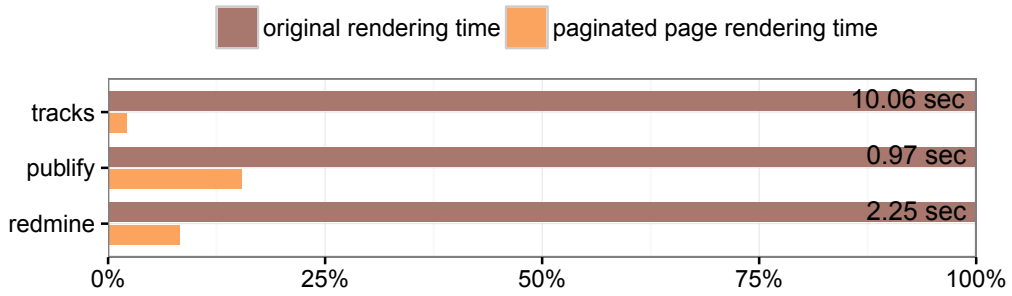


Figure 5.5: Pagination evaluation. The original page renders 1K to 5K records, as opposed to 40 records per page after pagination.

many syntactically equivalent queries to be issued across actions. First, Rails supports filters. A filter f of a class C is executed every time a member function of C is invoked. Consequently, the same queries in f are issued by many different actions as they invoke functions of C. Checking user permissions represents one such type of filters that is shared across many actions. Second, many pages share the same partial layout. Consequently, the same queries are repeatedly issued to generate the same page layout in different actions. For example, a forum application shows the count of posts written by a user on almost every page after the user logs in.

This pattern reveals an optimization opportunity — we can identify queries that will probably get issued again by later actions, and cache their results to speed up later actions, assuming that the database contents have not been altered.

Template equivalent queries. We observe that many queries with the same template, i.e., queries with equivalent structures but with different parameters, are issued across actions. One major reason for this is *pagination*, a widely used programming practice to reduce rendering time as discussed in Sec. 5.1.2.4. As a user visits these pages, the same actions with different parameters, such as page ID, are repeatedly invoked, thus issuing queries with the same template (e.g., the ones shown in Listing 5.4).

This pattern reveals an opportunity similar to common subexpression optimization. For example, if the sorted posts computed when processing `Q1` are cached (i.e., the query that corresponds to

`Post.order('created')`), then `Q2` can simply return the next batch of posts from the ordered list.

Listing 5.4: Q1 and Q2 are issued when visiting page1 and page2, sharing the same query template

```
1 Q1: SELECT * FROM posts ORDER BY created LIMIT 40 OFFSET 0
2 Q2: SELECT * FROM posts ORDER BY created LIMIT 40 OFFSET 40
```

We apply static analysis on the AFG to quantitatively understand how common the above two patterns are across different applications. Specifically, we analyze every *previous-current* action pair that is linked by the *next action* edge in the AFG described in Sec. 5.1.1. For each query `Q` in the *current* action, we check if there exists a query `Q'` from the corresponding *previous* action that is generated by the same code (e.g., the same filter or the same function) as `Q`. If such a `Q'` exists, `Q` and `Q'` share the same query template. We further examine their parameters to see whether they are syntactically-equivalent queries — if `Q` only takes constant value or the same data from the session cache as parameters, we consider it to be syntax-equivalent to `Q'` (i.e., same template and same parameter)(①); if `Q` takes user input and/or utility function result as parameter, we consider it to be template-equivalent to `Q'` with potentially different parameters(②).

We calculate the average number of the two types of queries (① and ②) issued in an action. If a query has a syntax or template-equivalent peer in any previous action, the earlier query can cache the result to be used by the later query[3].

Our static analyzer shows that **20% and 31% of the queries are syntactically or template equivalent to a query in a previous action respectively.**

5.1.3.2 Evaluation of anti-pattern fix

We evaluate the benefit of caching using sugar [62], a discussion forum application. Since it is difficult to predict a user's complete page-access sequence, our evaluation focuses on a common visit pattern on paginated webpages: after visiting the first page, users often visit the following pages to see subsequent results. We choose four slowest paginated webpages that show 1) latest discussions; 2) popular discussions; 3) latest posts; and 4) recent invitations. We populate the

[3]We imagine such cache can be invalidated similar to standard application caches [65].

database following the data distribution of online forum website as mentioned in Sec. 1.2, with 6GB data in total. Each page takes 0.2 to 24.3 seconds to generate, with more than 95% of time spent on database queries. We measure the total query time for each page since our caching optimizations only affect queries.

We then use the AFG to automatically identify which query results to cache, manually cache the results, and measure how much query time is saved. The results are shown in Figure 5.6. Although the query time in the first page becomes slightly longer (at most 5%) due to caching temporary results, queries in following pages are significantly faster, by more than 5× in all actions (up to 245×). The results illustrate an impressive benefit of caching to improve application performance.

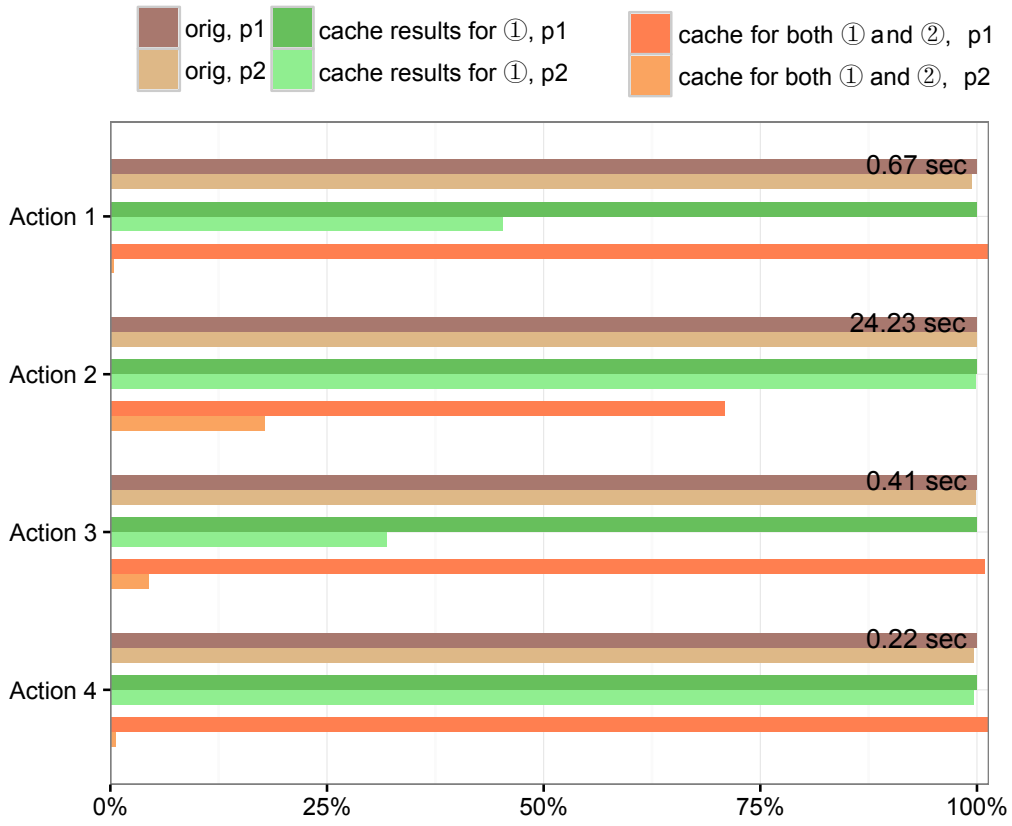


Figure 5.6: Caching evaluation on paginated webpages. p1 and p2 refer to the current the next pages, respectively. (we observed no significant latency difference among the pages that are reachable from p1). The x-axis shows the query time percentage with original first page's query time as basline.

5.2 PowerStation*: IDE Plugin for Fixing Anti-Patterns*

Since some of the above-mentioned anti-patterns can be automatically fixed, we build POWERSTATION, an IDE plugin for Ruby on Rails (Rails) applications to fix 6 anti-patterns. These 6 anti-patterns are among the anti-patterns described above and those found from the study of the applications' bug-tracking systems [155]. We demonstrate POWERSTATION with these 6 anti-patterns while it can be extended to fix other anti-patterns as well. We have integrated POWERSTATION into a popular Rails IDE, RubyMine [58], so that Rails developers can easily benefit from POWERSTATION to improve the efficiency of their applications. The source code of POWERSTATION is available on GitHub [46].

5.2.1 Detecting and fixing anti-patterns

Now we introduce in detail how are these 6 anti-patterns are detected and fixed.

Redundant data retrieval. For every Read query node n in the ADG, POWERSTATION first computes the database fields loaded by n that are used subsequently. This is the union of the *used fields* associated with every out-going data-dependency edge of n. POWERSTATION then checks if every loaded field is used. For every unused field, POWERSTATION either deletes n, if none of the fields retrieved by n are used, or adds field selection `.select(:f1, :f2, . . . )` to the original query in n so that only used fields `f1`, `f2` are loaded.

Common sub-expression queries. POWERSTATION checks every query node q_0 in ADG to see if q_0 has out-going data-dependency edges to at least two query nodes q_1 and q_2 in the same control path. If that is the case, then by default, Rails would issue at least two SQL queries that share common sub-expression q_0 at run time, one composing q_0 and q_1 and one composing q_0 and q_2, with the latter unnecessarily evaluates q_0 again. This can be optimized by changing the query plan and caching the common intermediate result for reuse. Doing so requires issuing raw SQL commands that are currently not supported by Rails `ActiveRecord` APIs.

Loop invariant queries. A query is repeatedly issued in every iteration of a loop to load the same database content. POWERSTATION first identifies all query nodes inside loop bodies in ADG.

For each such node n, it checks the incoming data-dependency edges of n. If all of these edges start from outside the loop L where n is located, then n is identified as a loop-invariant query. To fix this, POWERSTATION inserts a new Assign statement before the start of the loop, where a newly created variable v gets the return value of the loop-invariant query, and replaces every invocation of the loop-invariant query inside loop L with v.

Dead store queries. SQL queries are repeatedly issued to assign the same memory object with different database content, without any use of the memory object in between, making all but the last query unnecessary. POWERSTATION checks every ADG node that issues a reload query, i.e., `o.reload`, and checks its out-going data-dependency edge. If there is no such edge, i.e., the reloaded content is not used, then the query is marked as a dead-store query that is deleted by POWERSTATION.

API Misuses. Different ORM APIs can be used to retrieve the same results from the database, but they can differ drastically in terms of performance. Examples of API misuses can be found in [155]. POWERSTATION uses regular expression matching to find inefficient API misuses. Since these API mis-use patterns are simple, POWERSTATION also synthesizes patches for each API mis-use pattern through regular expressions.

Inefficient rendering. While rendering a list of objects, helper functions are often used to render a partial view for one object at a time, with much redundant computation repeated for every object. POWERSTATION checks every loop in the ADG to see if it iterates through objects returned by queries and contains a Rails view helper function such as `link_to` in every loop iteration. If so, POWERSTATION identifies the code as having the inefficient rendering problem. To fix this, POWERSTATION hoists the helper function out of the loop, assigns its result to a newly created variable, and replaces the original helper function in the loop with `gsub` (a string substitution function) on the newly created variable. Doing so removes the redundant rendering that is performed on every loop iteration in the original code.

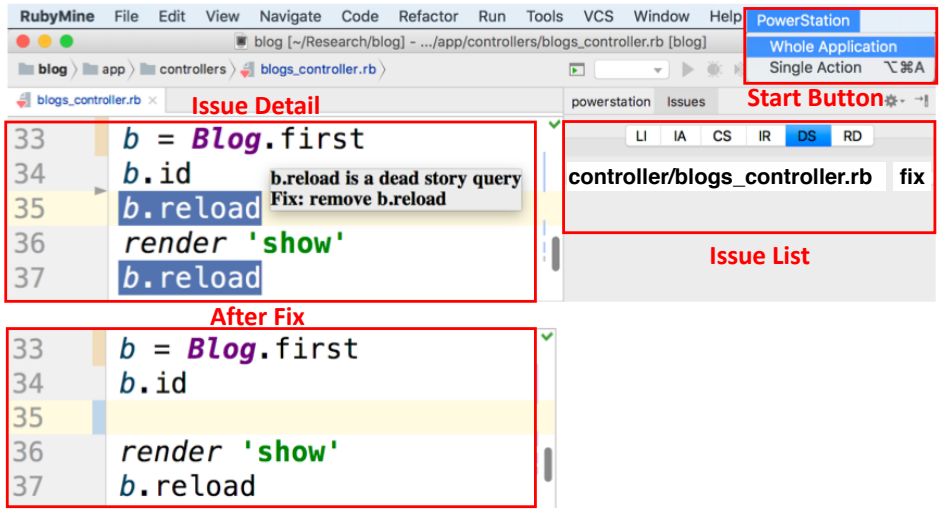


Figure 5.7: Screenshots of POWERSTATION IDE Plugin

5.2.2 PowerStation *IDE integration*

We have implemented POWERSTATION as an IDE plugin for RubyMine [58], a popular IDE for Ruby on Rails. A screenshot of POWERSTATION is shown in Figure 5.7. By pressing the "PowerStation" button in the IDE, users can choose an analysis scope, "Whole Application" or "Single Action," and launch POWERSTATION analysis accordingly. Our website includes a tutorial on how to use the tool [46].

Issues list. The right panel, as highlighted in Figure 5.7, lists all the inefficiencies detected by POWERSTATION, each represented by a button displaying the file where the inefficiency is located. By default, all the inefficiencies found in the project are listed. Users can also choose to display inefficiencies of a particularly type as shown in Figure 5.7—loop invariant queries (LI), dead store queries (DS), unused data retrieval queries (RD), common sub-expression queries (CS), API misuses (IA), and inefficient rendering (IR).

Issues highlight. Clicking the file button in the issue list will navigate users to the corresponding file in the editor, with the inefficient code highlighted. Hovering the cursor over the highlighted code will display the reason for highlighting, as shown in Figure 5.7.

Issue fix. Clicking the "fix" button next to each issue in the issue list will pop up window asking the user whether she wants POWERSTATION to fix the issue. If so, POWERSTATION will synthesize a fix, and display the fixed code in the editor panel. At that point, the original "fix" button becomes an "undo" button, allowing users to revert the fix if needed.

Table 5.2: Inefficiencies detected by POWERSTATION in 12 apps

App.	Loop Inv	Redundant data	Common sub-expr	API misuses	Inefficient render	SUM
Discourse	0	16	106	85	0	207
Lobsters	0	2	0	45	5	52
GitLab	0	14	92	23	1	130
Redmine	0	11	101	59	0	171
Spree	0	22	0	20	0	42
Ror-ecommerce	0	3	0	11	0	14
Fulcrum	0	12	15	2	1	30
Tracks	0	23	30	30	1	84
Diaspora	1	55	36	57	0	149
Onebody	0	17	39	76	0	132
FallingFruit	0	24	12	4	5	45
OpenstreetMap	0	89	60	16	0	165
SUM	1	288	491	428	13	1221

5.2.3 Evaluation

We evaluated POWERSTATION on 12 open-source web applications (as listed in Table 1.1). POWERSTATION can automatically identify 1221 inefficiency issues and generate patches for 730 of them (i.e., all but the common sub-expression pattern). The distribution is listed in Table 5.2. We randomly sampled and examined half of the reported issues and the suggested fixes, and found no false positives. Due to the limited resource and time, we reported 433 issues with 57 of them already confirmed by developers (none has been denied). POWERSTATION static analysis is fast, taking

only 12–625 seconds to analyze the entire application that ranges from 4k to 145k lines of code in our experiments on a Chameleon instance with 128GB RAM and 2 CPUs. Developers can also choose to analyze one action at a time, which usually takes less than 10 seconds in our experiments.

5.3 Summary and Future Work

In this chapter, we showed our study the database-related inefficiencies in 27 real-world web applications built using the Rails ORM framework. We built a static analyzer to examine how these applications interact with databases through the ORM. We also profiled some applications using workloads that follow real-world data distributions. The we presented POWERSTATION, a new tool that automatically detects and fixes some of these performance issues. POWERSTATION integration with RubyMine provides an easy way for Rails developers to avoid making performance-degrading mistakes in their programs. We have used POWERSTATION to identify and fix many performance-related issues in real-world applications, and will extend POWERSTATION to tackle further performance anti-patterns as future work.

Chapter 6

HYPERLOOP: VIEW-DRIVEN OPTIMIZATIONS WITH DEVELOPER'S INSIGHTS

In this chapter, we present HYPERLOOP, a system for optimizing database-backed web applications (DBWAs). Current approaches in optimizing DBWAs focus on partitioning the application among the browser, application server, and the database, and rely on each component to optimize their portion individually without developer intervention. We argue that this approach misses the goal of DBWAs in optimizing for end-user experience, and fails to leverage domain-specific knowledge that DBWA developers have. For instance, a news website might prioritize loading of the news headlines, even at the expense of slowing down loading of other visual elements on the page. HYPERLOOP illustrates the idea of *view-driven optimization* by allowing developers to specify *priorities* for each of the elements on the webpage, and uses such information to drive optimization of the entire webpage. HYPERLOOP currently focus on optimizing for render time of webpage components, and our preliminary results show that this view-driven approach can substantially improve DBWA performance by leveraging developer provided application knowledge.

6.1 View-driven Optimization

Webpages in web applications typically consist of multiple view components (e.g., tables, buttons, text blocks, etc.), with each component rendered using different code paths. In view design literature [67, 21], it is well-known that not all view components are created equal: a user might perceive a news website to have loaded already once the news headlines have appeared on the screen, even though the rest of the page has not been fully loaded yet. This has led to the development of asynchronous loading libraries allowing developers to modify their DBWA code to load page elements at different times, even at the expense of slowing down the rest of the page. Such tradeoffs

are prevalent in DBWA design: dividing a long list of items into multiple, shorter lists and rendering each of them across multiple pages (pagination), pre-loading data that is likely to be used in subsequent pages that the user will visit (caching), etc. Today developers make such tradeoffs manually by trying to change the code across layers and see how that impacts each page element, and repeating the process until the best design is reached. However, we are unaware of any system that would systematically capture such "view-specific" knowledge from developers, and exploit them for optimization of DBWAs.

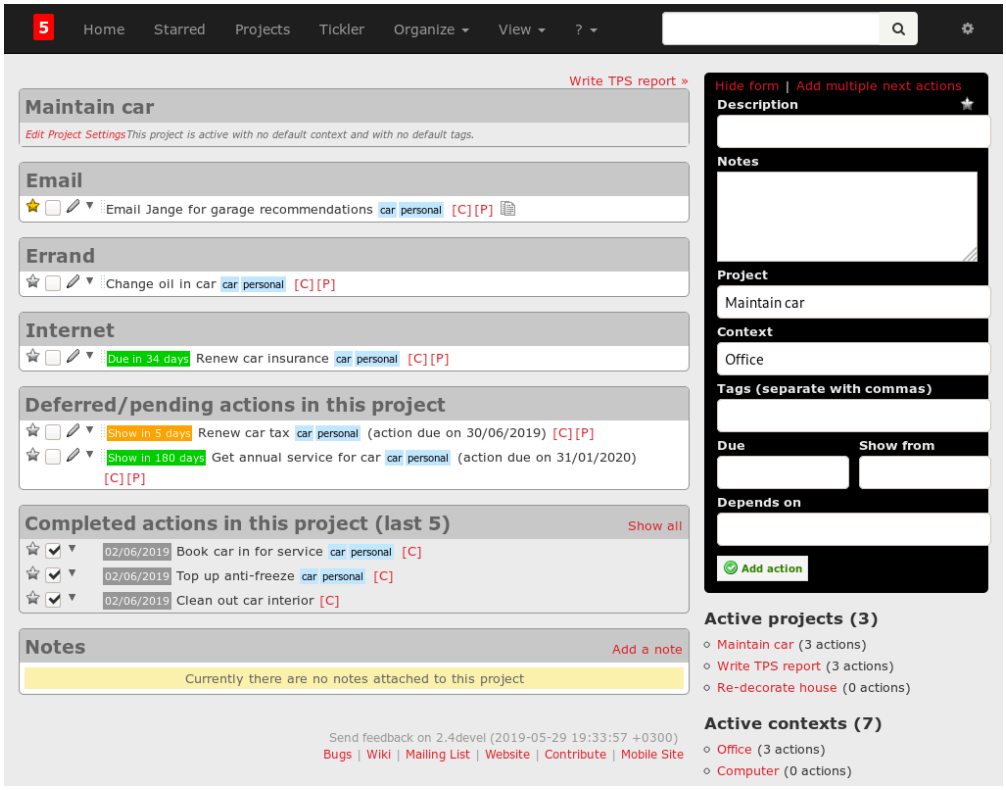


Figure 6.1: An example webpage from Tracks.

We argue that this page element-wise "trial-and-error" optimization of DBWAs is wrong. Instead, we believe the optimization of DBWAs should be *view-driven* by the domain-specific knowledge that developers possess. In this chapter, we describe HYPERLOOP, a new system we are designing with that purpose. HYPERLOOP concretizes view-driven optimization by allowing DBWA developers

```
1: class User:                          7:  @user = User.where(:id=?)
2:    has_many: projects => Project     8:  @projects = @user.projects
3: class Project:                       9:  @left_projects = @projects.select
4:    has_many: todos => Todo               {|p| p.todos.where(done=>false).count>0}
5: class Todo:                          10: @right_projects = @projects.select
6:    has_one: note => Note                 {|p| p.status='active'}

11: <% for p in @left_projects %>
12:    <a href=..%=p.id%..><%=p.name%></a>
13:    <% for t in p.todos %>:
14:        <li> t.name, t.context </li>
15:        <li> t.note </li>
16:        <span> t.tags.join{} </span>

16: <% for p in @right_projects>
17:    <li> <% p.name> </li>
18:    <span><% p.todos.count></span>
```

```
L7   Q1: SELECT * FROM users WHERE id = ?
L8   Q2: SELECT * FROM projects WHERE user_id = ?
L9   Q3: SELECT COUNT(*) FROM todos WHERE done=false AND project_id = ?
…    …
L13  Q4: SELECT * FROM todos WHERE project_id = ?
…    ….
L15  Q5: SELECT * FROM notes WHERE todos.note_id = note.id.
…    …
L18  Q6: SELECT COUNT(*) FROM todos WHERE project_id = ?
…    …
```

Figure 6.2: Abridged code to render the webpage shown in Figure 6.1, with blue numbers indicating which line of Ruby code at the top generated the query.

to provide domain-specific knowledge as *priority labels* for each webpage element to indicate those that should be rendered first.[1] Given priorities and a resource budget (HYPERLOOP currently supports specifying total memory available to store data in memory), we envision HYPERLOOP to automatically analyze the DBWA code to devise a plan to render each of the pages in the application, with the goal to reduce the render time of the high priority elements as much as possible. HYPERLOOP achieves this by applying different optimization across all three tiers, from changing the layout of each page to customizing data structures to store persistent data in memory. To help developers assign priorities, HYPERLOOP comes with a static analyzer that estimates render times

[1]Other notions of domain-specific knowledge are certainly possible, e.g., impact on user experience, interactivity, etc. We currently use the time to render as it is an easily quantifiable measure.

and presents the results via HYPERLOOP's user interface.

While we are still in the early implementation phase of HYPERLOOP, our initial experiments have shown promising results: we can improve the start render time (i.e., time taken for the first element to be displayed on the browser's screen after initiating the HTTP request) of high priority webpage elements in real-world DBWAs by 27×. We believe this illustrates the potential of view-driven optimization of DBWAs, with HYPERLOOP presenting an initial prototype that implements this concept.

6.2 HyperLoop Overview

We now discuss how DBWA developers can use HYPERLOOP to improve their applications. Figure 6.2 shows a code fragment from Tracks [64], a popular Ruby on Rails DBWA for task management. Figure 6.1 shows a page from the Tracks listing of projects created by a user, where each project contains a list of todo actions. This page has three panels. The left panel shows a list of undone projects (we call a project "undone" if it contains undone todos), with the detail of each todo shown when clicked. The upper right panel shows a form where user can add a new todo, and the bottom right panel shows a list of active projects and its todo count.

Figure 6.1 shows the abridged DBWA code used to render this page. Lines 1-6 show how persistent data is organized into the `User`, `Project` and `Todo` classes. It also specifies the relationship between the classes, for instance, a project has many todos, as implemented as foreign key constraint in the database. Line 8 retrieves the list of projects that belongs to the current user from the database and into the Ruby array variable `@projects`. Line 9 then filters `@projects` to return those to be rendered on the left panels based on the number of undone todos. The filter for the right panel selects the active projects in Line 10. The code uses the `where` API provided by the Rails library which translates the object query to SQL queries as shown in the bottom of Figure 6.2.

Lines 11-18 show the view file written in HTML with embedded Ruby code. The bottom of Figure 6.2 shows the SQL queries translated by the Rails library to generate this page. Q1 retrieves the current user, followed by Q2 to retrieve her projects. A number of queries (e.g., Q3) are issued to get the count of undone todo for each project. Similarly, some queries are issued to get

the todos for each project (Q4) and note for each todo (Q5) which are on the left panel, as well as todo count for each project (Q6) on the right.

HYPERLOOP allows developers to improve performance via its view-centric interface. Figure 6.3 shows the HYPERLOOP workflow. To use HYPERLOOP, the developer only needs to label the high priority elements on the webpage, and HYPERLOOP will automatically analyze the application code to suggest different ways to render the page by reducing the render time of high priority elements, while possibly increasing the render time of the low priority ones. We envision that HYPERLOOP will make different tradeoffs based on how the elements are labeled, and propose different render plans to the developer to further refine.

To help developers assign priorities, HYPERLOOP comes with an analyzer that statically estimates the load time of each page element, given the amount of data currently stored in the database. The estimates are presented to the developer as a heatmap. We envision other analyses will also be useful in aiding the developer to assign priorities, for instance the amount of memory used, query plans used to retrieve rendered data, etc.

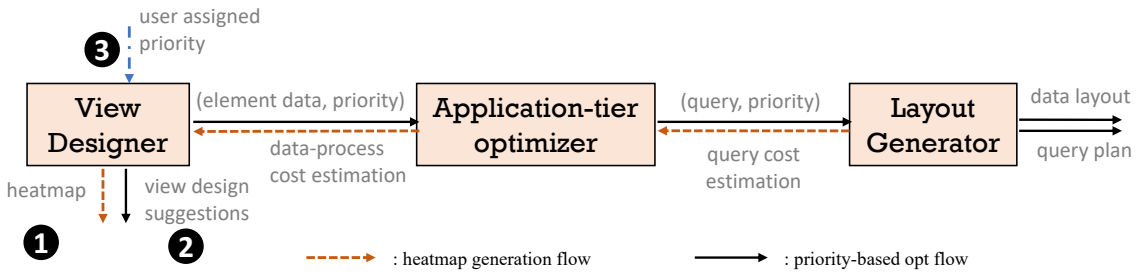


Figure 6.3: HYPERLOOP workflow

For the example shown in Figure 6.1, suppose the developer decides to label the list of undone projects as high priority, based on the current load time estimate. Given this information, HYPERLOOP will suggest different ways to render the page and optimizes the data processing leveraging priority. For instance, loading the undone projects panel asynchronously (to be discussed in Sec. 6.6), and furthermore storing them away in a dedicated list in memory for fast retrieval (to be discussed in Sec. 6.7). If the developer instead labels the active projects as high priority,

and the undone projects as low priority, then HYPERLOOP will generate different rendering plans, for instance paginating the list of active projects across multiple pages (if there are a lot of active projects) while approximating the undone projects from previously cached data. HYPERLOOP is designed to be interactive: if the developer prefers not to paginate as it disrupts the user experience, she can indicate her preference using the HYPERLOOP user interface (and potentially reassigning the priorities), and HYPERLOOP will devise another render plan for the page.

After the developer picks one of the rendering plans, HYPERLOOP will automatically apply changes to the application code. As our current focus is on leveraging priorities for reducing load time, in the next sections we discuss different code changes we have implemented based on priorities, along with evaluations using real-world DBWA benchmarks. Our current prototype is designed for applications built using the Model-View-Controller (MVC) architecture [140] where the application is hosted on a single web server with source code available to be modified by HYPERLOOP.

6.3 Priority-Driven Optimization

HYPERLOOP applies various optimization to different webpage elements depending on their priorities provided by the developer. These optimization often provides speedup to certain web-page elements (i.e., high-priority ones) at the cost of the loading time or the rendering quality of other elements (i.e., low-priority ones), and hence is not explored by traditional optimization techniques. We present a few optimization of this type below. We will then discuss how we implement these optimization in the next few sections.

Asynchronous loading. Asynchronously loading a view element e allows web users to see e before other potentially slow elements get loaded. The downside is that the total amount of computation or the total number of queries issued to the database may increase, because previously shared computation across asynchronously loaded components can no longer be shared. This optimization can be applied to high-priority elements, and will require view changes (Sec. 6.5) and application-tier changes (Sec. 6.6).

Pre-computing. While generating a web-page p_1, one can pre-compute contents needed to generate the next page p_2, which the web user is likely to visit next through a link on p_1. This will

speedup the loading time of p_2 at the cost of the loading time of p_1. HYPERLOOP supports this optimization only when the developer provides high priority to the link on p_1 that points to p_2. It is implemented through our app-tier optimizer (Sec. 6.6) .

Optimizing for heavy reads or writes. There are often both read and write accesses to the same database table. Our database layout generator (Sec. 6.7) can optimize for either heavy-write workload, at the cost of read performance, or heavy-read workload, at the cost of write performance, based on the priority information provided by the developer.

Pagination. It often takes long time to retrieve and display a long list of items. One way to improve performance is to only show the first K items in the list and allow users to navigate to subsequent pages to view the remaining items. This change can greatly improve the loading time of the list, but at the cost of taking users longer time to see later part of the list. It can be applied to a list that contains both high and low priority items and items, or an overall low priority list whose content-viewing experience is less important than its loading speed. This is implemented in HYPERLOOP's view designer (Sec. 6.5) and app-tier optimizer (Sec. 6.6).

Approximation. Approximation can be applied to many aggregation queries, such as showing "you have more than 100 TODOs" instead of "you have 321 TODOs." Like pagination, approximation presents a tradeoff between loading speed and the quality (accuracy) of the content, and is suitable for low priority elements. This is implemented in HYPERLOOP's view designer (Sec. 6.5) and the app-tier optimizer (Sec. 6.6).

Using stale data. Caching data in memory and updating only periodically can improve performance at the cost of data quality and freshness. Priorities provided by developers can help HYPERLOOP determine which page element to cache. This is implemented in HYPERLOOP's app-tier optimizer (Sec. 6.6) and layout generator (Sec. 6.7).

6.4 HyperLoop's User Interface

HYPERLOOP provides a unique interface for developers to understand the performance of their application and provide priority information. First, it presents the statically estimated cost (to be discussed in Sec. 6.6) to render each HTML element as a heat map in the browser. This cost includes

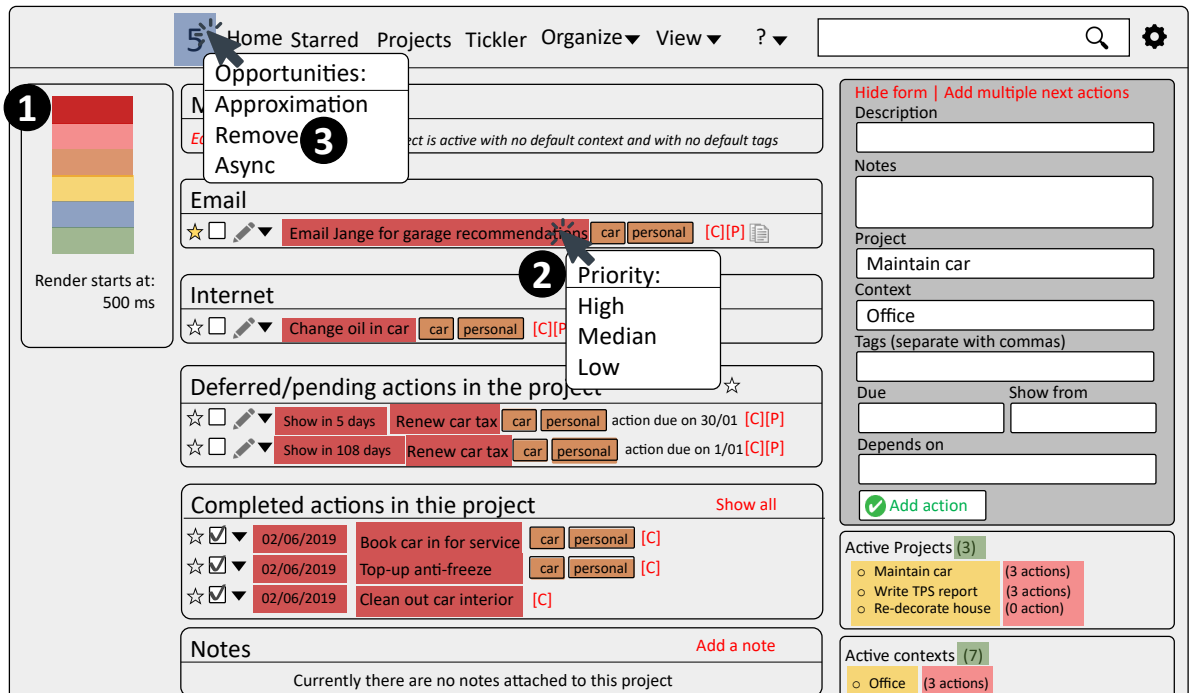


Figure 6.4: Heatmap showing the estimated loading cost of each webpage element, along with priority assignment and rendering recommendations generated by HYPERLOOP. (1) renders the heatmap; (2) assigns priority; (3) shows the opportunities.

the time to retrieve the data from the database and process it in the application server. Figure 6.4(a) shows an example heat map of the webpage shown in Figure 6.1, where darker color means higher cost. For example, the left and bottom right panels have a high cost because of the large number of projects stored in the database, making the queries that involve them (e.g., Q1 in Figure 6.2) slow.

As discussed in Sec. 6.2, after examining the estimates, the developer can click on an HTML element on the page to indicate priority. We intend the interface to support applying the same priority to a group of elements after highlighting them. HYPERLOOP supports different priority levels as shown in Figure 6.4.

After assigning priorities, the developer would click on the "analyze" button on the right. HYPERLOOP then analyzes each element together with its priority, and provides a list of suggestions

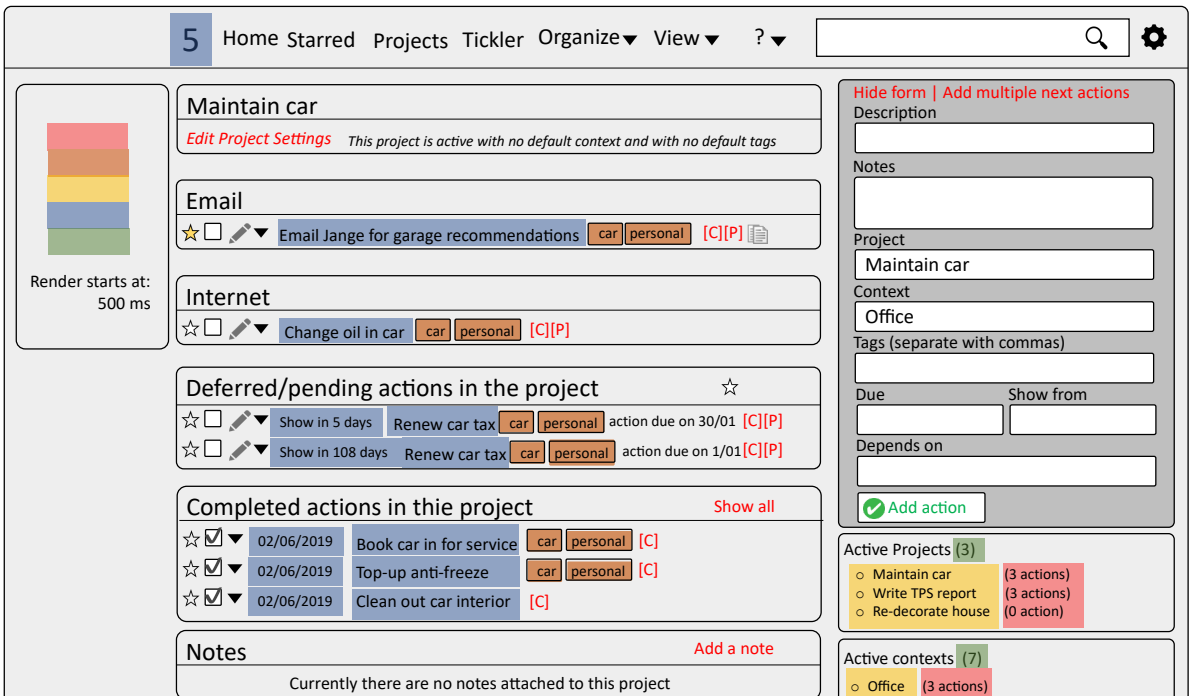


Figure 6.5: Heatmap after optimizing for undone projects on the left panel shown in Figure 6.1.

as shown in Figure 6.4. Some of these suggestions require further user input, for instance, how many objects to show on each paginated page. All of the suggestions are only related to webpage look and functionality designs, and the developer needs no database knowledge to choose a suggestion. We describe the list of suggested changes in Sec. 6.5.

The developer can right-click each element to view the rendering plan generated by HYPERLOOP. After choosing one of the plans, HYPERLOOP will change the application, re-estimates the cost, and renders the new webpage with a new heatmap, like the one shown in Figure 6.5, where the left panel is now loaded first.

HYPERLOOP not only suggests the rendering plan but also optimizes query processing based on priority assignment, as described in Sec. 6.6.1 and Sec. 6.7.2. These optimization strategies often involve tradeoffs, for instance, accelerating a query that retrieves data for high-priority HTML tags by slowing down queries for low-priority tags slightly. HYPERLOOP renders a list

of such optimizations in the IDE and lets developers enable and disable them individually (by default HYPERLOOP applies all optimizations), as shown in Figure 6.6. The developer can then ask HYPERLOOP to regenerate the heatmap to see the effect of certain optimization(s). Doing so allows the developer to do A/B testing and understand how these optimizations interact with each other. Furthermore, HYPERLOOP can show the refactored code (after choosing rendering recommendations and a set of optimizations) if the developer wants to know the change in more detail, as shown in Figure 6.6.

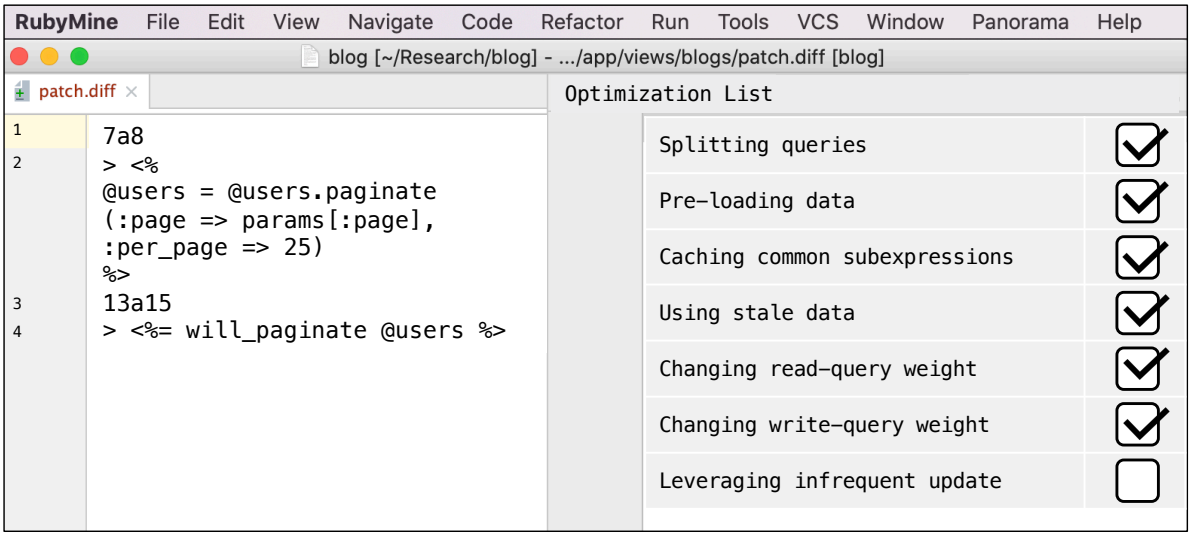


Figure 6.6: Left: refactored code. Right: a list of optimizations that developer can enable/disable individually.

HYPERLOOP supports assigning priority to both HTML tags as well as a form or a hyperlink to another webpage. If a form is assigned high priority, HYPERLOOP will attempt to reduce the time taken to process the form by changing the in-memory data layout of persistent data (to be discussed in Sec. 6.7). If the hyperlink is assigned high priority, HYPERLOOP will optimize the render time of the linked page, possibly by increasing the time taken to load the current page. As mentioned in Sec. 6.2, the developer can visualize such tradeoffs and reassign priorities using the HYPERLOOP interface as needed.

We next discuss the design of HYPERLOOP as shown in Figure 6.3 and how different tradeoffs are made given priority information.

6.5 View Designer

The *View designer* analyzes and transforms view files that define webpages' look and functionality. Its purpose is to identify which application tier object is rendered by which HTML element, and passes this information to the *Application-tier optimizer*. It also carries out priority-driven optimization as described below.

Asynchronous loading. To asynchronously load an HTML element e, the *View designer* splits the original HTML file into two files, one rendering e and the other rendering the rest of the page. To do so, the *View designer* first creates a new view file v to render e, and then creates new code to reside on the application server to compute the contents needed by e to render the view file v, and finally replaces e in the original view file with an AJAX request.

Pagination. The *View designer* detects pagination opportunities by checking whether an HTML element is rendering a list of Ruby objects in a loop. After the developer decides to paginate an element, the *View designer* rewrites the view file to render a constant number of elements first and adds a page navigation bar, as described in prior work [157]. It passes the design decision to the *Application-tier optimizer* who will change the query to return limited results (by adding `LIMIT` and `OFFSET`). Pagination itself can greatly accelerate the start render time. For example, paginating the left panel of Figure 6.1 to show 20 projects per page (out of 2K projects altogether) accelerates the panel rendering by 27×.

Approximation. The *View designer* detects approximation opportunities by checking if an HTML element is displaying a value that is returned by an aggregation query. Once the developer accepts an approximation optimization opportunity, the *View designer* changes the view file to add "at least" or "at most" before the aggregation value and passes it to the *Application-tier optimizer* to change the query to count only N values by adding a `LIMIT` clause. Doing so may reduce the long query by 2.1× and a page's start render time by 1.6×.

6.6 Application-Tier Optimizer

We leverage the analysis framework built in POWERSTATION that enables a wide variety of optimizations. While some optimization always improves performance, e.g., adding projection to load only fields being used, others, however, requires making tradeoffs. Next we give examples on how to extend this framework to support priority-driven optimization.

6.6.1 Priority-driven optimization

We now discuss a few examples on how the *Application-tier optimizer* supports priority-driven optimization.

Example 1: Splitting queries. Very often DBWAs would issue a query to retrieve one set of data that will be filtered/processed in multiple ways to render multiple view components, as doing so can reduce duplicate work in rendering related view components. For example, a web page may show both a list of projects and a total count of these projects. The application can issue a single query to retrieve all projects while counting them in memory. Another example is the query to retrieve all projects (Q1 in Figure 6.2) into a Ruby array `@projects` that is filtered separately in memory to obtain `@left_projects` and `@right_projects`.

Although the original query helps to reduce the total number of queries issued and the overall computation required to render the page, it could be sub-optimal if the multiple view components supported by it have different priorities. Specifically, to carry out an asynchronous loading optimization discussed in Sec. 6.3, the *Application-tier optimizer* splits a shared query if the result is used in asynchronously loaded elements. For example, if the left panel has high priority and is decided to be asynchronously loaded from other parts, the optimizer splits Q1 into Ql and Qr as shown below:

Listing 6.1: Example application code illustrating query splitting

```
1 Ql: @left_projects=user.projects.where(undone.count>0).include(todos,
    include(note))
2 Qr: @right_projects=user.projects.where(status=`active').include(todos.count
    )
```

After the split, each query retrieves the data shown on the corresponding panel. Doing so causes the projects shown in the two panels to be retrieved in separate queries, but allows separate optimization of Ql, such as eager-loading of todos and notes for the left panel query (`where` and `include` are query functions to filter data using predicate and to eager-load the associated objects). Besides, the *Application-tier optimizer* will pass the design decision to the *Layout generator* and the splitting will allow generator to customize a layout for Ql (described in Sec. 6.7). As an illustration, with 4K total projects and 50% of them to show on the left panel, splitting the query and optimizing Ql as mentioned above reduces the query time of Ql from 5.1s to 0.5s, and the overall start render time from 13.7s to 6.1s.

Example 2: Pre-loading data. By default, each page is computed from scratch upon receiving an HTTP request. However, developers might know the next page(s) the user will likely visit and wish to pre-load data to accelerate loading of the next page, even at the expense of increasing the load time of the current page slightly.

An example is when a user visits the home page of a forum and then visits different posts by clicking on the hyperlink. As the home page shows only the title of each post, generating it once and caching it on the client slide would be optimal, but subsequent pages of individual posts might be impractical to cache as each may contain large images and contents. Yet, the developer might want the posts to load fast and is willing to trade off the performance of the home page. In that case, she can indicate priorities on the current page and HYPERLOOP will pre-load data accordingly. We use the Sugar forum application [62] as an illustration, where the database query retrieving the posts on its homepage selects not only the title but also the contents of each post. With a forum of 500 posts on the home page, doing so shortens each of the post page rendering time by 82% while increasing the home page load time by 12%.

Example 3: Caching common subexpressions. Common subexpressions are often shared among queries across consecutive pages [152], and subsequent pages can reuse the results of these common subexpressions with a slight overhead for the current page due to caching. The *Application-tier optimizer* applies such caching if the developer chooses to pre-load data for subsequent pages after labeling with high priority. For example, for a page that shows the first 40 recent posts, the

developer can assign the subsequent pages with high priority, as users will likely explore beyond the most recent 40 posts. The queries for the first and second pages are shown in Listing 6.2. They share the same subexpression that sorts the projects. In this case, the *Application-tier optimizer* will rewrite the queries to sort the posts, store the sorted results in a list and cache them such that the queries for all subsequent pages can simply return from this sorted list, as shown in Listing 6.3. With 10K posts, doing so slightly sacrifices the render time of the first page (an increase by 5%) but speedup the other pages by 2.3×.

Listing 6.2: Two queries sharing a common sub-expression

```
1 P1 : @posts = post.order(:created).limit(40).offset(0)
2 P2 : @posts = post.order(:created).limit(40).offset(40)
```

Listing 6.3: Common sub-expression result is cached and reused

```
1 P1 : @posts_all = post.order(:created)
2      @posts = @posts_all.limit(40).offset(0)
3 P2 : @posts = @posts_all.limit(40).offset(40)
```

Example 4: Using stale data. It may be worthwhile to show stale data in a low-priority HTML element for better performance. The *Application-tier optimizer* implements this by changing the application code to cache data rendered in labeled HTML elements, and reuse it when the same element is rendered subsequently. For example, a developer may think the right bottom panel in Figure 6.1 occupies only a small and unimportant part of a webpage and thus labels it as low priority. The *Application-tier optimizer* will then suggest to cache the list of active projects, the total count, and the count of todos for each project. Doing so eliminates most of the queries to the database when rendering the page. To evaluate this, we use 2K active projects shown on the right panel of Figure 6.1, and the total rendering time is reduced by 65% after data for the right panel is cached.

6.7 Layout Generator

In this section we first introduce the basic optimization that the *Layout generator* can perform without user interaction. Then we give examples on how it performs priority-driven optimization.

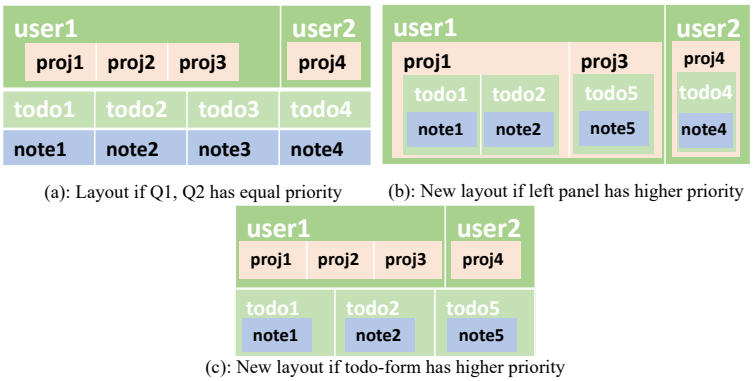
6.7.1 Basic optimizations

HYPERLOOP uses CHESTNUT to customize data layout and query plan for application queries. It leverages CHESTNUT in *Layout generator* to make the tradeoff based on the query workload. As introduced in Ch. 2, CHESTNUT first enumerates the possible data layout to store the data for each individual query as well as query plans that use particular layouts. Then it finds out the optimal layout for the entire workload by formulating an integer linear programming (ILP) problem. In this formulation, each data structure in every possible layout is assigned a binary variable to indicate whether it is included in the final layout; similarly for each query plan. It also estimates the memory cost for each data structure and the time for each query plan. For write queries, it generates one plan to update one data structure. The optimization constraints state that the overall cost of all included data structures are within the memory bound provided by the user, while the optimization goal is to minimize the overall runtime of all queries. It uses state-of-the-art solvers to solve the ILP problem and constructs the final layout accordingly. Finally, it generates an implementation of both data layouts and query plans.

6.7.2 Priority-driven optimization

We next discuss a few examples on how *Layout generator* supports priority-driven optimization. All examples below reduce the load time for the selected elements labeled with high priority at the expense of possibly slowing down other elements in the same page.

Example 1: Changing read-query weights. The *Layout generator* can design a data layout that better optimizes queries with high priority. Since it formulates the search of data layout into an optimization problem where the optimization goal is the sum of runtime cost of all queries, it can simply assign higher weights to those queries whose results are needed to render higher-priority



(a): Layout if Q1, Q2 has equal priority

(b): New layout if left panel has higher priority

(c): New layout if todo-form has higher priority

Figure 6.7: Priority-driven layout design.

HTML elements, as identified by HYPERLOOP's *Application-tier optimizer*.

For example, consider Q1 and Q2 shown in Listing 6.1 before assigning priorities. The *Layout generator* may produce a layout as shown in Figure 6.7(a). If the developer assigns higher priority to the left panel, Q1 will receive a higher weight, which results in the layout shown in Figure 6.7(b). In this layout, the `projects` belonging to a user are stored as a nested list within `user`; the `todos` are stored as nested objects in each `project`; and similarly for the `notes`. This layout is highly optimized for Q1: retrieving the `@left_projects` does not need to perform a join on `project`, `todo`, and `note` compared to using layout (a). This layout also avoids the expensive deserialization from denormalized table to nested objects. With this layout, the query time for the left panel is further reduced to 0.5s, as compared to 5.1s using the tabular layout.

Example 2: Changing write query weights. An HTML form might be assigned high priority if it is frequently used, such as the form shown in upper right of Figure 6.1 if new todos are frequently added. The queries used for new todo submission are shown below:

```
1 @todo = Todos.new(name=param['name'], ...)
2 @todo.note = Notes.new(content=param['note_content'],...)
3 @todo.save
```

In this case, developers can add larger weight to the write query creating new todo with embedded

notes. It would then generate data layout in Figure 6.7(c). Compared to (b) where inserting a new todo needs an extra query to locate the project that this todo belongs to, (c) is better optimized for adding new todos because the `todos` are stored as a top-level array and an insertion only appends to the array without the extra read query.

Example 3: Leveraging infrequent update (stale data). If an HTML element is assigned low priority, the query that retrieves data for this element can potentially read from stale data. The *Layout generator* can generate a more efficient data layout for this type of read queries. For example, if the count of all active projects (as shown in the right bottom panel of Figure 6.1) is assigned the lowest priority, then the generator will assign a very low weight to any plan that updates the data structure only used to compute this count. The generated data layout will pre-compute the count and store it in memory, which reduces the end-to-end webpage time when the count is rendered. For instance, using stale data for the active project and context count in the right bottom panel can accelerate rendering the panel by 26% (after the list is paginated). Without knowing the priority, it is unlikely to pre-compute the count because any delete query triggers a re-computation of this count, which greatly increases the total query cost.

6.8 Summary

In this chapter, we presented HYPERLOOP, a new system that helps developers optimize DBWAs. Unlike prior approaches, HYPERLOOP recognizes that developers often make tradeoffs when designing DBWAs, and leverages developers' knowledge to optimize DBWAs in a view-driven manner. Given priority information provided by the developer, HYPERLOOP automatically analyzes the application and suggests various design and code changes to improve the render time of different elements on the page. While still under implementation, preliminary results have shown that our view-driven approach is effective in improving end user experience of real-world DBWAs.

Chapter 7

FUTURE WORK

7.1 More Opportunities of Cross-Tier Optimization

There are more opportunities in leveraging the analysis across tiers for optimizations, and we will list a few in this section.

7.1.1 Leveraging front-end statistics

The statistics collected from the front end can help improve the query cost estimation. The client side often traces user statistics, for instance, how many users visit the website every day, how often do users submit comments to post, how often do users use the default values in the form, etc. These statistics can be used to estimate the query and data distribution on the backend. This is based on the observation that very often the user only provides a small amount of information when visiting a webpage or submit information with webpage form. Such information is often used to ingest a much larger amount of data inserted into the database. For instance, when a user submits an issue to a project, only filling the title and the body content of the issue (while leaving all the rest of the fields as default, which most users will do), the application actually inserts over 10 tuples into over 4 tables in the database, which means that a large amount of data is deeply correlated. By keeping track of a small amount of statistics on the client side, with the analysis of the application code to understand how the user input propagate to generate the data to be inserted to the database, we should be able to obtain a more accurate cost estimation than existing databases which collects statistics for every column independently.

7.1.2 Adjusting data layout when workload changes

In Sec. 2 we have shown how to customize data layout according to the application. However, CHESTNUT does not address the situation when workload changes. Because we observe that the application changes frequently and developers often deploy a new version while the application runs, and potentially a new version will lead to a different optimized data layout. Thus repopulating the in-memory data layout at each re-deployment is unrealistic. It is challenging to design a layout that is able to adapt online while keeping service. More trade-offs need to be considered in this scenario, for example, the trade-off between layout migration cost and query performance, between the performance of current workload and the ability to support future workload (which may not be completely known at present), etc.

7.1.3 Dynamic query rewrite optimization

In Sec. 3, CONSTROPT statically rewrites queries and verifies these rewrites, maintains a list of rewrites while dynamically searching for a match in the list for an incoming query. The downside of this approach is that it is only able to optimize a set of queries extracted through static analysis. It may fail to detect an optimization when the application code changes or when the application user inputs partial SQL query string, which is beyond the scope of static analysis. One potential solution is to dynamically rewrite and verifies queries within query optimizer. However, it is challenging to perform efficient rewrite and verification especially when the workload contains many small queries that return results quickly as the overhead of optimization can be non-trivial.

7.2 Improving Application Robustness

Cross-tier analysis can not only help improve application performance but its robustness as well. As our study about data constraint reveals [154], many issues of data constraint management exist with real-world applications. These issues include, but not limited to, inconsistent data constraints from the application and the database, incorrect implementation of a desired constraint, missing constraint in the database which may lead to a corrupted database when administrator modifies

the database bypassing the application. To build a robust application where data constraints are consistently managed, the main challenge lies in a specification of data constraints such that: 1) the constraints both defined in database and inferred from the application can be described using this specification; 2) the specification is comprehensive enough to describe most desired constraints of developers; 3) the specification can be easily compiled to verifiable correct implementation in any tier. Furthermore, we observe that the developers are not clear about their desired data constraint to begin with, and an easy-to-use interface to help them understand and communicate with the system about the expected behavior may be necessary.

7.3 Building an Intelligent Middleware

Based on the projects and future opportunities described in this thesis, we propose building an intelligent middleware to incorporate all the essential components to achieve these optimizations — a middleware that is able to optimize the application by analyzing both the application logic on the client, the server and data processing logic on the backend. It will not only include the optimizations and interactions described in earlier chapters, but also expose analysis frameworks that enables easy integration of more optimizations. It hides the details of implementations of data processing and the implementation of functionalities that needs to be implemented across multiple layers like data constraints. Meanwhile, it provides an interface for developers to easily understand the effect of how the changes they make affect the application behavior and performance.

Furthermore, such middleware also needs to adapt to runtime changes. For instance, it may dynamically build a cache to store frequently-used application data and re-route queries to be answered using the cache. It may also dynamically determine whether to execute a computation on the server or pushed down to database; and adapts the queries and subsequent data processing according to workload change. In sum, it is challenging to design a middleware that is efficient, extendable to many future optimizations while still mostly transparent with an efficient interface to learn insights from developers.

7.4 Extending Techniques to Other Systems

Many approaches introduced in this thesis can be extended to other systems as well. For instance, CHESTNUT can be extend to customize data layout to serve visualization systems. The data can be pre-processed and stored in a different layout like nested layout in order to accelerate commonly-used visualizations (e.g., group data and show the statistics of each group). The visualization workload may be different from the object-oriented application workload, which may require a different layout specification and query plan. CHESTNUT can also be extended to building query engine for non-relational data. Because nowadays a large amount of raw data is stored in format like JSON or XML, it is very time consuming to convert such data into relational format prior processing. One potential solution is to customize query processing directly on the raw data, where the query-plan-generation algorithm in CHESTNUT can be leveraged.

Another example is to leverage QURO's analysis to further improve the locking strategy of transaction processing systems. For instance, if multiple hotspots are often accessed at the same time, the database can grab a "hyperlock" on for all the hotspots to avoid potential aborts; the transaction may release lock early before commit while relying on a "remedy" to fix incorrect data read where the fix is synthesized based on application semantics, etc.

Chapter 8

RELATED WORK

This thesis presents many new techniques that enables much further performance improvement leveraging application semantics. In this chapter I will discuss prior work in database optimizations related to our approaches, studies in ORM applications that our work is based on, as well as other research in leveraging application semantics for performance improvement.

8.1 Related Database Optimizations

8.1.1 Improving transaction processing performance

This line of work is related to QURO, all targeting at transaction applications. Much work has been done to improve the locking-based transaction system performance. Previous work has explored mechanisms that violate 2PL to improve transaction performance. In distributed systems, Jones et al. [118] proposed a locking scheme that allows dependent transactions to execute when one transaction is waiting for its execution to finish in multiple partitions. The technique avoids making transactions wait for earlier transactions' distributed coordination. Gupta et al. [109] proposed a distributed commit protocol where transactions are permitted to read the uncommitted writes of transactions in the prepare phase of two-phase commit, for a similar purpose of avoiding expensive coordination before data can be visible to other transactions. Other work proposed non-locking concurrency control protocols that can read uncommited data. For instance, Faleiro et al. [100] proposed a protocol for deterministic databases that enables early write visibility, meaning that a transaction's writes are visible prior to the end of the execution. This protocol leverages the determinism where transaction execution is ordered prior to execution and the writes can be immediately visible to later transactions if the writing transaction will certainly commit. While aiming to reduce lock range at high contention similarly as QURO, this work targets at the protocol

itself and often requires constraints in the application (e.g., knowing read/write set earlier) or has mechanisms to mitigate the violation, and QURO uses the application semantic to achieve similar effect with fewer constraint and less overhead.

Besides lock-based methods, various concurrency control mechanisms have been proposed, such as optimistic concurrency control(OCC) [123] and multi-version concurrency control(MVCC) [79]. Yu et al. [158] studied the performance and scalability of different concurrency control scheme for main-memory DBMS. In this thesis, we compare the performance of 2PL with QURO-generated implementation to OCC and MVCC, and shows that it beats other concurrency control schemes by as much as 2× under high contention.

There has been work done on improving the efficiency of locking-based concurrency control schemes. Shore-MT [117] applies 2PL and provides many system-level optimization to achieve high scalability on multi-core machines. Jung et al. [119] implemented a lock manager in MySQL and improves scalability by enabling lock allocation and deallocation in bulk. Horikawa [112] adapted a latch-free data structure in PostgreSQL and implemented a latch-free lock manager.

8.1.2 Data layout and physical design

This line of work is related to CHESTNUT, similarly targeting at changing data layout to optimize queries. There is much prior work on automatic data layout design. One line of work explores ways to store data rather than the traditional row or column-major stores for each single table. For instance, Hyrise [108] and H2O [74] store columns together as column groups instead of individually. Widetable [124] uses a denormalized schema to accelerate OLAP queries. ReCache [76] uses dynamic caching to store data in tabular and nested layout to accommodate heterogeneous data source like CSV and JSON data. While prior work focuses on a restricted set of layouts, CHESTNUT integrally explores tabular and nested data layout together with auxiliary indexes, and also how objects are nested within each other, which field and subset of objects to store, etc. Such aspects are important for OODAs as our experiments show. CHESTNUT also synthesizes query plans from the generated data layout instead of relying on standard relational query optimizer. Doing so allows CHESTNUT's plans to better utilize the new data layout.

Another line of work focuses on learning the best data structure. For instance, Idreos et al. [113] design a key-value store that learns the configuration parameters to each or combinations of elementary data structures given query patterns. While reducing query execution time, such work focuses on queries with simple patterns on a single table. Queries in OODAs, which CHESTNUT optimizes, have complex query patterns involving many classes, and that leads to new challenges in deciding how to store objects of different classes and picking the optimal design across multiple queries.

Another approach is to use program synthesis [126, 125] to generate data structures from high-level specifications. Cozy finds a data structure that answers one query with minimum query execution time. While prior work optimizes only a single query without trading off update and memory, CHESTNUT handles workloads with many read and write queries subject to a memory bound and also determines how to share data structures among queries.

Automated physical design for relational databases is a well-studied topic. AutoAdmin [71, 88, 73, 85] finds best indexes and materialized views by trying different possibilities and asking the query optimizer for the cost. To reduce the number of optimizer calls, it uses heuristics to consider only a small number of candidates. In contrast, CHESTNUT employs a different approach that opens the "black box" of the query optimizer and uses program synthesis to enumerate query plans instead of "what-if" calls.

CHESTNUT is not the only work that leverages ILP solver for physical design. Cophy [98] and CORADD [121] use ILP solvers to find the best indexes and materialized views to add within a memory bound. NoSE [130] uses ILP to discover column families for NoSQL queries. Bruno et al. [84] use ILP to find the best sort order and pre-joined views for column store, and BIGSUBS [115] uses ILP to select common subexpressions that benefit the workload the most if materialized. These projects show that ILP is an attractive way to solve physical design problems. However, as CHESTNUT searches not just over relational (1NF) designs but also non-relational ones like nested objects and partial indexes, its formulation is different from prior work due to the nature of the problem.

Machine learning models are also used for physical design. Prior work [127, 132, 122] uses

reinforcement learning to decide on join order. They reuse the query optimizer and replace heuristics with machine learning models to utilize historical query performance to fine-tune plan choices. CHESTNUT replaces heuristics with efficient plan enumeration of the search space. It can find better plans as optimizers often do not have comprehensive rewriting rules.

8.1.3 Object-oriented databases

This line of work is related to CHESTNUT, where the query results are returned using object-oriented model. CHESTNUT leverages data structure concepts from object-oriented database systems [75, 120], such as the nested object model. Previous physical designers for OODBs focus on devising a language to describe the physical design [107, 102] or a single storage design that aims to optimize all OODAs [15]. Instead, CHESTNUT tailors the storage design to each application. CHESTNUT also proposes a language for storing objects and for query plan, but uses it for verification purposes.

Other NoSQL stores like document databases also store data in non-tabular format. However, the data model of document-oriented DBMS is different from OODAs. Document database's data model fixes the way how data are nested as defined by its JSON schema. Besides, document databases often support only limited query APIs, unlike Rails. These aspects make document databases unsuitable for OODAs given the complexity of such applications' query needs.

8.1.4 Leveraging database constraint

This line of work is related to CONSTROPT. Prior work has investigated verifying database-related constraints. ADSL [82] verifies data-model related invariants (e.g., whether each todo object is associated with a project object) using first order logic, while the invariants are provided by users using their invariant language. Singh and Wang [141, 148] check whether a set of DB constraints still hold when DB schema evolves while Caruccio [87] conducts a survey of related work in this domain. Pan [134] proposes a method to leverage symbolic execution to synthesize a database to verify different types of constraints like query construction constraints, DB schema constraints,

query-result-manipulation constraints, etc.

Another line of work focuses on using constraints provided by the DB or application for application verification and synthesis, like verifying the equivalence of two SQL queries[97, 146, 96], DB applications [147], synthesizing a new DB program with a new scheme given the original program with an old scheme [148], and handling chains of interactive actions [99].

There is also previous research looking into how to leverage functional dependency, a type of data constraint, to improve query performance. For instance, [135] studies how to use functional dependency to generate better query plans. However, most of the work are focusing on theoretical aspects, showing proofs about what optimization is correct. In contrast, CONSTROPT studies this topic in real-world applications. It discovers that many useful constraints are not defined in the database and should analyze the application to detect the constraints. Furthermore, some optimizations mentioned in the previous paper are found to be rarely exists or not practical in real-world applications. CONSTROPT also builds a system that automatically detects and implements the optimizations.

8.2 Performance Studies in ORM Applications

Previous work has confirmed that performance bugs are prevalent in open-source C/Java programs and often take developers longer time to fix than other types of bugs [114, 159]. Prior work [139] studied the performance issues in JavaScript projects. Previous work has also addressed specific performance problems in ORM applications, such as locating unneeded column data retrieval [92], N+1 query [90], etc.

8.3 Leveraging Application Semantics to Place Computation

There has also been work done in looking into improving database performance from the database application's perspective.

Previous work has been looking into improving database performance from the database application's perspective. DBridge [89, 110] is a program analysis and transformation tool that optimize database application performance by query rewriting. Such work detects patterns for potential batch

query processing and rewrite applications to enable the batch. QBS [95] detects the application code that processes the data in a way that can be pushed to the database and rewrites them as database queries. Sloth [94] and Pyxis [93] are tools that use program analysis to reduce the network communication between the application and DBMS. Most previous work focuses on the interface of between the application and the database: deciding where to place the data processing and reduce the communication between the application and database. The projects discussed in this thesis focus on an orthogonal direction which decides how the data processing is done. These projects leverage the application semantics to change more than the interface, but also the data layout, the query optimizer and the front-end as well. Furthermore, the thesis also discuss how to leverage the developer's insight to make performance tradeoffs, which has not been studied in previous work.

Chapter 9

CONCLUSION

Performance is critical for modern database-backed applications. These applications are often structured in three tiers, the front-end client, the application server and the database. These tiers are often developed using different languages, for instance, the front-end developed with HTML or JavaScript, the application server with object-oriented programming languages like Ruby, Python or Java, and the database processes SQL queries. As performance is critical for these applications because the end-to-end latency directly affects user experience, developers often put a lot of effort into performance tuning. Previous techniques have been looking into universal optimizations to improve the performance of each single tier. However, applying these techniques are often not enough to satisfy the latency requirement for many applications.

In this dissertation, I present a new way to optimize these applications through a series of projects. Rather than looking at individual tier, I show how leveraging the information from other tiers or even the developers can help improve application performance substantially. Opportunities for such optimization exists in every tier which I show through a number of projects optimizing each tier. At the database tier, I present CHESTNUT in Ch. 2 that optimizes the data storage according to how the application consumes the query result. Next I present CONSTROPT in Ch. 3 that leverages the data constraints inferred from how the application server processes the data to optimize database queries. At the application server tier, I present QURO in Ch. 4 that changes the application code to issue query in a different order such that the database transaction performance is improved. Then I present POWERSTATION in Ch. 5 which changes the queries issued by the application, replacing them with more efficient queries without breaking the application semantics. I further discuss how to leverage the developer's insight to make performance trade-off such that some parts of application which the developer cares the most are carefully optimized at the cost of slowing down

other non-important parts. In the HYPERLOOP project, presented in Ch. 6, I introduce an interface that allows the developers to interact with the system only at visual level (i.e., through the browser) to provide their insight while the system is able to leverage such insight to make trade-off at all tiers. There are more opportunities than those discussed in the above projects, and I briefly mention a few in Sec. 7. Evaluation shows that these optimizations can substantially improve the application performance, speeding up queries up to 43× and the entire webpage up to 5.5×.

Despite the various opportunities and great benefits, existing tools fail to explore them due to the inability to perform cross-tier analysis. In this thesis, I show how we solve this challenge by building a static analysis framework that performs cross-module and cross-language analysis with code refactoring. This framework serves as the basis for the tools we build in each project. It is able to perform cross-stack dependency analysis, for instance, how the view rendering in HTML uses the data processed by the application server and how such data is retrieved from the database. This framework is customized in each project to achieve certain goals, which reveals more challenges in such cross-stack analysis. In CONSTROPT, we customize the analysis to detect data constraints and represents the constraints in a grammar that can be leveraged in query rewrite verification. In QURO, we customize the data flow analysis to enable detecting query-level data dependencies in addition to program-variable level dependencies. In HYPERLOOP, we add a cost analysis on top of program flow analysis to attribute the data processing cost to every HTML tag such that developers can easily understand the cost by viewing the HTML-tag-heatmap alone. With these projects as evidence, I believe that such cross-tier analysis can be easily extend to support future optimizations.

Besides static analysis challenges, there are also many runtime challenges brought by cross-tier optimization. Very often the runtime interaction between tiers are changed due to the optimization and the system needs to be re-designed. In this thesis, I discuss what changes are required to support each optimization. CHESTNUT replaces in-memory query processing with CHESTNUT-generated layout and query plan while the backend database serves only as a persistent data storage. In CONSTROPT, the query rewrites are performed during static analysis while the replacement of optimized query happens at runtime in order to remove the non-trivial overhead of dynamic verification. In QURO, the transactions need to be executed and profiled before QURO's analysis to

understand the hotspot access in every transaction. These system changes vary for each optimization, which makes us wonder whether we can re-design the system to easily incorporate these changes as well as being compatible for future optimizations. In sum, these optimizations discussed in this thesis represents initial approaches to cross-tier optimizations. They open up future research towards building cross-stack analysis tool and re-designing the architecture for database-backed applications, while serving as inspirations for optimizing other systems and applications as well.

BIBLIOGRAPHY

[1] Active record association. https://guides.rubyonrails.org/association_basics.html.

[2] Active record query interface. https://guides.rubyonrails.org/active_record_querying.html.

[3] Airbnb. An online marketplace and hospitality service application. https://www.airbnb.com/.

[4] *Association Define in DataMapper*. https://www.rubydoc.info/github/datamapper/dm-core/DataMapper/Associations/Relationship.

[5] *Association Define in Django*. https://docs.djangoproject.com/en/3.0/ref/models/relations/.

[6] *Association Define in Entity Framework*. https://docs.microsoft.com/en-us/ef/ef6/fundamentals/relationships.

[7] *Association Define in Hibernate*. https://docs.jboss.org/hibernate/core/3.3/reference/en/html/associations.html.

[8] *Association Define in SQLAlchemy*. https://docs.sqlalchemy.org/en/13/orm/basic_relationships.html.

[9] The bidding benchmark from silo. https://github.com/stephentu/silo.git.

[10] Blog: Orm is an offensive anti-pattern. https://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html.

[11] Blog: Slow performance problems in django. https://medium.com/@hansonkd/performance-problems-in-the-django-orm-1f62b3d04785.

[12] Blog: Why orm shouldn't be your best bet. https://medium.com/ameykpatil/why-orm-shouldnt-be-your-best-bet-fffb66314b1b.

[13] Clang. http://clang.llvm.org.

[14] Common length of a comment. https://www.reddit.com/r/dataisbeautiful/comments/2mkp6u/comment_length_by_subreddit_oc/.

[15] Comparing oodb with LINQ. https://msdn.microsoft.com/en-us/library/aa479863.aspx.

[16] Curated list of awesome rails lists. https://project-awesome.org/ekremkaraca/awesome-rails.

[17] Database size reported by web developers. https://meta.discourse.org/t/slow-sql-queries/16604.

[18] Database size reported by web developers. http://www.redmine.org/issues/23318.

[19] dbt2 benchmark tool. http://osdldbt.sourceforge.net/#dbt2.

[20] dbt5 benchmark tool. http://osdldbt.sourceforge.net/#dbt5.

[21] A designer's guide to fast websites and perceived performance. https://www.sitepoint.com/a-designers-guide-to-fast-websites-and-perceived-performance/.

[22] Dexter, an automatic indexer for postgres. https://github.com/ankane/dexter.

[23] *Discourse*. A platform for community discussion https://github.com/discourse/discourse.

[24] Discussion on blog length. https://www.snapagency.com/blog/whatll-best-length-blog-article-2015-seo/.

[25] *Disjunctive subquery optimization*. https://nenadnoveljic.com/blog/disjunctive-subquery-optimization/.

[26] Django. https://www.djangoproject.com/.

[27] *Django field definition and options*. https://docs.djangoproject.com/en/3.0/ref/models/fields/#field-options.

[28] *Django polymorphic association API*. https://docs.djangoproject.com/en/3.0/ref/contrib/contenttypes/#generic-relations.

[29] *Django validation API*. https://docs.djangoproject.com/en/3.0/ref/models/fields/#validators.

[30] *Enumeration type in MySQL*. https://dev.mysql.com/doc/refman/8.0/en/enum.html.

[31] *Enumeration type in PostgreSQL*. https://www.postgresql.org/docs/9.1/datatype-enum.html.

[32] filter, validation and association. http://guides.rubyonrails.org.

[33] Find your new favorite web framework. https://hotframeworks.com/.

[34] Gitlab: a collaboration management website. https://github.com/gitlabhq/gitlabhq.

[35] Gurobi ilp solver. http://www.gurobi.com/.

[36] Hibernate. http://hibernate.org/.

[37] *Hibernate validation API*. https://hibernate.org/validator/.

[38] Huginn, an agent monitor. https://github.com/huginn/huginn.

[39] Hulu. A subscription video on demand service application. https://www.hulu.com/.

[40] Kandan. http://getkandan.com/. a chatting room website.

[41] Lobsters. https://github.com/jcs/lobsters. a discussion forum for developers.

[42] lpsolve. http://sourceforge.net/projects/lpsolve.

[43] Nested set model, a model to represent tree using relational table. https://en.wikipedia.org/wiki/Nested_set_model.

[44] *Onebody*. A social networking website to help management for churches. https://github.com/seven1m/onebody.

[45] Openstreetmap. https://github.com/openstreetmap/openstreetmap-website. a website for map service.

[46] Powerstation static analysis framework. https://github.com/hyperloop-rails/powerstation/.

[47] Protocol buffer, a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers.

[48] Publify. https://github.com/publify/publify. a blogging website.

[49] Rails api. https://api.rubyonrails.org/.

[50] *Rails query interface*. https://guides.rubyonrails.org/active_record_querying.html.

[51] *Rails state machine library*. https://github.com/aasm/aasm.

[52] *Rails state machine library*. http://github.com/pluginaweek/state_machine.

[53] ranking of popular open source rails applications. http://www.opensourcerails.com/.

[54] Redmine. https://github.com/redmine/redmine. a collaboration management website.

[55] Ruby on rails, a ruby web application framework. https://rubyonrails.org/.

[56] Ruby on rails association api. https://guides.rubyonrails.org/association_basics.html.

[57] Ruby on rails validation api. https://guides.rubyonrails.org/active_record_validations.html.

[58] Rubymine: The ruby on rails ide by jetbrains. https://www.jetbrains.com/ruby/.

[59] *Spree*. An ecommerce application. https://github.com/spree/spree/.

[60] Sql check constraint. https://www.w3schools.com/sql/sql_check.asp.

[61] Stx-btree library. https://github.com/bingmann/stx-btree.

[62] Sugar. https://github.com/elektronaut/sugar. a forum webiste.

[63] top7 forceful rails apps for business management. https://www.cleveroad.com/blog/effective-ruby-on-rails-open-source-apps-to-help-your-business.

[64] Tracks. https://github.com/TracksApp/tracks. a task management website.

[65] Update query cache. http://instantbadger.blogspot.com/2009/12/memcached-cache-invalidation-made-easy.html.

[66] Webpate loading time statistics. https://neilpatel.com/blog/loading-time/.

[67] Website response times. https://www.nngroup.com/articles/website-response-times/.

[68] who uses huginn. https://github.com/huginn/huginn/wiki/Companies-and-People-Using-Huginn.

[69] who uses redmine. http://www.redmine.org/projects/redmine/wiki/weareusingredmine.

[70] Z3 theorem prover. https://github.com/Z3Prover/z3.

[71] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[72] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[73] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[74] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, pages 1103–1114, 2014.

[75] Malcolm Atkinson, David DeWitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. *The object-oriented database system manifesto*. Elsevier, 1990.

[76] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *PVLDB*, 11(3):324–337, 2017.

[77] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, page 37–52, 2017.

[78] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. In *IEEE Trans. Softw. Eng.*, volume 5, pages 203–216, May 1979.

[79] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. In *ACM Comput. Surv.*, volume 13, pages 185–221, June 1981.

[80] José A. Blakeley, William J. McKenna, and Goetz Graefe. Experiences building the open oodb query optimizer. In *SIGMOD*, pages 287–296, 1993.

[81] Nikita Bobrov, Anastasia Birillo, and George Chernishev. A survey of database dependency concepts. In *Second Conference on Software Engineering and Information Management (SEIM-2017)*, page 43, 2017.

[82] Ivan Bocić, Tevfik Bultan, and Nicolás Rosner. Inductive verification of data model invariants in web applications using first-order logic. *Automated Software Engineering*, 26(2):379–416, 2019.

[83] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting scalar sql queries in the presence of disjunction. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 46–55. IEEE, 2007.

[84] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, 2005.

[85] Nicolas Bruno and Surajit Chaudhuri. Constrained physical design tuning. *PVLDB*, 19(1):4–15, 2008.

[86] C Robert Carlson, Adarsh K Arora, and Miroslava Milosavljevic Carlson. The application of functional dependency theory to relational databases. *The Computer Journal*, 25(1):68–73, 1982.

[87] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. Synchronization of queries and views upon schema evolutions: A survey. *ACM Transactions on Database Systems (TODS)*, 41(2):9, 2016.

[88] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997.

[89] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. DBridge: A program rewrite tool for set-oriented query execution. In *ICDE*, pages 1284–1287, 2014.

[90] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, pages 1001–1012, 2014.

[91] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, 2016.

[92] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. In *ICSE*, pages 1148–1161, 2016.

[93] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. Automatic partitioning of database applications. In *SIGMOD*, volume 5, pages 1471–1482, 2012.

[94] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *SIGMOD*, pages 931–942, 2014.

[95] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.

[96] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[97] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524, 2017.

[98] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011.

[99] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2005.

[100] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5):613–624, 2017.

[101] Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 159–170, 2008.

[102] Leonidas Fegaras and David Maier. An algebraic framework for physical OODB design. In *DBLP*, pages 6–8, 1995.

[103] Sheldon Finkelstein. Common expression analysis in database applications. In *SIGMOD*, pages 235–245, 1982.

[104] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, page 1859–1866, 2009.

[105] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: Killing one thousand queries with one stone. In *VLDB*, pages 526–537, 2012.

[106] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *USENIX*, pages 213–231, 2018.

[107] Dieter Gluche and Marc Scholl. Physical design in OODBMS. In *Grundlagen von Datenbanken*, pages 21–25, 1996.

[108] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. Hyrise: A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[109] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. In *SIGMOD*, page 486–497, 1997.

[110] Ravindra Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. In *VLDB*, pages 1107–1123, 2008.

[111] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[112] Takashi Horikawa. Latch-free data structures for DBMS: Design, implementation, and evaluation. In *SIGMOD*, pages 409–420, 2013.

[113] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[114] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[115] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[116] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Improving OLTP scalability using speculative lock inheritance. In *VLDB*, volume 2, pages 479–489, August 2009.

[117] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *EDBT*, 2009.

[118] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, page 603–614, 2010.

[119] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, pages 73–84, 2013.

[120] Won Kim and Frederick H Lochovsky. *Object-oriented concepts, databases, and applications*. ACM Press/Addison-Wesley Publishing Co., 1989.

[121] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Coradd: Correlation aware database designer for materialized views and indexes. *PVLDB*, 3(1):1103–1113, 2010.

[122] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[123] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. In *ACM Trans. Database Syst.*, 1981.

[124] Yinan Li and Jignesh M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[125] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. Generalized data structure synthesis. In *ICSE*, 2018.

[126] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *PLDI*, pages 355–368, 2016.

[127] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR*, 2019.

[128] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. In *Acta Inf.*, pages 121–163, 1990.

[129] Michael Meier, Michael Schmidt, Fang Wei, and Georg Lausen. Semantic query optimization in the presence of types. *Journal of Computer and System Sciences*, 79(6):937–957, 2013.

[130] Michael J. Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu. Nose: Schema design for nosql applications. In *ICDE*, pages 181–192, 2016.

[131] Stephen O'Grady. The redmonk programming language rankings: June 2017. `http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/`.

[132] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM*, pages 4:1–4:4, 2018.

[133] Kai Pan, Xintao Wu, and Tao Xie. Generating program inputs for database application testing. In *ASE*, pages 73–82, 2011.

[134] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):12, 2014.

[135] Glenn Norman Paulley. *Exploiting functional dependence in query optimization*. Citeseer, 2001.

[136] D.E. Porter and E. Witchel. Understanding transactional memory performance. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 97–108, March 2010.

[137] Joeri Rammelaere and Floris Geerts. Revisiting conditional functional dependency discovery: Splitting the "c" from the "fd". In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 552–568. Springer, 2018.

[138] Mingwei Samuel, Cong Yan, and Alvin Cheung. Demonstration of chestnut: An in-memory data layout designer for database applications. In *SIGMOD*, 2020.

[139] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: an empirical study. In *ICSE*, pages 61–72, 2016.

[140] Maya Carrillo Selfa, Diana M. and M. Del Rocio Boone. A database and web application based on mvc architecture. In *CONIELECOMP*, 2006.

[141] Rohit Singh, Vamsi Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Generating concise entity matching rules. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1635–1638. ACM, 2017.

[142] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. Tesma and catg: Automated test generation tools for models of enterprise applications. In *ICSE*, pages 717–720, 2015.

[143] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[144] Transaction Processing Performance Council. The TPC-H benchmark. `http://www.tpc.org/information/benchmarks.asp`, 1999.

[145] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11. Technical report, 2010.

[146] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Speeding up symbolic reasoning for relational queries. *PACMPL*, 2(OOPSLA):157:1–157:25, 2018.

[147] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. In *Proceedings of the ACM on Programming Languages*, pages 56:1–56:29, 2017.

[148] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300. ACM, 2019.

[149] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[150] Cong Yan and Alvin Cheung. Leveraging lock contention to improve oltp application performance. 9(5):444–455, 2016.

[151] Cong Yan and Alvin Cheung. Generating application-specific data layouts for in-memory databases. 12(11):1513–1525, 2019.

[152] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *CIKM*, 2017.

[153] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. View-driven optimization of database-backed web applications. In *CIDR*, 2020.

[154] Junwen Yang, Utsav Sethi, Cong Yan, Shan Lu, and Alvin Cheung. Managing data constraints in database-backed web applications. In *ICSE*, 2020.

[155] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *ICSE*, pages 800–810, 2018.

[156] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. PowerStation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *ESEC/FSE*, pages 884–887, 2018.

[157] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. View-centric performance optimization for database-backed web applications. In *Proceedings of the 41st International Conference on Software Engineering*, pages 994–1004, 2019.

[158] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *VLDB*, volume 8, pages 209–220, November 2014.

[159] Bram Adams Zaman, Shahed and Ahmed E. Hassan. A qualitative study on performance bugs. In *MSR*, pages 199–208, 2012.