Leveraging Application Data Constraints to Optimize Database-Backed Web Applications

Xiaoxuan Liu
University of California, Berkeley
xiaoxuan_liu@berkeley.edu

Shuxian Wang
University of California, Berkeley
wsx@berkeley.edu

Mengzhu Sun
University of California, Berkeley
zoey_sun@berkeley.edu

Sicheng Pan
University of California, Berkeley
pansicheng@berkeley.edu

Ge Li
University of California, Berkeley
geli2001@berkeley.edu

Siddharth Jha
University of California, Berkeley
sidjha@berkeley.edu

Cong Yan
Microsoft Research
Cong.Yan@microsoft.com

Junwen Yang
Meta
junweny@meta.com

Shan Lu
University of Chicago
shanlu@uchicago.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

ABSTRACT

Exploiting the relationships among data is a classical query optimization technique. As persistent data is increasingly being created and maintained programmatically, prior work that infers data relationships from data statistics misses an important opportunity. We present Coco, the first tool that identifies data relationships by analyzing database-backed applications. Once identified, Coco leverages the constraints to optimize the application's physical design and query execution. Instead of developing a fixed set of predefined rewriting rules, Coco employs an enumerate-test-verify technique to automatically exploit the discovered data constraints to improve query execution. Each resulting rewrite is provably equivalent to the original query. Using 14 real-world web applications, our experiments show that Coco can discover numerous data constraints from code analysis and improve real-world application performance significantly.

PVLDB Reference Format:
Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. PVLDB, 16(6): 1208 - 1221, 2023.
doi:10.14778/3583140.3583141

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hyperloop-rails/Coco.git.

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583141

1 INTRODUCTION

From key constraints to the uniqueness of data values, relationships among attributes in a dataset are bases for relational query optimization. These *data constraints* occur in datasets across many different application domains [52, 65], and have been used in optimization that ranges from normalizing relational schemas [62], detecting data errors [64], to leveraging functional dependencies to improve query execution [60].

There has been a long line of research that applies statistical [48] and machine learning [45] techniques to identify data constraints from persistently stored data. While such data-driven approaches have been effective in discovering constraints from *collected* datasets (such as those from census or physical experiments), we are unaware of techniques that target *programmatically-generated* datasets.

We encounter programmatically-generated datasets routinely in our daily lives—all websites process user inputs via web applications that generate persistently stored data. While there are means to express data constraints for programmatically-generated data, such as SQL constraints [43] and various *data validation* APIs provided by web application frameworks [8, 24], they all require developers to manually declare them in their applications, which has shown to be tedious and error-prone to developers due to their complexity [65]. As the artifacts that are used to generate or manipulate such datasets are often available (e.g., web application code, synthetic data generators), relying on data analysis techniques to "re-discover" constraints by analyzing the stored data while ignoring the programmatic artifacts is simply a missed opportunity.

In this paper, we investigate the feasibility of discovering data constraints by analyzing the programs that generate and process persistent data. We focus on database-backed web applications, given their prevalence and the public availability of artifacts. We study how such applications are developed and designed Coco, the first tool that automatically analyzes database-backed web applications to discover different data constraints in the stored data. Given

the source code of a web application, Coco statically analyzes the code to extract candidate constraints.

We demonstrate the usefulness of the discovered constraints by using them to optimize application performance. Coco leverages the extracted constraints to change the data schema to reduce storage, add parameter precheck to avoid issuing queries, and rewrite queries to improve their performance. To optimize query performance, we first install the extracted constraints into the database. However, as discussed in Sec. 6.3, mainstream commercial and open-source databases fail to utilize the constraints as they rely on pattern-matching rules to rewrite queries [7, 12, 36, 46, 61], and those rules are often application-specific. While we can extend existing optimizers with additional rules, doing so takes substantial effort[1] and only benefit few applications as rules are overly specific.

Moreover, blindly applying such rules can degrade query performance [39]. Coco instead uses an enumerate-test-verify approach, where a number of rewrites are first *enumerated* for each query based on various query features (e.g., whether it joins one of the tables with identified constraints). Coco then estimates the cost of the rewritten query using the database's optimizer. If the cost is less, it verifies that the rewritten and original queries are semantically equivalent with test cases and a formal verifier.

In sum, this paper makes the following contributions:

- Data constraints are often embedded in artifacts that generate and process persistent data programmatically. To our knowledge, this is the first work that discovers data constraints from application source code and uses them for query optimization.
- We use extracted constraints to automatically rewrite queries, optimize application code, and change the physical design (Sec. 6.2). To rewrite queries, rather than crafting a priori query transformation rules, we enumerate candidate rewrites and use formal verification to identify equivalent and efficient ones (Sec. 6.3).
- Evaluation of 14 popular open-source web applications shows that Coco can extract 4039 constraints (averaging 289 per application). We evaluate Coco's optimization on 6 such applications. Among queries with constraints, 13.8% queries can benefit from data layout optimization, and 47% queries are optimized by changing application code. Finally, Coco 's constraint-driven optimizer improves the performance of 2511 queries, 118 of which have over 2× speedup.

2 BACKGROUND

2.1 Structure of ORM applications

Applications built with the object-relational mapping (ORM) framework are structured using the model-view-controller (MVC) architecture. For example, when a web user submits a form through a URL `http://foo.com/wikis/id=1`, a *controller* action `wikis/id` is triggered. This action takes the parameters from the HTTP request (e.g., "1" in the URL) and interacts with the database via the ORM's API (e.g., ActiveRecord). The ORM translates its function calls (e.g., `Wiki.where(id=1)`) into SQL queries (e.g., a select query to retrieve wiki record with `id=1`), with results then serialized into *model* objects (e.g., `Wikis`) and returned to the controller. The returned objects are then passed to the *view* files to generate a web page returned to users.

2.2 Associations among classes and tables

ORM frameworks provide an object-oriented interface to manage persistent data, where each class or class hierarchy is mapped to database table(s). To support inheritance in relational databases, ORMs either use one table to store data for all types under the same inheritance hierarchy (also called the "Table Per Hierarchy" approach), or store each type in its separate table (i.e., "Table Per Type"). Table Per Hierarchy results in one table storing all entities in the inheritance hierarchy. The table includes a "discriminator" column, which stores the actual type for each row. Furthermore, developers can define relationships between classes to connect them, such as `belongs_to`, `has_one`, and `has_many`. Once defined, an object can simply retrieve its relevant objects of different classes through its fields without writing joins.

2.3 Data constraints in ORM applications

Data constraints are rules enforced on stored data. Including class relationships, there are three ways to express data constraints:

Front-end constraints. Developers can check for user input on the client side (e.g., logins must be longer than 6 characters) and returns an error if the check fails without contacting the server.

Application constraints. The application code running on the server can also contain constraints. For instance, it can validate data values before inserting them into the database and only persist data if validation passes. As we will discuss in Sec. 4, a variety of constraints can be expressed in the application code.

Database constraints. Developers can also declare constraints in the database, such as primary and foreign keys, uniqueness, value nullness, and string length constraints.

As shown in previous work [65], the first category contains very few constraints, while the latter two cover more than 99% of all constraints. In this paper, we focus on application constraints, and describe how they can be generalized into several common patterns and automatically discovered. As we will see in Sec. 7, *most* of the inferred constraints are not declared in the database, hence they are not leveraged by the database during query optimization.

2.4 ORM frameworks

ORMs provide a high-level API over a relational database. This allows developers to manipulate persistent data and its schemas using the programming language they are familiar with rather than SQL. When executed, ORMs automatically translate each API call into SQL queries. All ORMs that we are aware of translate such calls straightforwardly and leave query optimization to the underlying database. This makes sense as each ORM typically supports multiple databases[2] and is thus difficult for it to include optimizations that are compatible with all databases. Meanwhile, as shown in Table 2, none of the popular open-source or commercial databases supports all types of the rewrite optimization provided by Coco because many of them are application-specific and they take substantial effort to implement without using Coco's "enumerate-test-verify"

[1] It took more than two years for PostgreSQL developers to implement a pattern that removes the `DISTINCT` clause if the result is unique by definition, and this feature is yet to be merged [26].

[2] Rails for instance supports SQLite, MySQL, PostgreSQL. [20]

approach. While Coco 's functionalities can be implemented in the ORM or the database, we implement our prototype as an independent component as doing so demonstrates that our technique is agnostic to any specific ORM or database implementation.

3 OVERVIEW

We first give an overview of using Coco on an example abridged from Redmine [25], a popular collaboration web application built using Rails. Redmine defines a `User` class to manage user information and a `Project` class to store project details. The `Member` class keeps track of each user's membership information (for example, one user can be a member of many projects). Rails stores user, member, and project information in separate database tables, and developers manipulate the stored data by calling Rails' functions. Listing 1 shows the definition of the `Member` class on lines 1-4, and the code to create and save a `Member` object on lines 6-7. Line 4 utilizes Rails' built-in validation API and is called whenever the object is saved to the database, as shown in line 7. Rails executes the validation on Line 4 by executing a query to determine if a user with the same project already exists in the member table and raises an error if so. Here, the validation function implicitly defines a data constraint that *given a project, the users belonging to the project are unique*. Yet, it is only defined in the application code but not specified as part of the database schema, as developers can write arbitrary code in the validation function, and not all of them can be easily translated to SQL constraints.

```
1 # Member Class definition
2 class Member
3     belongs_to :user, :project
4     validates_uniqueness_of :user_id, :scope => :project_id
5 # Create a Member object and save it to the database
6 member = Member.new(user_id=1, project_id=2)
7 member.save
```

Listing 1: Redmine code with an implicit data constraint.

Once the uniqueness constraint is discovered, Coco leverages it to improve application performance. For example, Listing 2 shows a Redmine query that selects all the active users working on a given project. This query has a `DISTINCT` keyword to filter out duplicate users. But, given the constraint mentioned, users working on the same project are guaranteed to be unique, hence there is no need to run duplicate elimination. With 100K records in the users tables, this query can be accelerated by 1.65× by removing the `DISTINCT` keyword. However, we are unaware of any mainstream query optimizer that would perform such optimization as the optimizer is unaware of such constraints. As we will show, many similar constraints that manipulate persistent data are "hidden" in applications, and we are unaware of any existing tools that can discover them. Moreover, as we will show in Sec. 7, even if we install such "hidden" constraints into the database, the optimizer still fails to leverage them to optimize queries as traditional DBMS uses heuristics to rewrite queries, and they cannot cover all possible optimizations, such as the one shown here.

```
1 SELECT DISTINCT users.* FROM users
2 INNER JOIN members ON members.user_id = users.id
3 WHERE users.status = 'active' AND (members.project_id = 2)
```

Listing 2: Query issued by Redmine.

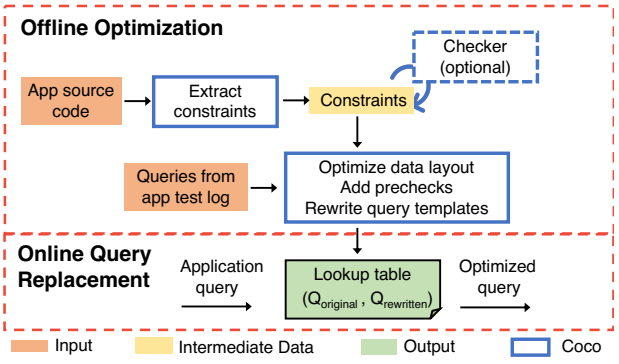


Figure 1: Coco Architecture.

```
1 SELECT COUNT(*) FROM "users" WHERE "users"."type" IN ($1, $2)  [[
      "User"], ["AnonymousUser"]]
```

Listing 3: Each log record consists of a query template and its parameter values. Here $1 and $2 are the query parameters with values "`User`" and "`AnonymousUser`" respectively.

Coco is built to bridge the gap between application and database. It analyzes application source code to automatically extract data constraints. Moreover, Coco automates the challenging process of using constraints to optimize queries. Even the most sophisticated commercial and open source databases, as shown in Table 2, do not support all of the rewrite types offered by Coco. Coco optimizes queries using the two components as shown in Figure 1.

- **Offline Optimization.** Coco first statically analyzes the application source code to extract constraints. It detects constraints by matching code patterns defined in Sec. 4. The extracted constraints are valid by construction, but Coco also generates a checker program for constraint validation against the stored data in case the application violates our assumptions stated in Sec. 5.1. Coco then extracts the queries that the application might issue by analyzing the logs generated by running application tests. Listing 3 shows an example log record with two parts: query template and parameters. Coco extracts the query templates and uses extracted constraints to optimize query performance by:
  - *Optimizing data layout (Sec. 6.2).* If a string column has a limited set of possible values, i.e., there is an inclusion constraint on the column, Coco changes the data type from string to `enum` to save the storage and speed up queries.[3]
  - *Adding prechecks on query parameters (Sec. 6.2).* Coco utilizes format and length constraints to optimize application logic by adding prechecks on user inputs to avoid issuing queries if the input violates length or format requirements. Coco takes constraints and application source code offline and emits an optimized version with precheck logic.
  - *Rewriting queries (Sec. 6.3).* Coco rewrites query templates to improve query performance using the extracted constraints. To rewrite the query templates, Coco enumerates feasible

[3] In cases where modifying the database schema directly is undesirable (e.g., breaking other applications that access the same database), Coco can also generate a SQL DDL script for the database administrator to determine when to apply the modifications.

rewrites (e.g., remove DISTINCT) based on the extracted constraints. It then filters out slow rewrites based on the estimated cost. Coco then uses test cases and a formal verifier to ensure that the rewritten query templates are semantically equivalent to the original. At the end of this offline step, Coco creates a lookup table comprising of the original and optimized query template pairs, and the table is used to rewrite queries online as the application runs.

- **Online query replacement.** As the application runs, Coco intercepts all queries issued by the application. If the query's template exists in Coco's lookup table, Coco issues the optimized query template with its parameters to the database. Otherwise, the query is issued as-is.

We now discuss different types of data constraints inherent in the application code and how Coco detects them. Then we introduce the optimizations with these constraints, followed by an evaluation using real-world database-backed ORM applications.

4 DETECTING CONSTRAINTS

Coco extracts both application constraints and database constraints automatically from the source code. Application constraints are embedded semantically in the application code when developers define the model class. Database constraints are specified explicitly in the migration files [21], which are used to alter database schema over time. Constraints defined in migration files will later be installed into the database as database constraints.

Because of the flexibility, convenience of use, and capacity to manage errors, many constraints are written in the application rather than the database, as indicated in Sec. 7.2. Defining constraints in the application code is more flexible as the constraint type is not limited, and developers can write complex logic to express constraints. Meanwhile, constraints not supported by the database must be expressed as user-defined functions (UDFs), which are typically written in SQL and are tedious to use for complex logic. As shown in Sec. 4.1, ORM frameworks also have a number of simple built-in APIs to express common constraints. Finally, developers can associate meaningful error messages when constraints are violated. Whereas the database only throws low-level errors that are rarely caught by the developers [65]. Hence, once triggered, the web user's session will most likely crash, with all the filled-in contents lost with a cryptic SQL error.

Coco uses both the database constraints and application constraints to optimize the application. Coco works by parsing the app source code, building an abstract syntax tree (AST) for each file, and pattern-matching on the AST nodes. If the current node contains any of the patterns shown in Table 1 and [54], Coco will continue to visit its children. Therefore, if the application source code has n tokens, the complexity of pattern matching is $O(n)$.

Coco extracts the following types of constraints:

- **Inclusion**: the field value is restricted to a limited set.
- **Presence**: the field value cannot be null. This is the same as SQL `NOT NULL` constraint but is implicitly defined in the application.
- **Length**: the length of a string field should be in a certain range.
- **Uniqueness**: same as the SQL uniqueness constraint, but is only defined in the application.
- **Format**: the value of a string field must match a regular expression, which is specified in the application code.
- **Numerical**: the value of a numerical field must lie within the range specified in the application code.
- **Foreign key**: same as the SQL foreign key constraint, where the field points to the primary key of the referenced table.

For each constraint, we first describe how it conceptually arises from application code, followed by an example, and then the general code pattern that Coco uses for extraction.

4.1 Data validation

Data validation is the most important method for extracting the application constraints. It's an important feature of ORM-based apps since it ensures that only valid data is kept in the database. Similar to SQL triggers, most ORMs provide callback mechanisms (e.g., validation functions provided by Rails [24], Django validators[8], Hibernate validators [34], etc.) where a function is triggered automatically every time before data is saved to the database. The callback can be one of the ORM's built-in functions that capture common attributes to validate, or one of the user's own validation procedures. Inside the callback, a built-in or customized property is checked on the data to be saved, and the function returns an error without saving the data if the check fails. Consequently, such checks lead to constraints that must be satisfied by all stored data.

As an example, Listing 4 shows two types of validation callbacks from OpenProject [18]. First, in line 3, a Rails built-in validation function, `validate_format_of`, is used to check if the specified field `email` satisfies the regular expression in Line 4. Then, in line 7, a custom validation function `validate_name`, defined in lines 8–9, is registered to ensure the length of the `name` field does not exceed 30 characters, implying a data constraint that `length(name)`$\leq$`30`.

```
1 class User < ApplicationRecord
2   # built-in validation
3   validates_format_of :email, :with =>
4   "/\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\Z/i"
5   # custom validation
6   validate :validate_name
7   def validate_name
8     if length(name) > 30
9       errors.add(:name, "is too long (maximum is 30 characters)")
```

Listing 4: Validation function excerpt from OpenProject.

To extract constraints from validation functions, Coco identifies uses of built-in and custom validations registered with `validate`. As each built-in validation checks for a particular data property, such as `validates_length_of` checks for the length of passed-in fields, we define a constraint template for each built-in validation as shown in Table 1, and use the template to generate a constraint by analyzing the passed parameter(s).

To handle custom validations, Coco walks through the parsed AST of the function body to identify the branch condition leading to an `error` statement. It then matches the branch condition with pre-defined patterns as detailed in [54], and uses the negation of the condition as a constraint if the pattern-match succeeds. For instance, in Listing 4, Coco identifies that the condition `name.length > 30` leads to an error statement shown in Line 10. The branch condition is matched as `api_call(field) > constant` from the grammar. It then derives a constraint on the `User` table as the negation of the condition, i.e., `!(name.length > 30)`.

Table 1: Rails built-in data validation patterns. In Django [8] and Hibernate [34], all patterns are implemented similarly.

Built-in Validation		
Code Pattern	**Category**	**Constraint Description**
`validates_inclusion_of:` *field*, in *value_list*	Inclusion	*field* takes value from *value_list*
`validates_presence_of:` *field*	Presence	*field* is not `NULL`
`validates_uniqueness_of:` *field*, `scope:` *scope_field* | *scope_field_list*	Uniqueness	*(field,scope_field)* | *[field]* + *scope_field_list* is unique
`validates_length_of | validates_size_of:` *field*, `minimum =>` *value*, `maximum =>` *value*, `in | within` *value_range*	Length	*field* has type string, and its length is within the given range
`validates_format_of:` *field*, `:with =>` *regex*	Format	*field* matches the format specified by the *regex*
`validates_numericality_of:` *field*, `greater_than:` *value*, `greater_than_or_equal_to:` *value*, `equal_to:` *value*, `less_than:` *value*, `less_than_or_equal_to:` *value*	Numerical	*field*'s numerical value matches the condition specified by the comparison keywords

4.2 Class relationships

ORM frameworks support complex relationships between classes such as class hierarchies, polymorphic one-to-many, etc. The framework maintains those relationships, which can be found as constraints on persistent data. We list the full patterns in [54].

4.2.1 Type hierarchy. As discussed in Sec. 2.2, ORMs use different mechanisms to support inheritance in relational databases, such as one table per entire class hierarchy, or one table per class. Rails [22] (and similarly Django [1]) by default employs the table per hierarchy mechanism, which uses a separate string field to record the classname to distinguish between various class instances. This field is called `type` by default, or can be explicitly defined by the user in the `inheritance_column`. Similar to the example in Listing 5, where both `Firm` and `Client` inherit from `Company`. Rails keeps a single `companies` table to store instances of both `Firm` and `Client`, with the `type` field of this table indicating which class the record belongs to. This mechanism introduces an inclusion constraint, where the value of the `type` field can only be `Firm` or `Client`.

```
1 class Company < ActiveRecord::Base;
2 class Firm < Company;
3 class Client < Company;
```

Listing 5: Example of type inheritance, with bodies of the class definitions omitted due to space.

To extract class inheritance constraints, Coco maps each class to its inherited classes by searching for class definitions in the code. Coco then generates inclusion constraints for each entry in the mapping where the key is the table and column name, and the value is the value range of the inclusion constraint. For instance, in the case shown in Listing 5, Coco will detect that `Company.type` can only have values `'Firm'` or `'Client'`.

4.2.2 Polymorphic definitions. Similar to `type` fields in class inheritance, ORM frameworks allow developers to define a field that refers to the primary key of multiple tables, and uses an extra string field to identify which table the record belongs to. An example is shown in Listing 6: each `Organization` and `User` object contains a single `Address`, and hence an `Address` object can belong to either an `Organization` or a `User`. This polymorphic relationship is declared in line 2, where each `Address` is set to belong to an `addressable` interface, and the `Organization` and `User` classes are declared as `addressable` in lines 5 and 8.

Internally, since all `Address` records are stored in a single `address` table, Rails adds an integer field `addressable_id` that refers to a primary key in either the `organizations` or `users` table, and a string field `addressable_type` to store the type name of the object that an `address` belongs to. This mechanism allows for inclusion constraints where `addressable_type` can only take the value of `'Organization'` or `'User'`. Similar to the type hierarchy, Coco automatically analyzes and infers such constraints.

Coco also extracts polymorphic definitions by building a mapping from the polymorphic class name and the interface name (e.g., `(Address, addressable)`) to a list of classes that use the interface (e.g., `['Organization', 'User']`)

```
1 class Address < ActiveRecord::base
2   belongs_to :addressable, polymorphic: true
3
4 class Organization < ActiveRecord::base
5   has_one :addresses, as: :addressable
6
7 class User < ActiveRecord::base
8   has_one :addresses, as: :addressable
```

Listing 6: Example of a polymorphic interface definition.

4.2.3 has_one association. ORMs provide `has_one` associations to express that exactly one other class has a reference to a class object. Listing 7 shows an example, where line 2 declares a `WikiPage` object belonging to a `Project` object, while each `Project` has only one `WikiPage` (line 5). With the `belongs_to` association, a foreign key field `project_id` is added to the `wikipages` table. This implies that the constraint `project_id` is unique across the `wikipage` table.

```
1 class WikiPage < ActiveRecord::base
2     belongs_to project, class_name: 'Project'
3
4 class Project < ActiveRecord::base
5     has_one wikipage, class_name: 'WikiPage'
```

Listing 7: Example of data association declaration.

Coco extracts such associations by matching the `has_one` keyword for each class, and records its `class_name` and the associated class. A unique constraint is then generated on the `class_name_id` field of the table identified by the associated class name.

4.3 Field definition with state machines

Instead of issuing an `UPDATE` query, applications often use libraries to change the value of a field. For instance, the `state_machine` library is commonly used to define how the value of a field can be

changed [23]. An example is shown in Listing 8 from Spree [30], where changing the `state` field is done by a `state_machine`: as shown in line 2, the value of `state` can be changed from `'checking'`, `'pending'`, `'complete'` to `'processing'` only when the `start_processing` event is triggered. Using `state_machine` to update `state` implies a constraint where the value of `state` can only be one of the string literals defined within the state machine.

```
1 class Payment
2   state_machine : state, initial: 'checkout' do
3     event :start_processing do
4       transition from:['checkout','pending','complete'],
5             to: 'processing'
6     event :failure do
7       transition from:['pending', 'processing'], to:'failed'
```

Listing 8: Example of field definition from Spree.

Coco extracts state machine constraints by first identifying `state_machine` library calls. It goes through each of the event functions and extracts all parameters in the `from:` and `to:` expressions to obtain potential state values. If all the values are string literals, Coco will generate a corresponding inclusion constraint. In using state machines, the state variable only changes when the specified event takes place [31]. Thus, all possible states must appear in the same `state_machine` construct, as shown in Listing 8.

5 VALIDITY OF OUR APPROACH

We first describe the requirements for applications to use Coco. Next, we list the assumptions Coco makes, and how Coco handles application changes.

5.1 Requirements and Assumptions

Coco extracts constraints with pattern matching on the validation APIs defined by the ORM. While Coco currently focuses on Rails, the validation APIs defined by other ORMs such as Hibernate and Django are essentially identical to Rails'. Therefore, a large number of ORM applications should readily benefit from Coco.

Moreover, to guarantee the correctness of extracted constraints, we make the following assumptions about ORM applications:

- Data validation, as discussed in Sec. 4, is not bypassed. Even though developers can skip validation when saving an object by setting the validate parameter to false (e.g., `obj.save(validate: false)`, such behavior violates the design principle of validation and is highly discouraged [33]. We scan the code of all 14 applications used in the evaluation and only find two cases where the developer skips the validation. Those two cases update only fields that are irrelevant to validation, and therefore do not affect the validity of the extracted constraints.
- The application does not use any non-analyzable third-party library that breaks existing constraints. We assume that the application code is analyzable, as we cannot guarantee constraint validity if a third-party library whose source code is not available modifies the fields involved in the constraints. However, this case is rare and we did not find any in the evaluated applications.

Given these assumptions, Coco will generate valid constraints. While we focus on the database being accessed by a single application, if there are multiple applications accessing the same database, Coco will ensure that the same constraint holds for *all* applications before extracting it as detailed in [54].

5.2 Code upgrade

Developers can run Coco to re-detect constraints when application code is updated. As shown in Sec. 7.3, re-extracting constraints is fast and efficient. Moreover, incremental constraint detection can be performed by scanning only modified files to further reduce the constraint extraction time. Note that code changes might add new constraints or alter previously extracted constraints so that new constraints are incompatible with old data. As shown in [65], the mismatch between constraints and data is problematic and developers should ensure that data integrity is preserved. Multiple methods have been proposed to fix this. For example, migration files can be used to ensure that data in the database satisfy the newly inferred constraints. Coco's checker program described in Sec. 5.3 can also help developers identify such data integrity problems and ensure the validity of extracted constraints.

5.3 External changes and constraints checker

In cases where the DB contents are not only modified by the application, such as when a DB administrator manually changes the contents of the database via a command line interface, the constraints detected by Coco may become invalid. Although such behavior is discouraged and rare, Coco comes with a checker that examines if constraints still hold after a third party modifies the database. The checker can also be used to validate the extracted constraints under the two assumptions described in Sec. 5.1. For each extracted constraint, Coco generates a Ruby script that scans all data in the database and checks whether the extracted constraints are valid. The checker script can be run against a concrete database instance and remove any Coco-extracted constraints that are no longer valid. In general, we expect such cases to be rare and leave the decision of when to run the checker script to the developer.

6 QUERY ANALYSIS AND OPTIMIZATION

We now discuss how Coco rewrites queries with extracted constraints. We summarize the constraints used in different optimization in Table 3, and discuss how Coco leverages them to optimize queries by changing the source code or database schema and rewriting queries with our enumerate-test-verify approach.

6.1 Query extraction

Coco extracts SQL queries that can be issued by the application by running its provided test cases and analyzing the log file that records all SQL queries executed. Coco replaces the issued query template (as discussed in Sec. 3) with the optimized version if it exists in the lookup table. As the application runs, queries are issued against different query templates after pairing them with parameters. Therefore, Coco can still apply the optimization as long as the query template exists in its lookup table. Furthermore, as tests are carefully written with a high level of code coverage (over 92% for the applications used in our evaluation), Coco should have analyzed most templates that the application can possibly issue when deployed.

6.2 Optimizing code logic and physical design

Coco leverages the extracted constraints to optimize performance by rewriting application code and changing the data type of the underlying storage. We describe both below.

Adding prechecks to avoid issuing queries. A query can be completely removed if it returns empty results. For example, for a query containing a predicate that compares a field with user input, we can add a precheck in the application code to issue the query only if the input matches the field's associated constraints. Listing 9 shows an example from Dev.to [3], where Coco adds precheck on the string field `username`. Here Coco extracts a constraint that all characters in `username` must be digits, letters, or underscores, as expressed by matching the regular expression in line 1. For the query that retrieves users given username (line 2) , we add a precheck to the parameter (line 4) that avoids issuing the query altogether if the parameter contains invalid characters and hence is impossible to match a username in the database.

```
1 + if param[:username].match(/\A[a-zA-Z0-9_]+\z/)
2       user = User.where(username: param[:username])
3 + else
4 +     user = nil
```

Listing 9: Adding parameter check example from Dev.to [3]. + indicates code added by Coco.

To implement this, Coco checks each query Q for the presence of the predicate `TableName.field=`*param*. Using the data flow graph, Coco traces if *param* is computed from the user input and if there is a constraint associated with `field`. If Q satisfies these criteria, Coco adds the precheck as shown in Listing 9.

Altering database schema. If a string field has an inclusion constraint (i.e., its value can only come from a limited set of literals), Coco changes its physical design and replaces the field with an enumeration type. Since `enum` comparison is faster than string comparison, changing the storage can reduce the time to process predicates on that field. Additionally, it improves space efficiency, as the `enum` type is stored using only four bytes in the database [9, 10].

To implement this, Coco checks extracted inclusion constraints and changes the type of field `f` to enumeration type if there is an inclusion constraint on the field. Coco creates an `enum` datatype based on the inclusion constraints. It generates `ALTER TABLE` statements that change `f`'s type from `varchar` to `enum` based on the value list of the inclusion constraint. Coco can execute those statements directly or return them to the DB administrators to determine when to apply them (e.g., when there are no updates to the affected columns). Queries involving field `f` will remain unchanged as the database will perform type conversion when running the query.

Notice that the above-mentioned optimizations can only be performed by Coco: the database is unaware if a query parameter is derived from user inputs and is unable to add prechecks on input columns. Moreover, the database is also unaware of inclusion constraints and thus cannot alter its schema automatically. Using Coco, however, a large number of queries can benefit from the two optimizations, as we will discuss in Sec. 7.3.

6.3 Rewriting queries

The previous two optimizations do not need to modify the query, however, many queries do require exploiting extracted constraints to rewrite the query to a semantically equivalent but more efficient one. In Table 3, we outline the taxonomy of using constraints for query optimizations in Coco.

Table 2: Rewrite types supported by mainstream DBMS.

DBMS	Remove Distinct	Add Limit One	Predicate Elimination Introduction	Join Elimination Introduction	Detect Empty Set
PostgresSQL 10.7	×	✓	×	×	×
MySQL 8.0.2	×	✓	×	×	×
SQL Server 2019	×	×	×	✓	×
DB2 10.5 Kepler	×	×	✓	✓	×
Coco	✓	✓	✓	✓	✓

We first install the Coco-extracted constraints into the database and see if the database can leverage them to optimize queries. However, as shown in Sec. 7.4, only a few queries can benefit even after the constraints are in place. Most major database management systems, as shown in Table 2, are unable to leverage Coco-extracted constraints to optimize queries. The reason is that existing DBMS relies on heuristics to exploit constraints and rewrite queries. Each type of rewrite necessitates different rules, and some of which are rather complex. For instance, to remove the `DISTINCT` keyword in a query after identifying a uniqueness constraint, the optimizer must track the uniqueness of all the columns used in every operator, since not all query operators preserve the uniqueness of the input (e.g., projection) [26]. Implementing such rules requires substantial effort, yet it is unclear how generally applicable they are. For instance, in the Redmine, only 399 out of 2283 queries use `DISTINCT`, among which only 67 can be removed.

Algorithm 1 Rewrite generation, test, and verification algorithm

Input: Original query template Q, and all constraints C.
Output: Equivalent rewrite R with the minimum cost.

```
// Step 1: enumerating potential rewrites.
fields = get_used_fields(Q)
CQ = get_constraints_on_fields(fields, C)
rewrite_types = get_constraint_rewrites(CQ)
R_enumerate = {Q}
for rt ∈ rewrite_types do
    R_t = { } // rewrites after applying rt
    for candidate ∈ R_enumerate do
        R_t += apply_transformation(rt, candidate)
    R_enumerate += R_t
// Step 2: Cost estimation and testing rewrites.
R_cost = { }
for candidate ∈ R_enumerate do
    if cost(instantiate(candidate)) < cost(instantiate(Q)) then
        R_cost.add(candidate)
R_test = { }
for candidate ∈ R_cost do
    if test_eq_on_test(instantiate(candidate), instantiate(Q)) then
        R_test.add(candidate)
// Sort rewrites based on the cost in ascending order
R_test_sort = R_test.sort(key = cost, asc=True)
// Step 3: formally verifying rewrite equivalence.
for candidate ∈ R_test_sort do
    if verify_eq(candidate, Q) then
        Return candidate // Early stop
return NULL // fail to find an optimized rewrite
```

Coco instead uses an enumerate-test-verify approach as shown in Algorithm 1 to automate the rewrite process. First, Coco enumerates all possible transformations of a query template given the rewrites shown in Table 3. To reduce the number of potential rewrites, only those query templates containing fields with constraints are enumerated. Coco then puts the parameter values into the rewritten templates and uses the query optimizer to estimate the cost of each instantiated rewrite. Rewrites with a lower cost than the original are then checked for semantic equivalence. As formal verification can be costly, Coco instead generates a test database to execute each instantiated rewrite and the original query. Coco filters away those cases where the instantiated rewrite returns different results from the original. Coco then sorts the remaining rewrites based on their estimated costs in ascending order and sends them to Coco's formal verifier to check for query equivalence. Coco finally emits the rewritten template once it finds the first equivalent one. We now describe these steps in detail.

Step 1: Heuristic-guided rewrite. We describe the query optimizations with constraints that Coco leverages in Table 3. As described, each rewrite leverages certain types of constraints. Therefore, Coco enumerates potential rewrites only when the corresponding constraints exist. As shown in Algorithm 1 lines 2–10, Coco extracts all columns used in a query template and checks them against the extracted constraints. For each existing constraint, Coco applies the corresponding potential transformations as shown in Table 3. For instance, for the query in Listing 10, Coco extracts the used columns `members.user_id`, `users.id, users.status`, and `members.project_id`. Coco then determines that `users.id` and each pair of (`members.user_id`, `members.project_id`) is unique. It then applies the *Remove DISTINCT* and *Add LIMIT One* transformations and generates three candidate rewrites as shown in lines 14–16. Coco's modular design makes it easy to add new types of rewrite rules. Users can simply add a new semantic query rewrite rule to the search space by providing the utilized constraints and associated enumeration.

The candidate rewrites generated by the enumeration step are not guaranteed to be semantically equivalent to the original or to perform better, and that is the goal of Step 2.

```
1  -- Original Query
2  SELECT DISTINCT users.* from users
3  INNER JOIN members ON members.user_id = users.id
4  WHERE users.status = $1 AND (members.project_id = $2)
5
6  -- Used columns
7  members.user_id, users.id, users.status, members.project_id
8
9  -- Extracted constraints on used columns
10 Uniqueness: (members.user_id, members.project_id) pair is unique
11 Uniqueness: users.id is unique
12
13 -- Candidate rewritten templates:
14 1. SELECT users.* from users INNER JOIN members ON members.user_id =
       users.id WHERE users.status = $1 AND (members.project_id = $2)
       -- remove DISTINCT
15 2. SELECT DISTINCT users.* from users INNER JOIN members ON members.
       user_id = users.id WHERE users.status = $1 AND (members.
       project_id = $2) LIMIT 1  -- add LIMIT 1
16 3. SELECT users.* from users INNER JOIN members ON members.user_id =
       users.id WHERE users.status = $1 AND (members.project_id = $2)
       LIMIT 1  -- remove DISTINCT and add LIMIT 1
```

Listing 10: Heuristic-guided rewrite for a Redmine query.

Step 2: Cost estimation and check for rewrite equivalence using test database. For each enumerated candidate template, Coco first instantiates it by binding the parameter values as the original query. Next, Coco asks the database optimizer to estimate its cost, as shown in lines 13–16, and only retains it if it has a potentially lower cost than the original query. After filtering away the slow rewrites, Coco attempts to eliminate as many incorrect rewrites as possible to reduce the number of rewrites that need to be verified. To achieve this, Coco generates a synthetic database given the table schema and compares the outputs of the candidate and the original query. The templates of rewrites that produce the same outputs as the original query are sent to step 3 for verifying query equivalence formally.

Step 3: Formal verification of the candidate rewrites. As the last step, Coco calls its verifier to check the equivalence of the original query template and rewritten ones for all remaining candidates. The verifier is based on Cosette's U-semiring semantics [40–42] and implements the U-semiring decision procedure (UDP). Given a pair of query templates to check, UDP translates each SQL query template to an expression of U-semiring, rewrites into sum-product normal form (SPNF, as given in [40]), and finally attempts to unify both SPNFs. The UDP approach already models uniqueness, key constraints, and treats aggregations as uninterpreted functions. To model the `LIMIT 1` transformation, we introduce a new axiom for the `LIMIT` operator.

$$\sum_a R(a) = \left\| \sum_a R(a) \right\| \implies \text{Limit}(1, R) = R.$$

This captures the idea that when there are no more than one row in R, we can turn $\text{Limit}(1, R)$ into R.

We model constraints on columns by adding a predicate over columns, which will be injected as additional predicates in the U-semiring expressions. For example, the numerical constraint of less than 100 for some column k in table R is handled by the solver with the rewrite $R(k) = [k < 100] \times R(k)$ after normalization. The verifier then calls an SMT solver [29] to check for logical equivalence between the predicates during the unification of SPNFs.

Data Generation. We implemented a data generator to create synthetic data. Data generation is done in three phases. First, Coco creates a graph of class relationships where the vertices represent Rails models, and the edges denote the dependencies between models (e.g., foreign key). Coco then runs a topological sort on the graph to determine the order to populate test data. Coco then generates a Ruby script for each Rails model that inserts data into the database when executed. Finally, Coco executes each generated Ruby file based on the topological sort order from the first phase. Because we insert data through the Rails API, all generated data will be validated before inserting it into the database.

7 EVALUATION

7.1 Experiment setup

Application corpus. We select 14 real-world web applications built using the Ruby on Rails framework [28] that cover 5 categories: forum, collaboration, e-commerce, social network, and map application. We select the most popular, actively maintained applications from each category based on the number of GitHub stars as

Table 3: Description of rewrite types, the constraints that trigger the rewrite, and the implementation.

Rewrite Type	Constraints	Enumeration implementation	When is the enumeration correct and why is it beneficial
Remove DISTINCT [47]	Uniqueness	Remove the `DISTINCT` keyword if there is any uniqueness constraint on any of the used columns in the query.	The selection result is known to be unique. It is advantageous because the `DISTINCT` operator sorts or aggregates data, both of which are expensive.
Add LIMIT One [66]	Uniqueness	Add `LIMIT 1` to the end of the query and any subquery if there is any uniqueness constraint on any used columns in the query.	If no more than one record is selected, add `LIMIT 1` to avoid unnecessary operations (e.g., scan) after finding one satisfying record. This is useful if there is no index on the unique column, and a sequential scan must be performed to find the matching result.
Predicate Elimination [38, 39]	Numerical or Presence	Remove the predicate if there are any numerical or presence constraints on the fields included in the predicate.	The predicate is known to be always true given the constraints. This is beneficial because of avoiding unnecessary predicate comparison.
Predicate Introduction [38, 39, 63]	Numerical	Use a solver [29] to enumerate all non-redundant formulas derivable from the predicates and numerical constraints, and add them to the predicate if there is a numerical constraint on any of the used columns in the query.	The added predicated is guaranteed to be true. A new predicate on an indexed attribute may allow for a more efficient access method. Similarly, a new predicate on a join attribute may reduce the number of join records, thus improving the join performance.
Join Elimination [38, 39, 63]	Foreign Key and Presence	Enumerate all possible ways to drop the join table and the join conditions if there is any foreign key constraint on any used columns in the query.	A join may be constrained such that its result is known a priori and does not need to be evaluated. For example, queries that join two tables are related through a referential integrity constraint.
Detecting the Empty Answer Set [38, 39]	Numerical or Presence	Modify the predicate to `False` if there are any numerical or presence constraints on the fields included in the predicate.	If the query predicates are inconsistent with the integrity constraints, the query result is always empty, and we can avoid issuing the query completely.

Table 4: Details of the applications chosen in our study. Fields show the number of average fields across all tables.

Category	Abbr.	Name	Stars	Tables	Fields
Forum	Ds	Discourse[5]	30.8k	180	9
	Dv	Dev.to[3]	2.1k	92	13
	Lm	Loomio[14]	1.9k	50	19
	Lb	Lobsters[13]	3.0k	15	4
Collaboration	Re	Redmine[25]	3.7k	54	8
	Gi	Gitlab[11]	22.2k	337	8
	Op	OpenProject[18]	1.2k	114	7
E-commerce	Ro	Ror ecommerce[27]	1.2k	65	6
	Sp	Spree[30]	11.4k	57	11
SocialNetwork	Da	Diaspora[4]	12.4k	50	7
	On	Onebody[17]	1.4k	57	11
	Ma	Mastodon[15]	3.6k	78	6
	Ta	Tracks[32]	1.0k	17	6
Map Applications	Os	OpenStreetmap[19]	2.1k	46	7

Table 5: Number of model constraints.

Application	Ds	Dv	Lm	Lb	Re	Gi	Op
Data Validation	167	395	61	109	223	853	164
Class Relations and Field Definition	285	111	64	48	89	525	103
Total	452	506	125	157	312	1378	267
Application	Sp	Ro	Da	On	Ma	Ta	Os
Data Validation	129	184	64	103	108	24	152
Class Relations and Field Definition	148	103	59	65	126	21	63
Total	277	287	123	168	234	45	215

listed in Table 4. All the applications have over 1K stars and have been developed for more than four years.

Evaluation platform and Synthetic dataset. We implement Coco using Ruby, Python, and Rust, where the Ruby code analyzes the application code to detect constraints, the Python code enumerates rewrites, and the Rust code formally verifies query equivalence. We use the AWS c5.4xlarge instance with 16 vCPUs and 32GB memory to measure query performance and use PostgresSQL 10.7 as the database for all apps. We use the tool described in Sec. 6.3 to generate data for evaluation and scale the application data size to be 5–10GB, with 10K to 1M records per table, which is close to the size of data reported by application developers [6].

7.2 Constraint detection

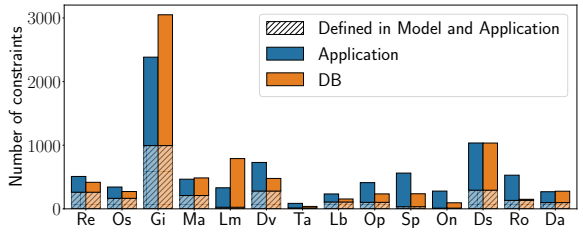
We report the total number of constraints detected by running Coco on each application in Table 5. Coco extracts an average of 289 constraints for each application.

Missing constraints and accuracy analysis. Coco extracts all constraints defined using built-in APIs and a subset of constraints defined in custom validators. As Coco is based on pattern matching, we are unable to enumerate every conceivable pattern defined in the user's custom validation code and thus can potentially miss constraints defined in custom validation functions. However, based on our observation of the 6 evaluated applications, as detailed in [54], over 85% validation functions are defined using built-in validations and Coco already extracts a large portion of total constraints.

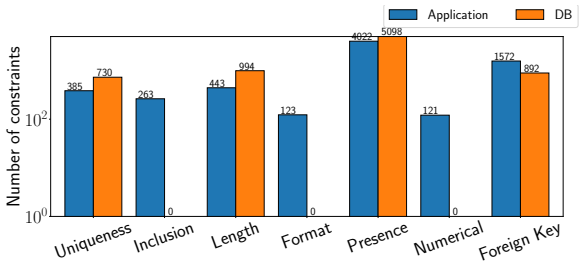
Comparison with database constraints. As described in Sec. 4, developers can also explicitly declare database constraints in migration files. Constraints defined this way are installed as database constraints. Comparing the constraints extracted from model files and those installed in the database as shown in Figure 2(a), there is a small percentage overlap which accounts for only 12% of the total number of constraints. This demonstrates Coco's ability in discovering latent constraints defined only in the application code.

Moreover, as shown in Figure 2(b), the constraint type determines if it is defined in the application or the database. For example, inclusion, format, and numerical constraints are only defined in the application, as defining them in the database requires writing UDFs or as `CHECK` constraints [2], which is tedious to implement.

Comparison with prior work. We compare the correctness of extracted constraints and the execution time between Coco and two prior data-driven algorithms [16] on our synthetic data. We run the HyUCC [59] discovery algorithm to detect unique columns



(a) Constraint defined in different applications



(b) Constraint type distribution. For both DB and application constraints, we sum the number across all 14 applications and group them by constraint type.

Figure 2: Application and DB constraint comparison.

Table 6: Compare with HyFD [58] and HyUCC [59]. ER = Error Rate. UCC = Unique Column Constraint. FD = Functional Dependency. Error rates for Coco are all 0% because UCCs/FDs extracted by Coco are valid by construction. We show the total number of UCCs extracted by Coco and HyUCC, and the overlap defines the UCCs extracted by both methods. Similarly, we show the total number of FDs extracted by Coco and HyFD, as well as the overlap.

App	Table name (size in MB)	Coco FD /HyFD (ER)/overlap	Coco UCC /HyUCC (ER)/overlap	HyFD/HyUCC/ Coco exec(s)
Dv	notes (312)	7 / 29 (79%) / 6	1 / 5 (80%) / 1	3.06/2.94/0.06
Re	journal_details (703)	5 / 19 (74%) / 5	1 / 4 (75%) / 1	6.64/5.93/0.03
Op	auth_sources (316)	34 / 15 (60%) / 10	2 / 5 (60%) / 2	2.80/2.94/0.04
Ma	users (73)	70 / 74 (73%) / 28	3 / 7 (57%) / 4	3.21/2.65/0.08
Os	users (111)	584 / 1007 (44%) / 551	5 / 34 (85%)/ 4	3.80/3.31/0.07
Sp	spree_assets (2.9)	15 / 15 (60%)	1 / 2 (50%) / 1	0.44/0.45/0.09

and HyFD [58] to detect functional dependencies on the biggest synthetic table from each evaluated application.

As shown in Table 6, although HyUCC generates a superset of the UCCs that Coco generates, most constraints that are only detected by HyUCC are incorrect. For instance, in Openstreetmap, HyUCC detects 34 unique columns, where only 5 of them are true keys when manually checking the application code. The other 29 detected columns appear to be unique only looking at the data. For example, the `creation_time` field of `User` is detected as a key as users are unlikely to be created at the same time. However, it is not a valid key as multiple users can theoretically be created concurrently. In contrast, Coco does not detect `creation_time` as a key. It correctly identifies 5 true keys without false positives. This illustrates a general issue with data-driven approaches to constraint discovery: while they can extract similar ones compared to Coco, the extracted constraints can be ephemeral due to the persistent data available at the time.

Table 7: Number of queries that can benefit from parameter precheck and changing data storage.

Application	Queries with constraints	Length Precheck	Format Precheck	Change Physical Design (String to Enum)
Dev.to	4738	676 (14.3%)	456 (9.6%)	430 (9.1%)
Redmine	3511	1270 (36.2%)	302 (8.6%)	877 (25.0%)
OpenProject	14845	6329 (42.6%)	502 (3.4%)	6373 (42.9%)
Mastodon	7059	5469 (77.5%)	5519 (78.2%)	19 (0.3%)
Openstreetmap	4889	555 (11.4%)	0 (0.0%)	72 (1.5%)
Spree	4261	15 (0.4%)	0 (0.0%)	185 (4.3%)

On extracting functional dependency constraints, We compare Coco with HyFD [58]. As shown in Table 6, Coco and HyFD have some overlaps on detected functional dependencies (FDs), but Coco can detect FDs that HyFD missed. HyFD also reports FDs that are not detected by Coco, but upon manual inspection, the majority of such constraints are false positives. HyFD, like HyUCC, recognizes any column related to time information as a part of functional dependencies. Also, because HyDB uses a fast approximation algorithm by only calculating from a subset of rows in the table, it falsely claims that many columns containing random values are unique (e.g., `encrypted_password`).

Finally, Coco runs much faster than HyUCC and HyFD as it only scans the application source code once with complexity as described in Sec. 4 to extract unique columns and function dependencies concurrently. On the other hand, HyUCC/HyFD performs extraction by analyzing the data, which scales with the data size and is typically much bigger than the source code.

7.3 Optimization opportunities

We next investigate Coco's ability to leverage constraints to optimize application performance on six applications, Redmine [25], Dev.to [3], OpenProject [18], Mastodon [15], OpenStreetmap [19], and Spree [30]. We list the number of queries that use a constrained column in Table 7. We first show the number of queries that can benefit from optimizing application code and physical design. We next measure the number of queries that can be rewritten using the Coco-detected constraints.

7.3.1 Precheck and optimizing physical design. As shown in Table 7, the number of applicable optimizations in each category is determined by the number of constraints and queries that can benefit from the optimization. For example, despite having only 6 format constraints and 13 length constraints on user information columns (e.g., username, domain name), many queries in Mastodon are optimized, as 75% of its queries search for username related records. On the other hand, as OpenStreetmap is a map application, its queries are almost exclusively about searching for location coordinates. Although there are only 3 format constraints and 26 length constraints, none of them are on coordinate related columns. Therefore, none of its queries benefit from the format precheck optimization. Similarly, as Spree does not contain any format constraints, none of its queries benefit from the format precheck optimization either.

7.3.2 Query Rewrite. **Rewrite count.** In Figure 3, we show the total number of queries rewritten by Coco, as well as the number

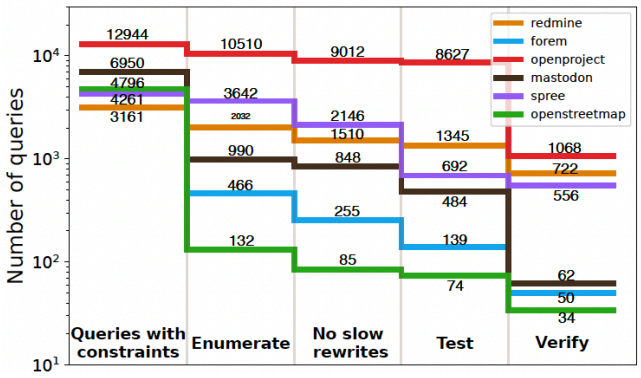


Figure 3: Number of queries after each rewrite step.

of rewrites after each step. To count the number of queries with constraints, we extract all fields utilized by a query template and check if any of them contains a constraint. By enumerating those queries, Coco generates candidate rewrites. The number of enumerated candidates varies across applications due to the varying styles of query templates used in each application. For example, in Dev.to [3], we cannot apply the remove `DISTINCT` optimization since none of the query templates has the `DISTINCT` keyword. Similarly, the estimating cost and testing steps are affected by query characteristics and vary by application. Since the first three steps filter away a considerable amount of the incorrect or slow rewrites, 1894 queries on average are finally sent to the verifier, which then proves that an average of over 415 queries can be optimized.

We examine the verified rewrites and find that some optimized queries benefit from a combination of optimizations. For instance, Coco performs two optimizations on the query shown in Listing 11 using two different constraints: the uniqueness leads to the removal of `DISTINCT`; and Coco changes the datatype of `users.type` from `varchar` to `enum` and performs selection on the transformed storage. Both optimizations lead to 2.4× speedup.

```
--Constraints
--1.(members.user_id, members.project_id) pair is unique
--2.users.type can only take values from ['User', AnonymousUser']
--Query before
SELECT DISTINCT users.* FROM users INNER JOIN members ON members.
      user_id = users.id WHERE users.status = $1 AND (members.
      project_id = $2) AND users.type IN ($3)
--Query After (user.type has been changed to enumeration type)
SELECT users.* FROM users INNER JOIN members ON members.user_id =
      users.id WHERE users.status = $1 AND (members.project_id = $2)
      AND users.type IN ($3)
```

Listing 11: Two optimizations applied to a Redmine query.

Execution time. The constraint extraction time for Redmine, OpenProject, Dev.to, Mastodon, Openstreetmap, and Spree are 1.30s, 1.41s, 0.55s, 0.49s, 0.62s, 0.27s, and 0.83s respectively. The average time to enumerate, remove slow rewrites, and run tests is less than 1s across all applications as detailed in [54]. Verification takes the longest time. However, even with Redmine and OpenProject, which have more complicated queries (with the longest query consisting of 1551 characters), the average verification time is still within 100s.

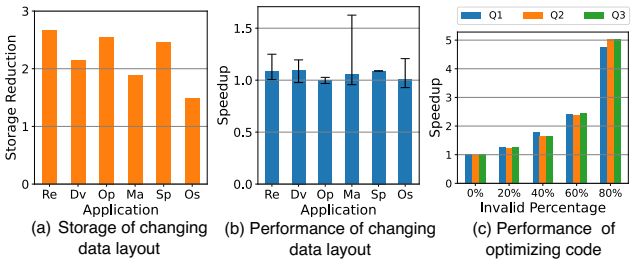


Figure 4: Evaluation of optimizing code and data layout.

7.4 Performance Evaluation

We first evaluate the storage reduction of changing data storage. For columns that can benefit from this optimization, assuming the candidate values are distributed evenly in our synthetic dataset, the average number of bytes for columns involved in the optimization are 10.70, 8.56, 10.21, 7.57, 5.97, and 9.80, respectively. After changing to `enum`, only 4 bytes are needed for each field, which reduced storage by 2.19× on average. We also observe minor performance improvement by changing data storage. As shown in Figure 4 (b), changing datatype from string to `enum` improves query performance by 1.05× across all applications. The speedups from changing the database schema are not very significant as the majority of queries involve many columns, and columns with inclusion constraints only make up a small part of all those that are queried.

To evaluate the benefits of adding prechecks on user inputs, we sample 3 queries and evaluate them under different ratios of invalid input. As shown in Figure 4 (c), the speedup consistently increases as the percentage of invalid input increases. When 80% of the input is invalid, we obtain the average speedup of 5×. Meanwhile, adding precheck introduces negligible overhead.

We next evaluate the benefits of utilizing constraints to improve query performance. For each rewritten query, we first record the query execution time with predefined database constraints as the baseline. The time spent only doing the rewrite, only installing application constraints, and both installing application constraints and performing query rewriting are then recorded. We issue each query 30 times and average the results. Note in Coco, the whole optimization process (constraint extraction, code optimization, data layout changes, and query rewrites) happens offline, as discussed in Sec. 3. Therefore, we only assess the query performance before and after rewrites to simulate the online setting. Figure 5 depicts the number of queries with various speed improvements.

```
--Constraints, address is unique
--Original Query, creating a unique index on address speedups 12.3x
SELECT email_addresses.* FROM email_addresses WHERE email_addresses.
      address = 'bcfy@yaho.com' or user_id = 10;
```

Listing 12: Example of database optimizations with constraints from Redmine.

As shown in Figure 5, the database only optimizes a small fraction of queries. The largest speedup comes from creating an index on the unique column once the uniqueness constraints are installed. As demonstrated in Listing 12, installing the uniqueness constraint on the `address` column creates an index, altering the query plan from a linear scan to an index lookup and getting 12.3× speedup.

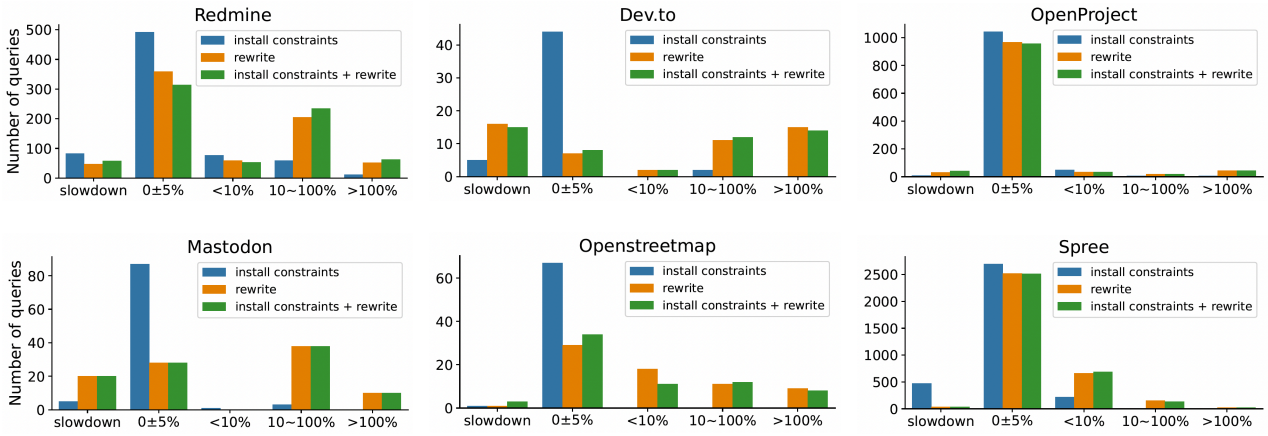


Figure 5: Performance improvement after installing extracted constraints and performing rewrites. Improvement = ((execution time with only database constraints / execution time after optimization) - 1)× 100%.

We also examine the performance of just rewriting queries without installing the application constraints as shown in Figure 5. The rewrite achieves comparable results as both installing and rewriting queries, which corresponds to our observation that the database underutilizes the installed constraints. Lastly, we install the extracted constraints, utilize Coco to rewrite queries, and evaluate query performance. Among queries with constraints, over 7.2% (52/722) for Redmine, 21.6% (11/51) for Dev.to, 2.9% (31/1086) for OpenProject, 12.9% (8/62) for Mastodon, 17.6% (6/34) for Openstreetmap, 1.8% (10/556) for Spree have a speedup of more than 2×. For some queries both installing constraints in the database and performing rewrites lead to more gains than only performing one of two. For example, in Listing 12, installing the uniqueness constraint on the `address` column causes an index to be created. Subsequently, rewriting the query by adding `LIMIT 1` further speeds it up by 1.4× as it allows early return after finding the first matching record using the index. Overall, the speedups mainly come from Coco 's ability to leverage constraints to simplify queries and save unnecessary computation to improve performance. For example, when the selection result is known to be unique, Coco removes the `DISTINCT` operator from the query to avoid expensive sort or aggregate operations.

We observe some slowdowns from the experiment and the causes are twofold. First, a few queries become slower after rewriting as the optimizer fails to predict the cost accurately, resulting in the rewritten query taking longer to execute. We believe the performance of those queries can still be improved with more accurate cost estimates. Second, many slowdowns are caused by our synthetically generated test dataset, which does not necessarily have the same data distribution as real-world data. This leads to some of the queries returning empty results (particularly for selections and joins), and any additional operation added as a result of Coco's rewrites causes a minor but obvious slowness. We believe our evaluation will be more accurate with real-world user data.

8 RELATED WORK

Constraint detection. Techniques have been proposed to discover constraints from data. [35, 48, 57, 67] extract dependencies by modeling the search space as a power set lattice of attribute combinations and traverses it, while [37, 49, 50, 53] automatically discover soft and hard functional dependencies for big data query optimization. Coco instead detects constraints by code analysis. Compared to data-driven methods, Coco scales well regardless of data size. It can also discover many other types of constraints in addition to functional and inclusion dependencies.

Leveraging database constraints for query optimization. Semantic query rewrite has been widely studied in the database literature. Some work provides theoretical results [38, 51, 56, 60, 63], while other work shows how this can be done in real systems [44, 55, 68]. All prior work uses heuristics [7, 12] to leverage constraints, the contribution of Coco lies in combining different heuristics and automating the optimization process.

9 CONCLUSION

We presented Coco, a tool that extracts data constraints from applications to optimize queries. Our experiments show that Coco can discover many constraints from real-world applications and speed up 118 queries across 6 evaluated applications by over 2×.

ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work is supported in part by NSF grants IIS-1955488, IIS-2027575, CNS-1764039, CCF-2119184, DOE award DE-SC0016260, ARO award W911NF2110339, and ONR award N00014-21-1-2724.

REFERENCES

[1] [n.d.]. Class inheritance in Django. https://docs.djangoproject.com/en/4.1/topics/db/models/#model-inheritance. Last accessed 18 February 2023..

[2] [n.d.]. Constraint type in PostgreSQL. https://www.postgresql.org/docs/13/ddl-constraints.html. Last accessed 18 February 2023..

[3] [n.d.]. Dev.to. An open source software for building communities. https://github.com/forem/forem. Last accessed 18 February 2023.

[4] [n.d.]. Diaspora. https://github.com/diaspora/diaspora. Last accessed 17 February 2023.

[5] [n.d.]. Discourse. https://github.com/discourse/discourse. Last accessed 18 February 2023..

[6] [n.d.]. Discourse Database size. https://meta.discourse.org/t/57598/. Last accessed 18 February 2023..

[7] [n.d.]. Disjunctive subquery optimization. https://nenadnoveljic.com/blog/disjunctive-subquery-optimization/. Last accessed 18 February 2023..

[8] [n.d.]. Django validation API. https://docs.djangoproject.com/en/4.1/ref/validators/. Last accessed 18 February 2023..

[9] [n.d.]. Enumeration type in MySQL. https://dev.mysql.com/doc/refman/8.0/en/enum.html. Last accessed 18 February 2023..

[10] [n.d.]. Enumeration type in PostgreSQL. https://www.postgresql.org/docs/9.1/datatype-enum.html. Last accessed 18 February 2023..

[11] [n.d.]. Gitlab. https://github.com/gitlabhq/gitlabhq. Last accessed 17 February 2023.

[12] [n.d.]. Join elimination. https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/. Last accessed 18 February 2023..

[13] [n.d.]. Lobsters. https://github.com/lobsters/lobsters. Last accessed 17 February 2023.

[14] [n.d.]. Loomio. https://github.com/loomio/loomio. Last accessed 17 February 2023.

[15] [n.d.]. Mastodon. https://github.com/mastodon/mastodon. Last accessed 17 February 2023.

[16] [n.d.]. Metanome. https://bit.ly/metanome. Last accessed 18 February 2023..

[17] [n.d.]. Onebody. A social networking website to help management for churches. https://github.com/seven1m/onebody. Last accessed 18 February 2023.

[18] [n.d.]. OpenProject, A web-based project management software. https://github.com/opf/openproject. Last accessed 18 February 2023..

[19] [n.d.]. OpenStreetMap. https://github.com/openstreetmap/openstreetmap-website. Last accessed 17 February 2023.

[20] [n.d.]. Rails Backend Database. https://guides.rubyonrails.org/v2.3/getting_started.html#configuring-a-database. Last accessed 18 February 2023..

[21] [n.d.]. Rails migration files. https://guides.rubyonrails.org/active_record_migrations.html. Last accessed 18 February 2023..

[22] [n.d.]. Rails single table inheritance. https://api.rubyonrails.org/classes/ActiveRecord/Inheritance.html. Last accessed 18 February 2023..

[23] [n.d.]. Rails state machine library. https://github.com/aasm/aasm. Last accessed 18 February 2023..

[24] [n.d.]. Rails validation API. https://guides.rubyonrails.org/active_record_validations.html. Last accessed 18 February 2023..

[25] [n.d.]. Redmine. A project management application. https://github.com/redmine/redmine. Last accessed 18 February 2023.

[26] [n.d.]. Remove Distinct in PostgreSQL. https://commitfest.postgresql.org/35/2433/. Last accessed 18 February 2023..

[27] [n.d.]. Ror ecommerce. https://github.com/drhenner/ror_ecommerce. Last accessed 17 February 2023.

[28] [n.d.]. Ruby on Rails, a ruby web application framework. https://rubyonrails.org/. Last accessed 18 February 2023..

[29] [n.d.]. Solver z3. https://pypi.org/project/z3-solver/. Last accessed 18 February 2023..

[30] [n.d.]. Spree. An ecommerce application. https://github.com/spree/spree/. Last accessed 18 February 2023.

[31] [n.d.]. State Machine API Document. https://www.rubydoc.info/github/pluginaweek/state_machine/StateMachine/MacroMethods. Last accessed 18 February 2023..

[32] [n.d.]. Tracks. https://github.com/TracksApp/tracks. Last accessed 17 February 2023.

[33] [n.d.]. Validation API Document. https://guides.rubyonrails.org/v3.2/active_record_validations_callbacks.html#skipping-callbacks. Last accessed 18 February 2023..

[34] [n.d.]. Validators in Hibernate. https://hibernate.org/validator/. Last accessed 18 February 2023..

[35] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. 2014. DFD: Efficient functional dependency discovery. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. 949–958.

[36] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[37] Paul G Brown and Peter J Haas. 2003. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings 2003 VLDB Conference*. Elsevier, 668–679.

[38] Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-Based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.* (1990), 162–207.

[39] Qi Cheng, Jarek Gryz, Fred Koo, TY Cliff Leung, Linqi Liu, Xiaoyan Qian, and Bernhard Schiefer. 1999. Implementation of two semantic query optimization techniques in DB2 universal database. In *VLDB*, Vol. 99. Citeseer, 687–698.

[40] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (2018), 1482–1495.

[41] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.

[42] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. In *PLDI*. 510–524.

[43] Roberta Cochrane, Hamid Pirahesh, and Nelson Mattos. 1996. Integrating triggers and declarative constraints in SQL database systems. In *VLDB*, Vol. 96. Citeseer, 3–6.

[44] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *VLDB*. 330–341.

[45] Luc De Raedt, Andrea Passerini, and Stefano Teso. 2018. Learning constraints from examples. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[46] Beatrice Finance and Georges Gardarin. 1991. A rule-based query rewriter in an extensible dbms. In *Proceedings. Seventh International Conference on Data Engineering*. IEEE Computer Society, 248–249.

[47] Jean Habimana. 2015. Query optimization techniques-tips for writing efficient and faster SQL queries. *International Journal of Scientific & Technology Research* 4, 10 (2015), 22–26.

[48] Yka Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal* 42, 2 (1999), 100–111.

[49] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 647–658.

[50] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B Zdonik. 2009. Correlation maps: a compressed access method for exploiting soft functional dependencies. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1222–1233.

[51] Alon Y. Levy and Yehoshua Sagiv. 1995. Semantic Query Optimization in Datalog Programs (Extended Abstract). In *SIGMOD*. 163–173.

[52] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 615–629.

[53] Hai Liu, Dongqing Xiao, Pankaj Didwania, and Mohamed Y Eltabakh. 2016. Exploiting soft and hard correlations in big data query optimization. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1005–1016.

[54] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sharon Lee, Sicheng Pan, Joshua Wu, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2022. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. https://doi.org/10.48550/ARXIV.2205.02954

[55] Michael Meier, Michael Schmidt, Fang Wei, and Georg Lausen. 2013. Semantic query optimization in the presence of types. *J. Comput. System Sci.* 79, 6 (2013), 937–957.

[56] J. Minker (Ed.). 1988. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc.

[57] Noel Novelli and Rosine Cicchetti. 2001. Fun: An efficient algorithm for mining functional and embedded dependencies. In *International Conference on Database Theory*. Springer, 189–203.

[58] Thorsten Papenbrock and Felix Naumann. 2016. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*. 821–833.

[59] Thorsten Papenbrock and Felix Naumann. 2017. A hybrid approach for efficient unique column combination discovery. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)* (2017).

[60] Glenn Norman Paulley. 2001. *Exploiting functional dependence in query optimization.* Citeseer.

[61] Hamid Pirahesh, Joseph M Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. *ACM Sigmod Record* 21, 2 (1992), 39–48.

[62] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database management systems*. McGraw-Hill.

[63] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. 1987. A System for Semantic Query Optimization. In *SIGMOD*. 181–195.

[64] Jing Nathan Yan, Oliver Schulte, MoHan Zhang, Jiannan Wang, and Reynold Cheng. 2020. SCODED: Statistical Constraint Oriented Data Error Detection. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of*

Data. 845–860.

[65] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing data constraints in database-backed web applications. In *ICSE '20: 42nd International Conference on Software Engineering*. ACM, 1098–1109.

[66] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 800–810.

[67] Hong Yao and Howard J Hamilton. 2008. Mining functional dependencies from data. *Data Mining and Knowledge Discovery* 16, 2 (2008), 197–219.

[68] Zuohao She, S. Ravishankar, and J. Duggan. 2016. BigDAWG polystore query optimization through semantic equivalences. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.