Detecting Query Inefficiencies in Web Applications

Chengpeng Wang
cwangch@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Abstract

The emerging database-as-a service platforms and persistent data framework create the convenience of deploying web applications than before. Developers can implement the functionality of an application in a compact development cycle. However, it is quite challenging to develop efficient web applications which can query records from database in an optimal manner. Developers may introduce anti-patterns unconsciously when coding SQL statements in the applications. These anti-patterns, which can be in different forms, are intended to meet their requirements, while they sometimes cause the efficiency issues in different ways.

In this survey, we summarize the existing works on the detection of inefficient queries in Web applications. According to the manners of generating queries, the big picture can be partitioned into three parts. Firstly, the interfaces offered by ORM frameworks allows developers to construct the queries by composing them, and the abstraction in these interfaces might cause the inefficient query construction. Secondly, the data required in the application is often fetched by SQL queries and data manipulation in programming language, which means the queries are composed by SQL queries and the interfaces of data structures, such as traversal in containers. Redundant records or attributes can be fetched by SQLs but not actually used in the application. Thirdly, the same query can be expressed by SQL queries in multiple ways, while different implementations differ in terms of efficiency. We will discuss recent works from these three perspectives and draw several conclusions and future works in the end.

1 Introduction

To manage a huge amount of data, web applications are designed by following a two-stack architecture. A back-end applications stack stores the persistent data, generate the requests of processing data retrieval. A front-end application stack implements the program logic, i.e., process the data fetched by the back-end stack. This architecture has gradually evolve into more concrete designs of architectures, such as Model-Controller-View(MVC) design. Figure 1 displays the architectures of MVC application. Controllers respond to the user's actions and invoke the proper queries to fetch data from the DBMS. In some cases, such queries are constructed by the APIs in frameworks, such as ORM frameworks, and

PL'18, January 01–03, 2018, New York, NY, USA
2018.

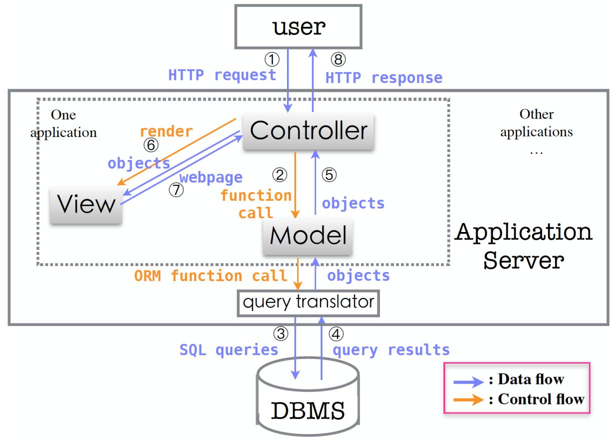


Figure 1. MVC architecture [1]

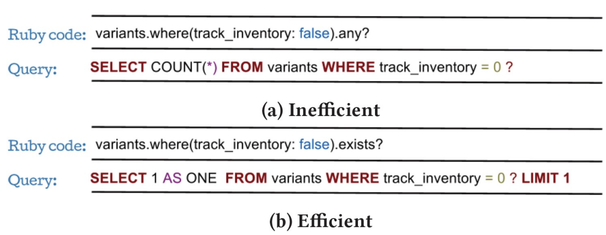
then translated to native SQL queries. The result of queries will be mapped to the objects in the applications, which are the instances of the models, and then these models are processed by the applications.

The application code often interacts with the DBMS in three ways, and mostly they are combined in many real-world applications. Firstly, developers can write native SQL statements to manipulate the database directly. Secondly, applications can use frameworks to operate the data in the DBMS instead of construct native SQL statement directly. For example, Object Relational Mapping(ORM) frameworks of general-purpose programming languages expose convenient interfaces for SQL construction and easy to be applied to a MVC architecture. Thirdly, program logic can also help SQL queries manipulate the data fetched from the DBMS. For instance, a query might return a list of the objects which we are interested in, and the list is then processed by the interfaces of a list in order to find the final desired result. Therefore, the implementations of a certain functionality often have multiple alternatives, with different combinations of native SQL statements, APIs in the frameworks and the program logic, and each implementation might behave differently in terms of efficiency. In this survey, we will discuss the performance issue of a query from these three aspects.

The organization of the survey is as follows. Section 2 summarizes the inefficient use of APIs in ORM framework and introduce how to detect them. Section 3 reviews a fundamental problem of SQL statements and discusses its application in the removal of redundant SQL execution. Section 4.1 summarizes the optimization of SQL statements, ORM usage and program logic simultaneously. The limitation of each approach is briefly discussed and some interesting future works are also mentioned at the end of each section.

2 Detecting Inefficient ORM Usage

To help developers implement database-backed applications, Object Relational Mapping(ORM) frameworks have been proposed and gained a significant increase of popularity. For almost all common programming languages, there are corresponding implementations: Hibernate for Java, SQLAlchemy for Python, and the Ruby on Rails for Ruby. Benefited from the APIs exposed by these ORM frameworks, developers are able to operate persistent data stored in the database in the same way of manipulating regular heap objects. It is proper to take an analogy that an ORM frameworks is an variant of Java Collection Framework which wraps the tables and records in the database in its containers and offers the interfaces manipulating them.



(a) Inefficient

(b) Efficient

Figure 2. Different APIs cause huge performance difference [2]

Although ORM frameworks release the burden of writing complex native SQL statements, developers often ignore the implementation details of these APIs and write the inefficient queries. A functionality can be implemented by different compositions of the APIs, while they can make big difference in performance. For example, Figure 2 shows two implementations of checking if there are products whose inventories are not tracked in an online shopping system. The Ruby on Rails contains two APIs, namely *any*? and *exists*?, which supports this functionality. However, these two queries have a substantial difference in terms of performance. Specifically, the first implementation is transformed into a SQL query which scan the whole table, collect the records satisfying the specified property and count the number of these records, while the query transformed from the second implementation stops scanning the remaining records if one records with the property is found. By fixing such API misuse, the server time of an application can be improved significantly. As reported in [2], the replacement of *any*? with *exists*? improves server time by 1.7×. It is quite a promising performance optimization because the detecting and fixing process are both simple and straightforward, which do not demand advanced program analysis and testing techniques.

Table 3 lists other equivalent pairs in the Ruby on Rails. Based on these patterns observed by experts, Yang and Lu

Functionality	Inefficient	Efficient
Pick one item by a condition	where.first	find_by
Update records by a condition	each.update	update_all
Find the number of items	.count	.size

Figure 3. Equivalent pairs in Rails

propose an approach to detecting the existing of such inefficient patterns by regular expression matching [2, 3]. The analysis only perform syntactic check upon the source code, and even does not require the integrity of the application code. Therefore, it permits the incremental analysis of the applications by checking each modules individually so that developers can analyze their own modules once developed, which is quite important to analyze large-scaled applications.

One major limitation is that the detection rules are higly dependent to expertise knowledge. The equivalence relation and performance superiority are sometimes not easy to obtain. It is an interesting problem to automatically generate such equivalent API sequences with the fact of their performance superiority from available artifacts, such as the documentations of ORM frameworks. Some techniques in the community of programming language, such as equational reasoning, might provide new insight to this problem [4, 5].

3 Optimizing Native SQL Statements

Writing native SQL statements is often more efficient than constructing SQL queries by ORM APIs because no construction process is involved. Even if native SQL statements have optimized the performance to some extent, the writing native SQL statements still often exhibit significant overlap of computation. Typically, redundant execution of particular sub-queries introduce the unnecessary overhead of applications. If we find two SQL statements across the application perform the same operation on the storage of DBMS, we can omit the second one safely and reuse the former query result directly.

```
Q1: SELECT COUNT(*) FROM
      (SELECT * FROM
        (SELECT * FROM EMP WHERE DEPT_ID = 10) AS T
      WHERE T.DEPT_ID + 5 > T.EMP_ID);

Q2: SELECT COUNT(*) FROM
      (SELECT * FROM
        (SELECT * FROM EMP WHERE DEPT_ID = 10) AS T
      WHERE 15 > T.EMP_ID);
```

Figure 4. Equivalent SQL queries [6]

It is an challenging problem to determine if two SQL queries are semantically equivalent. It has been proved that the general form of this problem is undecidable [7]. Fortunately, it is still possible to identify a subset of relational algebra. For example, the problem of deciding the equivalence of two *SELECT-PROJECT-JOIN* queries is decidable.

Zhou has proposed a series of works on verifying the equivalence by a logic method [6, 8]. He borrows the insight from the equivalence verification in programming language [9] and describe the input-output relation of a SQL query by logical formula. Technically, the expressions and predicates in a SQL query are encoded by logical formulae. By checking the implication of two logical formulae, it is easy to determine whether two queries are equivalent or not. Figure 4 shows an example of two equivalent SQL queries. Regardless of the table content, the result of executing these two SQL queries must be the same, which is the number of employees that satisfy certain predicates.

```
Q1: <COND1, COLS1, ASSIGN1>
COND1: (v3 = 10 and !n3) and
       (v3 + 5 > v1) and (!n3 and !n1) )
COLS1: {(v4,n4)}
ASSIGN1: ---

Q2: <COND2, COLS2, ASSIGN2>
COND2: (v3 = 10 and !n3) and ((15 > v1) and !n1 )
COLS2: {(v5,n5)}
ASSIGN2: ---
```

Figure 5. Constraints in set semantics [6]

Figure 5 shows the constraints corresponding to Q1 and Q2. Here $(v4, n4)$ and $(v5, n5)$ represent the values of the aggregate function COUNT returned by Q1 and Q2. The COND in each constraint determines the existence of a record in the query result completely. It is obvious that Q1 and Q2 are equivalent by inspecting COND1 and COND2.

The approach is applied to detect equivalent SQL queries in real-world systems, such as Alibaba's MaxCompute database-as-a-service platform. It is reported that 11% queries are detected as equivalent thus redundant ones among a set of 17,461 queries, which reduces the compute and memory resource consumption by 36% and 35%, respectively.

Other automated approaches of determining SQL query equivalence adopt algebraic approaches [10]. They performs query rewrites by applying a set of rules to algebraic expressions of queries, and check the isomorphisms and homomorphisms between the rewritten algebraic expressions. However, these algebraic approaches are limited and unable to support certain widely-used forms of SQL queries. In comparison, the logic method proposed by Zhou is more general and applicable. In the future, logic representation of SQL queries can be an interesting and meaningful problem. Except for equivelance checking for performance optimization, the logical formulae abstracting the effect of SQL queries can also help us understand the functionality of application code and perform further analysis [11].

4 Optimizing Imperative Code

In real-world web applications, the logic of processing data in DBMS is often implemented in a combined way, i.e., the imperative code in the application and the SQL statements manipulate the data together. In this section, we discuss two kinds of the works which focus on the optimize imperative code by transforming it to SQL statements and identifying redundant data access respectively.

4.1 Rule-based Transformation

A web application is a hybrid system composed of imperative code in programming language and the statements in query language. One functionality can be achieved by different compositions of these two kinds of languages. On the one hand, we can query all the data we possibly need and post-process the data byimperative code in a programming language. On the other hand, we can implement the post-processing in the query language and get the data we actually desire by a query directly. Due to the mature optimization mechanism of the DBMS, the second implementation is likely to be more efficient.

```
findMaxScore(){
  boards = executeQuery("from Board as b
      where b.rnd_id = 1");
  scoreMax = 0;
  for(t : boards) {
    p1 = t.getP1();
    p2 = t.getP2();
    p3 = t.getP3();
    p4 = t.getP4();

    score = Math.max(p1, p2);
    score = Math.max(score, p3);
    score = Math.max(score, p4);
    if(score > scoreMax)
      scoreMax = score;
  }
  return scoreMax;
}
```

Figure 6. Code for highest score calculation [12]

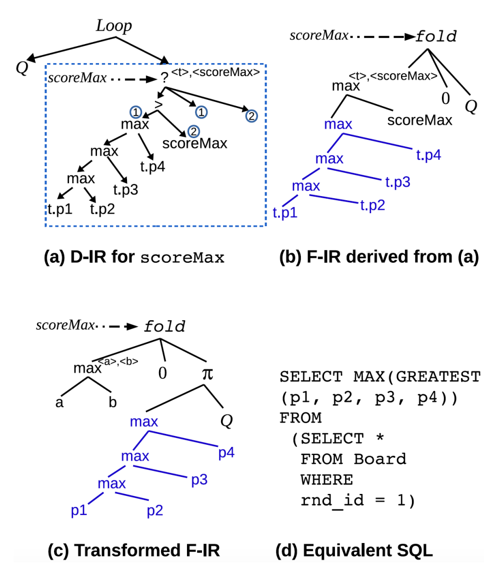


Figure 7. Walk-through of equivalent SQL derivation [12]

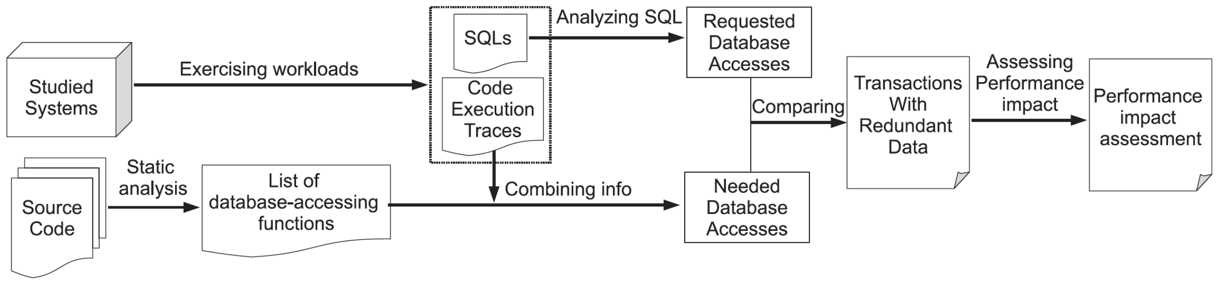


Figure 8. An architecture of redundant data detection [13]

Different from the works summarized in Section 2 and 3, the works located into this scope focus on multiple subjects of languages, and a uniform representation is necessary to reason the semantics of the code written in different languages. Emani and Sudarshan propose an algebraic representation of imperative code and queries uniformly [12, 14]. They define rule-based transformations on the algebraic representations to find whether it is possible to push computation into the relational algebra query or not.

Consider the code snippet shown in Figure 6. The highest score across all tables is found by a SQL query followed by a traversal of the fetched result. Figure 7(a) is the algebraic representation of the code where Q denotes the query $\sigma_{rnd\_id=1}(Board)$. After the transformations, a representation which can be expressed in the relational algebra finally reaches, i.e., the algebraic representation in Figure 7(c), and the SQL statement in (d) performs the same computation as the original code fragment.

The rule-based transformation is a powerful equational reasoning techniques, which is widely applied not only in query optimization, but also in the code of programming languages [15, 16]. The transformation can utilize more opportunities of optimization in the side of target languages, such as the query optimization in the DBMS. However, these optimizations still preserve all the computations even if they are performed in a more efficient way. Therefore, these approaches can not discover the efficient issues in the computation itself, such as the redundant data access.

4.2 Dynamic Profiling and Static Analysis

To analyze the web application in a finer grain, Chen and Flora propose an approach to analyzing the data access mode by dynamic profiling and then comparing profiled data with the result of static analysis [13, 17]. They focus on the problem that a huge amount of records and attributes are not actually used in the applications after being fetched from the database, and redundant data access takes the unnecessary bandwidth and execution time, which slows the respond time of the web application. Motivated by finding the performance impact of redundant data access, they compare the needed database accesses with requested database accesses to find the clues of redundant access.

Figure 8 shows the architecture of their approach. Firstly, they take advantage of static analysis to identify how the database accessing functions read and modify the instance variables mapped to database columns. Secondly, they leverage code instrumentation to collect the system traces which reflect which data is read or modified in the database. Finally, the comparison of these two facts exactly indicates the existence of redundant data access. The combination of dynamic profiling and static analysis provides a new possibility of detecting sophisticated query inefficiencies in the granularity of database columns, and can find more optimization opportunities than detecting redundant SQL execution discussed in Section 3.

5 Future Work

Based on the survey, we discover several interesting topics to be further explored.

Automatic discovery of equivalent API invocations. As summarized in Section 2, the inefficiency patterns are specified manually and highly depend on expertise knowledge. For a new ORM framework, developers have to endure a long process to discover such empirical rules to guide the performance analysis. If the equivalent relations of API invocation sequences are automatically obtained, then the whole detection process does not rely on the human guidance. Fortunately, the works on SQL equivalence checking has shown the feasibility of encoding the SQL interfaces and APIs in ORM frameworks by a constraint in bag semantics. The documentations of ORM frameworks define the semantics of each API, which is easily encoded by a constraint. Based on these materials, it is possible to enumerate API sequences and identify their equivalence relations by checking the implication of the constraints.

Data dependence analysis. Data dependence analysis is a fundamental static analysis techniques. When analyzing inefficient queries in the database-backed applications, data dependence analysis enables a detailed reasoning of def-use relationship. However, it is often restricted in the presence of frameworks and configuration files. Data dependency might be affected by the framework code and the segments in configurations [11]. It is an open problem to design a robust data dependence analysis for these system.

Loop analysis in SQL. Some queries are frequently invoked in a loop or a recursive function while return the same results. Loop hoisting techniques can be applied to these program structures to enhance the performance of the systems. Meanwhile, a SQL query might behave similarly to a function containing a loop, which means that there is a sub-query scanning all the records in a table. The results might be reused if the sub-queries are executed multiple times and the results are always the same. Loop analysis, such as loop hoisting, is a classical problem in programming language, and the analysis of SQL might discover more interesting optimizations for web applications.

6 Conclusion

This survey reviews three lines of works on query inefficiency detection in web applications. ORM frameworks, native SQL statements and imperative code in applications can introduce the inefficient queries in different ways. Pattern-matching based approaches are effective in detecting inefficient API usage of ORM frameworks. Constraint based approaches enables the identification of redundant SQL queries which can be removed to enhance the performance. Equational reasoning of imperative code and SQL statements provides more opportunities of optimization, by utilizing optimization mechanism of DBMS engine. Hybrid approaches, which combine dynamic profiling and static data dependence analysis, can cover the redundant data access problem in a finer grain. The sub-problems in each category are worth further exploring, most of which are the common concerns in the communities of database and programming languages.

Acknowledgment

The figures and examples are extracted from the papers. The citations are added to the titles of the figures.

References

[1] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1299–1308, 2017.

[2] 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 *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 800–810. IEEE, 2018.

[3] PowerStation. A Tool for Detecting Performance Bugs in Rails Applications, 2021.

[4] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in C for better testing and refactoring. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 874–888. ACM, 2019.

[5] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1066–1082. ACM, 2020.

[6] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment*, 12(11):1276–1288, 2019.

[7] Joseph Albert. Algebraic properties of bag data types. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 211–219. Morgan Kaufmann, 1991.

[8] Qi Zhou. A Symbolic Approach to Proving Query Equivalence Under Bag Semantics, 2021.

[9] Deokhwan Kim and Martin C Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 528–541, 2011.

[10] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for sql. In *CIDR*, 2017.

[11] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. Scaling static taint analysis to industrial soa applications: a case study at alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1477–1486, 2020.

[12] K Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S Sudarshan. Extracting equivalent sql from imperative code in database applications. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1781–1796, 2016.

[13] 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*, 42(12):1148–1161, 2016.

[14] K Venkatesh Emani and S Sudarshan. Cobra: A framework for cost-based rewriting of database applications. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 689–700. IEEE, 2018.

[15] Grégory M. Essertel, Guannan Wei, and Tiark Rompf. Precise reasoning with structured time, structured heaps, and collective operations. *Proc. ACM Program. Lang.*, 3(OOPSLA):157:1–157:30, 2019.

[16] Anders Møller and Oskar Haarklou Veileborg. Eliminating abstraction overhead of java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.*, 4(OOPSLA):168:1–168:29, 2020.

[17] 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 *Proceedings of the 36th International Conference on Software Engineering*, pages 1001–1012, 2014.