A Review of Database-backed Web Application Performance Anti-Pattern

Weisheng Wang
UNI: ww2609

Abstract
Data storage has become one of the most common features for web applications. While software developers design database schemas and program queries to retrieve data, they may inevitably introduce database-access performance anti-patterns (AP) which not only violate basic design principles and best practices but also bring down the overall system performance. For instance, APs may cause redundant storage and increase applications' response time. Besides, identifying APs and implementing fixes require much developers' effort and are time-consuming. Therefore, many studies aim to tackle the database-access performance AP problem by identifying APs and suggesting fixes. This paper will first introduce the background and explain common terms of the research space for the reader. Then, this paper will introduce various methods to solve database-access performance AP problems. This paper will also summarize and compare different AP categories from each literature and experiment with SQLCheck [6] to evaluate the tool's competence in detecting APs.

1 Introduction
This paper will give the reader a background introduction to the database-access performance AP research space. After explaining the basic knowledge about the research space, the paper will present various methods to address the performance AP issue such as ORM level APs [3, 4, 5, 7, 11, 13, 14, 15] and SQL level APs [6,8,9,10,11,12]. Although there are studies focusing on local database performance APs on mobile applications [14, 15], due to time constraints and my interests, this paper will not review literature about this sub-research space, but interested readers may read the cited works for more details. Next, this paper will illustrate studies that transferred knowledge about SQL APs to other SQL-like programming languages through the example of DeFiHap [10] built for HiveQL. After reviewing the past literature, the paper will address the following research questions:

- **RQ1**: What are the APs covered by the prior literature?
- **RQ2**: How well does SQLCheck do in detecting APs?

Finally, this paper will conclude with my discussion about what he learned and what other subject matters could be added to increase the comprehensiveness of the paper.

After reading this paper, the reader can expect to learn about (1) background knowledge about different categories of database-access performance APs such as SQL-level and ORM-level APs, (2) specific APs and their examples to better understand how and why APs impair data access performance, (3) an overview of AP coverage by different literature, (4) how AP detection tools such as SQLCheck works and performance benefits they bring.

2 Background
Performance of web applications is crucial for user experience. Oftentimes, users will leave a web page if it's been loading for a few seconds. Additionally, performance APs will pose catastrophic user experiences. For instance, many performance issues, such as insufficient capacity, presented in Healthcare.gov's initial deployment and resulted in its users encountering numerous difficulties in

accessing its functionalities [1]. Among different performance factors, database-access performance plays an important role in affecting web applications' ability to support data storage and retrieval efficiently.

Database-access performance APs are the design choice or practice that aims to solve a database-access problem initially but fail to comply with some fundamental design principles, which lead to other problems. In section 4.1, this paper will present a case study from SQLCheck to illustrate a SQL level AP example.

While database-access performance APs raise developers' attention, it is challenging to identify potential APs when developing applications and it often requires the developers to have sufficient knowledge and experience to avoid all the bad design decisions that impair performance. To understand and handle database-access performance issues, researchers have examined popular real-world web applications to extract and categorize common performance APs and evaluated the APs' impacts on applications' performance. Some studies propose SQL level APs detection and fix suggestion approaches. With using object-relational-mapping (ORM) frameworks, written in object-oriented programming languages to provide data model abstraction for developers to reduce the burden of CRUD operations, to build web applications become popular, some researchers have examined ORM-based applications and also found ORM level APs that affect performance. Some studies focus on mobile applications and analyze how APs affect energy consumption and may introduce security vulnerabilities.

Because many SQL APs are rule-based, they also exist in SQL-like languages. Some studies have successfully transferred the knowledge of SQL APs to detect and fix APs in SQL-like languages. In DeFiHap, the authors develop a tool to detect and fix HiveQL APs.

Dynamic analysis v.s. static analysis - researchers approach the database-access performance AP problem with different methods. Some built their tools with dynamic analysis, which run and detect APs at runtime, while others employ static analysis, which runs on the source code, detect and suggest fix at the coding stage. Also, there are studies that use a hybrid approach of leveraging both static and dynamic analysis to improve the precision of their tools.

Create, read, update, delete (CRUD) - these are the most basic and commonly used data manipulation operations when interacting with databases. In SQL language, create maps with INSERT; read maps with SELECT; update maps with UPDATE; delete maps with DELETE.

3 ORM Performance APs

With the increasing demand for improving productivity, many web developers adopt ORM frameworks to connect their web applications with databases. However, although the ORM frameworks, providing developers with an object-oriented abstraction of data accessing to help them focus on programming business logic, bring convenience to conduct CRUD operations, prior work [14] found that developers encounter many difficulties to develop performant ORM-based applications due to the existence of APs and that AP detection and avoidance require much relevant knowledge and experience.

Many studies have been conducted to understand and handle ORM level APs. In [4], the paper purposed an automated framework for the detection and assessment of 2 common performance APs in ORM-based applications. Through statistical assessment, the framework can prioritize detected APs based on expected performance gains in terms of response time improvement. In the authors' later work [5], they focused on redundant data issues and proposed a hybrid approach, combining static and dynamic analysis, to locate the source code that causes the problem with good accuracy for applications using Java Persistent API. In [14], the authors examined 12 popular ORM-based applications and summarized 9 common performance APs that affect application efficiencies. To prove the validity of their findings, they measured performance improvements after manually fixing the APs. In Powerstation [15], the authors built an AP detection and fixing suggestion tool for Ruby on Rails ORM-based applications in the forms of a RubyMine IDE plugin, and the tool is capable of automatically detecting 6 common APs (summarized from [5, 13, 14]) and providing patches for 5 of them. In Dbridge [7], the work presents a novel approach to translating the imperative object-relational codes to SQL queries in order to reduce unnecessary data accessing and unused data. In [11], the authors perform a comprehensive literature survey of prior literature and collect 24 known performance APs while finding an additional 10 new APs through experiments. Since the new APs deal with direct accessing which is similar to SQL level APs, in the Venn diagram of Appendix 8.2, I put this work into the SQL level AP circle as well. In [3], the authors collect 17 performance APs from prior works and experiment with an industrial web application written in the PHP Laravel framework. They have found that when the workload scale is large, the performance impact caused by APs becomes significantly high, whereas, operating under a low scale, such impact might not be as obvious. Furthermore, they discover 2 new performance APs through dynamic analysis.

It's worth noting that in [5], the author explicitly distinguishes performance APs from redundant data problems because performance APs care more about not violating design principles that impair the performance, whereas redundant data problem is caused by requests of unused data. However, in [14], the authors categorize redundant data problems as an ORM level AP called unnecessary data retrieval (UD). In my opinion, summarizing redundant data as an ORM AP is reasonable, because, in essence, retrieving data that will not be used by the application increases network traffic and the application's capacity of handing requests, which implicitly impact the application performance.

3.1 ORM Performance AP Example

In this section, I will use a Ruby code fragment example from PowerStation to illustrate an ORM level performance AP called loop invariant queries (LI).

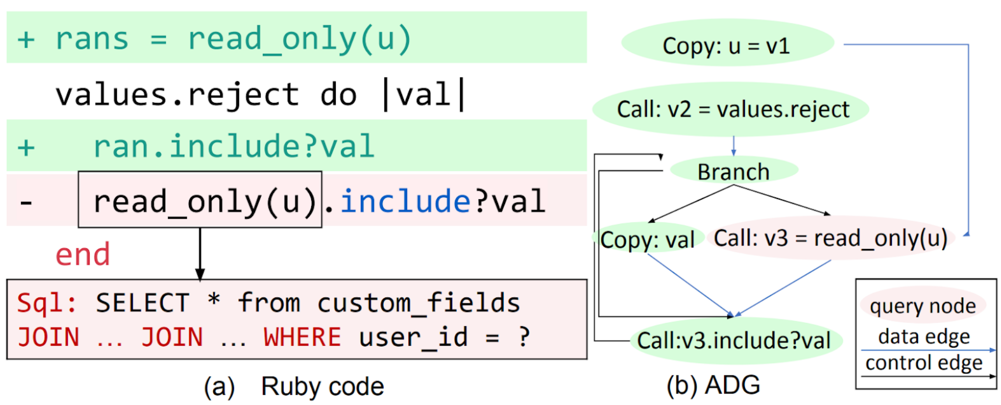


Figure 1. Redmine code fragment with LI AP

Loop invariant queries refer to the queries that are repeatedly called per iteration inside a loop which can be avoided by storing the query result as an object before executing the loop and calling the object's methods during each iteration. In Figure 1.a, the Ruby code fragment checks whether user *u*'s read-only custom fields contain any element *val* from *values.reject* list. However, the original code fragment puts the query call inside the loop which executes the query the size of values.reject times. Whereas, with fixing the AP through assigning the query result before executing the loop (line 1), only 1 query call will be made and thus, avoid repeated calls and bring performance improvement. Action dependency graph (ADG), which takes in Ruby on Rails source code and then generates a dependency graph of the program execution process, is a crucial component in PowerStation's static analysis. In Figure 1.b, the authors present an ADG for explaining the order in which the member methods of different Ruby classes and objects are called during the execution.

4 SQL Performance APs

While using ORM frameworks to improve developers' productivity in web applications become popular, still many developers perform CRUD operations with plain SQL statements to retrieve or store data from web applications to databases or vice versa.

Many researchers study various aspects related to SQL. In DbDeo [12], categorizing database smells (APs) into three categories, schema, query and data smells, the paper particularly examines performance issues in database schemas and summarizes 13 database schema smells (APs) from past literature. They also built a tool called DbDeo to detect smells from embedded SQL statements. In SQLCheck [6], finding that DbDeo has low precision and recall and only supports AP detection, the authors developed SQLCheck, which not only detects APs but also ranks them based on statistically estimated performance impact and provides fix suggestions. The authors compiled a catalog of four categories of APs from various sources, such as Kaggle and StackOverflow, which discuss the best database schema design practices and fundamental principles in structuring SQL quires.

4.1 Case Study: GlobaLeaks

In this section, I will illustrate the case study on GlobaLeaks from SQLCheck to give readers an example of how database schema APs affect performance.

Tenant_ID	Zone_ID	Active	User_IDs
T1	Z1	True	U1 , U2
T2	Z3	True	U3 ; U4

(a) Tenants Table

User_ID	Name	Role	Email
U1	N1	R1	E1
U2	N2	R2	E2
U3	N3	R3	E3
U4	N4	R4	E4

(b) Users Table

Figure 2. Original GlobalLeak tables

User_ID	Name	Role	Email
U1	N1	R1	E1

(a) Users Table

Tenant_ID	Zone_ID	Active
T1	Z1	True
T2	Z2	True

(b) Tenants Table

Tenant_ID	User_ID
T1	U1
T1	U2
T2	U3
T2	U4

(c) Hosting Table

Figure 3. Refactored GlobaLeak tables

In Figure 2, there are two tables, Tenants and Users, from the database schema of GlobaLeak. We can observe that, for the Tenants table, the User_IDs column supports the multi-valued attribute. However, it appears the column is of VARCHAR or TEXT type that the multi-valued attribute is accomplished by splitting two values with a comma or a semicolon. This schema design leads to performance and maintainability issues. For instance, as the length of the values in the User_IDs field increase, the cost of updates will also increase. Besides, without proper restrictions, allowing the values to be split by different delimiters increases the chance of mishandling the data and causing system errors. Moreover, such schema design introduces security vulnerabilities such as the risk of SQL injection. Figure 3 shows a refactored database schema that fixes the original logical design AP.

Algorithm 3: Detecting Anti-Patterns via Data Analysis

```
input  : context C, database D
output : detected APs
anti-patterns P ← { }
for rule d in data rules D do
    for table t in D.tables do
        // sample tuples from the table
        sampled tuples s = SAMPLE(t)
        // use data rules to reduce false positives and negatives
        if r(C[t], s) then
            P.append(p)
return P
```

Figure 4. Data analysis algorithm designed by SQLCheck

We can observe from Figure 2 that examining database schema alone is not enough for probing performance APs, because, in SQL statements, we only assign the User_IDs column as VARCHAR or TEXT without knowing what data will be stored. Therefore, leveraging data analysis, SQLCheck implemented an algorithm (Figure 4) to detect the APs and thereby, improved the tool's precision and recall. For the GlobaLeak multi-valued example, SQLCheck will sample some rows from a table, and then apply a data rule to check if a column field with VARCHAR or TEXT type has been used to store delimiter-separated strings.

5 Transferring AP Knowledge to SQL-Like Language

While SQL is the dominant standardized programming language in the data management world, there also exist SQL-like programming languages which function as SQL but provide additional functionalities to serve special purposes.

For instance, Apache Hive, providing data warehouse infrastructure for data query and analysis, uses a SQL-like language called HiveQL for queries. In DeFiHap, the authors review prior literature and generalize 38 performance APs. Then, they develop DeFiHap, a tool to automatically detect 25 HiveQL APs and suggest fixes for 17 of the APs. In addition to statement APs, they also examine configuration APs that particularly exist in Hive for it runs on top of the Hadoop MapReduce framework which spread queries to distributed clusters. For example, Figure 5 shows the experiment results from DeFiHap. The authors find that setting the *mapred.reduce.tasks* parameter properly can significantly improve the performance of a join query. Then, they categorize this configuration AP as inappropriate number of reducers and develop its fix suggestion through training machine learning models to recommend the proper reducers amount given a join query.

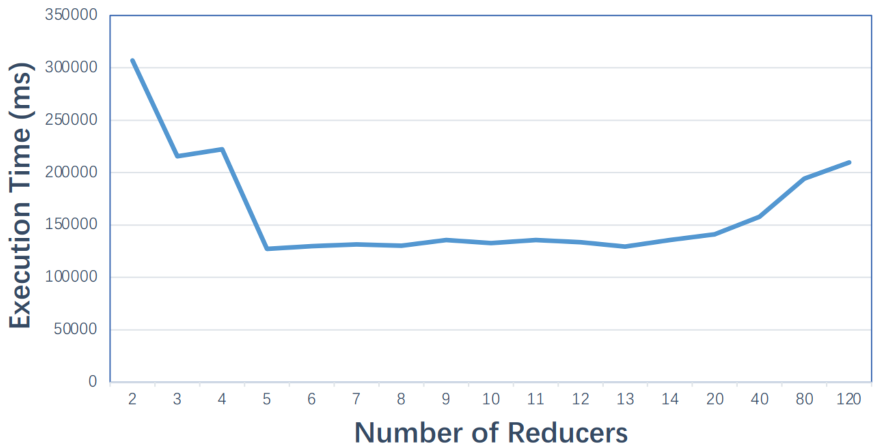


Figure 5. Execution time of a join query v.s. number of reducers specified in the configuration

6 Research Questions

In this section, I will go over how I study the two proposed research questions through literature review and a small experiment with SQLCheck. For RQ1, I collected APs from [6, 10, 11], discussed similar APs, and compared different AP categories. For RQ2, I experimented with SQLCheck on an example SQL query provided by the paper and then, run the tool to detect APs in the database schema of a Kaggle dataset called *The History of Baseball*.

6.1 RQ1: What are the APs covered by the prior literature?

Methodology - (1) Instead of collecting APs from all the literature, I mainly gather APs from [6, 10, 11]. One reason I pick these three pieces of literature is that they each represent their domain, one for ORM APs, one for SQL APs, and one for HiveQL APs. Another reason is that [11] has done a great job in summarizing the ORM APs from [3, 4, 5, 14, 15] and SQLCheck also covers the SQL APs that [12] studies. Therefore, APs from these three studies are comprehensive and representative. (2) After that, I will discuss similar APs and compare different AP categories.

AP Collection and Labeling - SQL level APs are presented in Figure 6, while ORM and HiveQL APs are shown in Appendix 8.1. I will use the same AP ids, AP-01 to AP-34, for [11] APs; for APs from SQLCheck, I will label the APs from SQ-01 to SQ-26; for APs in DeFiHap, because they are in two categories, I will label the statement AP from S-AP-01 to S-AP-14 and the configuration AP from C-AP-01 to C-AP-11. (DeFiHap only lists 25 APs, which it's capable of detecting, among the reported 38 HiveQL APs.)

AP Discussion - I find that AP-05 dead-store queries, which stores the results of multiple queries in one object and is left unused, shares some similarities with AP-14, AP-15, and AP-16 that they all cause the retrieval of unused data which should be avoided to improve performance, although these APs uses subtly different fix strategy. Moreover, AP-25 and AP-26 are similar to SQ-11 and SQ-12 in that the APs are caused by improper usage, either overuse or underuse, of indexes. Also, AP-27, AP-28, Ap-29, SQ-19, SQ-20, S-AP-01, S-AP-03, and S-AP-11 all work on the different performance issues caused by improper usage of the JOIN operator.

Category	Anti-Pattern Name	Description
Logical Design APs	Multi-Valued Attribute	Storing list of values in a delimiter-separated list violating 1-NF.
	No Primary Key	Lack of data integrity constraints.
	No Foreign Key	Lack of referential integrity constraints.
	Generic Primary Key	Creating a generic primary key column (*e.g.*, `id`) for each table.
	Data In Metadata	Hard-coding application logic in table's meta-data.
	Adjacency List	Foreign key constraint referring to an attribute in the same table.
	God Table	Number of attributes defined in the table cross a threshold (*e.g.*, 10)
Physical Design APs	Rounding Errors	Storing fractional data using a type with finite precision (*e.g.*, `FLOAT`).
	Enumerated Types	Using `enum` to constrain the domain of a column.
	External Data Storage	Storing file paths instead of actual file content in database.
	Index Overuse	Creating too many infrequently-used indexes.
	Index Underuse	Lack of performance-critical indexes.
	Clone Table	Multiple tables matching the pattern <TableName>_N
Query APs	Column Wildcard Usage	Selecting all attributes from a table using wildcards to reduce typing.
	Concatenate Nulls	Concatenating columns that might contain `NULL` values using `||`.
	Ordering by `RAND`	Using `RAND` function for random sampling or shuffling.
	Pattern Matching	Using regular expressions for pattern matching complex strings.
	Implicit Columns	Not explicitly specifying column names in data modification operations.
	`DISTINCT` and `JOIN`	Using `DISTINCT` to remove duplicate values generated by a `JOIN`.
	Too Many Joins	Number of JOINs cross a threshold.
Data APs	Missing Timezone	Date-time fields stored without timezone.
	Incorrect Data Type	Actual data does not conform to expected data type.
	Denormalized Table	Duplication of values.
	Information Duplication	Derived columns (*e.g.*, age from date of birth).
	Redundant Column	Column with NULLS or same value (*e.g.*, en-us)
	No Domain Constraint	All values should belong to particular range (*e.g.*, rating)

Figure 6. SQL level APs with AP ID ranges from SQ-01 to SQ-26 from top to bottom.

With careful examination of the root cause and fix strategy of all the APs, I find that despite that some APs share similar functionality, merging them as one AP might be overgeneralizing and provide fewer hints for giving specific AP fixes. Also, it's hard to merge HiveQL APs with SQL APs because although some HiveQL statement APs are inherited from SQL APs, still the HiveQL APs mostly work in the context of Hive, and thus the two categories should not be merged.

6.2 RQ2: How well does SQLCheck do in detecting APs?

Methodology - (1) Collect the SQLite schema of *The History of Baseball* used in the SQLCheck paper. (2) Run SQLCheck on an example SQL query and evaluate AP detection correctness (3.a) Run SQLCheck on the schema and compare with the original reported result. (3.b) Sample a random AP from the detected APs and check if SQLCheck correctly identifies the APs. Raw data, SQLCheck results, and sample commands are in https://github.com/BotMichael/6156-PerformanceAP-paper-experiment.

6.2.1 Query Experiment

Data Description - This SQL query wants to retrieve the number of orders for each order priority. It also restricts counting orders which is within the 3 months after 08/01/1994 and the order has more than 1 line item whose commit date is earlier than its receipt date.

```
select
        o_orderpriority,
        count(*) as order_count
from
        orders
where
        o_orderdate >= date '1994-08-01'
        and o_orderdate < date '1994-08-01' + interval '3' month
        and exists (
                select
                        *
                from
                        lineitem
                where
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
order by
        o_orderpriority;
```

Figure 7. An example SQL query statement from SQLCheck

Result - SQLCheck detects the SQL with having 1 high-risk query AP, *SELECT* *, and 1 low-risk query AP, *GROUP BY Usage*. (Verbose results are in GitHub) This SQL may suffer the following performance issues:

From *SELECT* *,

1. Retrieving unused data: Using SELECT * in SQL statements will normally result in retrieving unneeded columns. Retrieving such unused data from the database to the server slows the data access process, increases the overall load on machines and the network.
2. Indexing issues: Since SELECT * may retrieve more columns than needed, specially defined indexes might not be used to improve data retrieval. For instance, there might not be an index that covers all the columns in the SELECT list.
3. Binding: If the query intends to retrieve columns with the same name (i.e. as a result of joining two tables that have some fields sharing the same name), then the data consumer might not be able to determine the corresponding table for the same name columns which pose a huge risk for data usability. Although normally backend servers will consume the query results and, therefore, know the structure of the query results, yet, leaving backend servers to handle such logic will increase the developers' burden when maintaining the system and developing new features.

From *GROUP BY USAGE*,

1. referencing non-grouped columns: In most DBMSs, they follow the Single-Value Rule that columns in the SELECT clause must either be in the GROUP BY clause or used inside aggregation functions. Otherwise, the DBMS will report an error. Therefore, developers should be cautious when writing queries with SELECT and GROUP BY clauses.

Evaluation - (1) *SELECT* * query AP corresponds to the *column wildcard usage* AP (SQ-14) which selects all the columns from the table. SQLCheck correctly detects this AP pattern in the example query. Although we can find that since the wildcard is used in an aggregation function, the potential issues listed above are not likely to happen, yet, in terms of following the best engineering principles, avoiding using wildcards is a better choice. (2) GROUP BY Usage query AP corresponds to the *SELECT Inconsistent with GROUP BY* AP (S-AP-06) listed in DeFiHap, while no corresponding query AP has been found in

SQLCheck paper. The reason is that the authors implemented more AP patterns after their paper was published. In addition, this is a false positive AP detection because the SQL query only uses the column attribute from the GROUP BY clause and puts the wildcard in an aggregation function which will not cause errors as well. For this example, the false positive rate is 50%.

6.2.2 Schema Experiment

Data Description - *The History of Baseball* dataset contains 26 CREATE SQL statements.

Result - SQLCheck detected 27 APs, including 25 high-risk AP *Multi-Valued Attribute* (all with the same kind), 1 medium-risk AP *Metadata Tribble*s, and 1 low-risk AP *Spaghetti Query Alert*. (Result details are in GitHub repo) Since *Multi-Valued Attribute* AP has been illustrated in section 4.1, I will only summarize the potential performance issues due to *Metadata Tribble*s and *Spaghetti Query Alert*:

From *Metadata Tribble*s,

1. breaking down columns by specific values: it's a bad practice to use some columns' values (metadata) such as dates for naming the column, which will make the relational table hard to maintain. SQLCheck suggests 2 remedies for addressing this issue: one is sharding and another approach is creating a dependent table.
2. storing similar value-named columns in one column: some tables may have columns whose names are values with similar meanings. Store these values in one single column can help improve maintainability and optimize data storage.

From *Spaghetti Query Alert*,

1. combined multiple queries into one complex query: overly complex queries may accidentally introduce unintended Cartesian products which may end up generating unused data. SQLCheck advises decomposing overly complex queries into small simple queries to prevent performance issues.

```
CREATE TABLE college (
    college_id TEXT,
    name_full TEXT,
    city TEXT,
    state TEXT,
    country TEXT);
```

Figure 8. College table in *The History of Baseball*

Evaluation - In the SQLCheck paper, the authors propose to use data analysis to assist AP detection. However, their tool only takes in a SQL file and their cool hardcoded the *Multi-Valued Attribute* Pattern as "id text". Therefore, it's no surprise that the tool reports the 25 such APs. However, it may lead to false positives because some tables may use TEXT type for their id attribute. For instance, in Figure 8, the College table uses TEXT type for its college_id attribute as its primary key. However, matching the expression "id text", this statement has been identified as having the *Multi-Valued Attribute* AP by SQLCheck. Because this dataset uses TEXT for all its tables' id fields (except only the *yearid* attribute in the *hall_of_fame* table has INTEGER type), my experiment shows that SQLCheck has a false positive rate of 100% in identifying *Multi-Valued Attribute* AP, in this example. In addition, for the query AP *Spaghetti Query Alert,* SQLCheck mistakenly treats the *team* table's schema statement as a query. For

*Metadata Tribble*s, it is detected in the *team* table's *team_id_lahman45* field which is correct. For this example, the false positive rate is about 96.3%.

7 Conclusion

This paper first covers the background of database-backed web application performance APs. Then, the paper describes the two popular research branches of the research space, ORM level AP and SQL level, with illustrations about AP examples and a case study on GlobaLeak to help the reader understand how and why APs impact application performance. Furthermore, this paper presents how other SQL-like languages can leverage SQL AP knowledge to solve their APs. Finally, this paper answers the two research questions with a literature review of [6, 10, 11] and two experiments with SQLCheck.

After reading the cited literature, I find that newly published papers can always discover new performance APs apart from the summarized ones. This fact should remind AP detection tool developers of the importance to achieve good modularity and maintainability so that when new APs are discovered, the developers can conveniently implement the rules (for rule-based APs) to the tool.

Through reading DeFiHap, I have learned to think out of the box such as applying knowledge about SQL APs to other SQL-like programming languages and even other domains. With some past experience with using Neo4J, a graph database management system, and its graph query language Cypher, I searched on Google Scholar with keywords “Neo4J”, “Cypher” and “Anti-Pattern”, but could not find a paper that deals with graph query APs. Based on the above literature review, it’s almost impossible that developers won’t bring in APs when coding graph queries. Therefore, I believe there are numerous interesting and unexplored research opportunities in the graph query AP research space.

Moreover, this assignment gives me the opportunity to explore a specific research domain. Not only have I gained experience in doing literature reviews but also I have learned how to approach an unfamiliar topic step by step and where to find credible resources to guide my exploration. Besides, during the research process (i.e. grouping similar literature together for drawing the Venn diagram), I also learned how to extract the key contents, summarize and organize them.

8 References

1. Anthopoulos, L., Reddick, C. G., Giannakidou, I., & Mavridis, N. (2016). Why e-government projects fail? An analysis of the Healthcare. gov website. *Government information quarterly*, *33*(1), 161-173. https://doi.org/10.1016/j.giq.2015.07.003

2. Camacho-Rodríguez, J., Chauhan, A., Gates, A., Koifman, E., O'Malley, O., Garg, V., ... & Hagleitner, G. (2019, June). Apache hive: From MapReduce to enterprise-grade big data warehousing. In Proceedings of the 2019 International Conference on Management of Data (pp. 1773-1786). https://doi.org/10.1145/3299869.3314045

3. Chen, B., Jiang, Z. M., Matos, P., & Lacaria, M. (2019, November). An industrial experience report on performance-aware refactoring on a database-centric web application. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 653-664). IEEE. https://doi.org/10.1109/ASE.2019.00066

4. Chen, T. H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2014, May). Detecting performance anti-patterns for applications developed using object-relational mapping. In Proceedings of the 36th International Conference on Software Engineering (pp. 1001-1012). https://doi.org/10.1145/2568225.2568259

5. Chen, T. H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2016). 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. https://doi.org/10.1109/TSE.2016.2553039

6. Dintyala, P., Narechania, A., & Arulraj, J. (2020, June). SQLCheck: automated detection and diagnosis of SQL anti-patterns. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (pp. 2331-2345). https://doi.org/10.1145/3318464.3389754

7. *Emani, K. V., Deshpande, T., Ramachandra, K., & Sudarshan, S. (2017, May). Dbridge: Translating imperative code to sql. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 1663-1666).* https://doi.org/10.1145/3035918.3058747

8. Lyu, Y., Alotaibi, A., & Halfond, W. G. (2019, September). Quantifying the performance impact of SQL antipatterns on mobile applications. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 53-64). IEEE. https://doi.org/10.1109/ICSME.2019.00015

9. Lyu, Y., Volokh, S., Halfond, W. G., & Tripp, O. (2021, July). SAND: a static analysis approach for detecting SQL antipatterns. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 270-282). https://doi.org/10.1145/3460319.3464818

10. Mao, Y., Yuan, S., Cui, N., Du, T., Shen, B., & Chen, Y. (2021). DeFiHap: detecting and fixing HiveQL anti-patterns. Proceedings of the VLDB Endowment, 14(12), 2671-2674. https://doi.org/10.14778/3476311.3476316

11. Shao, S., Qiu, Z., Yu, X., Yang, W., Jin, G., Xie, T., & Wu, X. (2020, September). Database-access performance antipatterns in database-backed web applications. In *2020 IEEE*

International Conference on Software Maintenance and Evolution (ICSME) (pp. 58-69). IEEE. https://doi.org/10.1109/ICSME46990.2020.00016

12. Sharma, T., Fragkoulis, M., Rizou, S., Bruntink, M., & Spinellis, D. (2018, May). Smelly relations: measuring and understanding database schema quality. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (pp. 55-64). https://doi.org/10.1145/3183519.3183529

13. Yan, C., Cheung, A., Yang, J., & Lu, S. (2017, November). Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (pp. 1299-1308). https://doi.org/10.1145/3132847.3132954

14. Yang, J., Yan, C., Subramaniam, P., Lu, S., & Cheung, A. (2018, May). 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) (pp. 800-810). IEEE. https://doi.org/10.1145/3180155.3180194

15. Yang, J., Yan, C., Subramaniam, P., Lu, S., & Cheung, A. (2018, October). Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 884-887). https://doi.org/10.1145/3236024.3264589

8 Appendix

8.1 AP Collection

Category	Anti-Pattern Name	Description	Impact*	Detection	Fix
Statement AP (S-AP)	Large Table on the Left	Putting table with more records on the left of JOIN.	P	rule-based	rule-based
	Greedy Selection	Using SELECT * which could retrieve redundant result.	P, M	rule-based	rule-based
	Too Many JOINs	Using more than one JOIN operation.	P	rule-based	-
	Misusing HAVING	Using HAVING without GROUP BY.	P	rule-based	rule-based
	Misusing INTERVAL	Combining INTERVAL and DATE_SUB( ) for date query.	E	rule-based	rule-based
	SELECT Inconsistent with GROUP BY	Missing selected columns after GROUP BY.	E	rule-based	rule-based
	Calculation in Predicate	Calculating in predicates after ON or WHERE.	P	rule-based	-
	Calling Functions in Predicate	Calling functions in predicates after ON or WHERE.	p	rule-based	-
	No Group By	Using aggregation functions without GROUP BY.	E	rule-based	-
	Using ORDER BY	Using ORDER BY instead of SORT BY.	P	rule-based	-
	JOIN in Subquery	Using JOIN in the sub-query.	P	rule-based	-
	Creating Duplicate Table	Creating a table having the same column properties as another table in the database.	P, M	rule-based	-
	Querying without Partition	Querying on a partitioned table without using partition filter.	P	rule-based	rule-based
	Data Skew	Querying on a dataset with a non-uniform distribution.	P	rule-based	-
Configuration AP (C-AP)	Inappropriate Number of Reducers	Setting too many or too few reducers for a JOIN operation.	P	tuning	tuning
	Disabled Column Pruner	Not enabling the column pruner configuration item.	P	rule-based	rule-based
	Disabled Partition Pruner	Not enabling the partition pruner configuration item.	P	rule-based	rule-based
	Disabled Output Compression	Not enabling the output compression configuration item.	P	rule-based	rule-based
	Disabled Parallelization	Not enabling the parallelization configuration item.	P	rule-based	rule-based
	Disabled Cost based Optimizer	Not enabling the cost based optimizer (CBO) configuration item.	P	rule-based	rule-based
	ExecutorService Rejection	Task is rejected by executorService.	E	rule-based	rule-based
	Inserting without Dynamic Partition	Inserting the partition table without setting dynamic partition.	E	rule-based	rule-based
	Disabling Partial Aggregation	Not enabling the function of partial aggregation on the map side.	P	rule-based	rule-based
	Disabling Map Join	Not enabling Map Join when small tables join large tables.	P	rule-based	rule-based
	Disabling Small File Merging	Not enabling the function of automatically merging small files.	P	rule-based	rule-based

* **"P" is for poor performance, "M" for low maintainability, and "E" for program error.**

These are statement APs and configuration APs shown in DeFiHap. AP ID ranges from S-AP-01 to S-AP-14 and from C-AP-01 to C-AP-11, both from top to bottom.

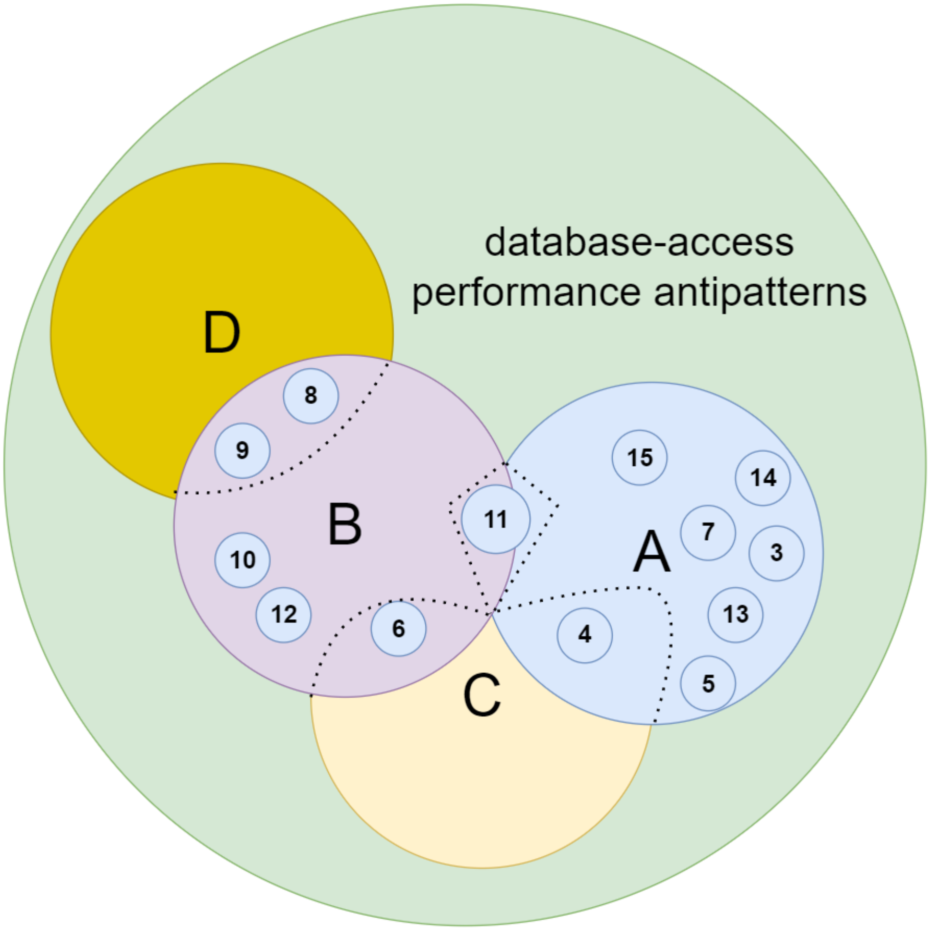
ID	Name	Root cause	Fix strategy
AP-01	Inefficient queries	Issuing queries where semantically equivalent but more performant alternatives exist.	Using the more performant alternatives.
AP-02	Moving computation to the DBMS	Computing with the results of multiple queries, where the computation can also be done by the DBMS, and the network round-trip cost is larger than the query processing cost in the DBMS.	Moving the computation to the DBMS to save the network round-trip cost.
AP-03	Moving computation to the server	Computing some results by the DBMS, which unfortunately is less performant compared with computing by the server, despite the increase in round-trip cost.	Moving the computation to the server despite extra round-trip cost.
AP-04	Loop-invariant queries	Queries issued repeatedly in a loop always load the same database contents and hence are unnecessary.	Moving the query out of the loop and storing the queried results to intermediate objects.
AP-05	Dead-store queries	The results of multiple queries are loaded to the same object, but the object is not used between some of these reloads.	Removing queries whose results are not used.
AP-06	Queries with known results	Issuing queries whose results can be determined by examining the queries and program contexts without actually being executed.	Replacing the queries with the known results.
AP-07	Redundant condition check	Queries issued inside condition checks and branches are identical and return the same results.	Storing the queried results to intermediate objects and using them in both the condition checks and branches.
AP-08	Not caching	Issuing multiple queries that are syntactically equivalent or of the same template without caching the query results.	Adding caching either using a new cache layer or storing the query results in static objects.
AP-09	Inefficient lazy loading	Issuing one query to retrieve N objects from one table, and N other queries to retrieve information related to the N objects from another table.	Issuing one query with a join clause of the two tables.
AP-10	Not merging selection predicates	Issuing multiple `SELECT` queries where each loads only a subset of the needed rows.	Loading all needed rows in one query.

AP-11	Not merging projection predicates	Issuing multiple `SELECT` queries where each loads only a subset of the needed columns.	Loading all needed columns in one query.
AP-12	Inefficient eager loading	Eagerly loading associated objects that are too large.	Delaying the loading of the associated objects.
AP-13	Inefficient updating	Issuing N separate queries to update N database records.	Batching the N update queries into a single query.
AP-14	Unnecessary column retrieval	Retrieving more columns than needed.	Retrieving only the columns that are needed.
AP-15	Unnecessary row retrieval	Retrieving more rows than needed.	Only retrieving the rows that are needed.
AP-16	Unnecessary whole queries	The results of certain queries are completely unused.	Removing the queries.
AP-17	Inefficient rendering	When a view file renders a set of objects, inefficient APIs are used.	Using more performant APIs for view rendering.
AP-18	Missing fields	Fields that are costly to be derived from other fields are not stored directly in database tables.	Storing the fields in database tables directly.
AP-19	Missing indexes	Appropriate indexes are not included in table schema.	Adding the necessary indexes.
AP-20	Table denormalization	Issuing queries with fixed join predicates.	Storing the pre-joined, i.e., denormalized, table based on the fixed join predicates in the DBMS.
AP-21	Partial evaluation of projections	Issuing queries that mostly use a subset of stored fields in a table and the mostly unused fields are much larger in data size.	Partitioning the table column-wise into a table for frequently queried small fields and another for less queried large fields in the DBMS.
AP-22	Partial evaluation of selections	Issuing queries whose selection predicates contain constant values.	Storing table rows matching the predicates with constant values in the DBMS as a separate table.
AP-23	Unbounded queries	Queries returning an unbounded number of records to be displayed.	Pagination, i.e., splitting and displaying records on different pages.
AP-24	Functionality trade-offs	Developers introducing new functionalities that are too costly.	Removing the costly new functionalities.

ID	Name
AP-25	Existing indexes not leveraged
AP-26	Non-optimal force index
AP-27	Changing subqueries to join operations
AP-28	Changing join operations to subqueries
AP-29	Joining unused tables
AP-30	Unnecessary locks
AP-31	Subquery returning duplicated rows
AP-32	Conditions containing subsuming clauses
AP-33	Unnecessary `where` clause when all conditions are selected
AP-34	Unnecessary query construction

These are APs collected from [11]. AP ID ranges from AP-01 to AP-34 from top to bottom. (Detailed description about AP-25 to AP-34 can be found in its section V)

8.2 Venn Diagram



A: ORM Performance Anti-Patterns
[3, 4, 5, 7, 11, 13, 14, 15]
B: SQL Anti-Patterns
[6, 8, 9, 10, 11, 12]
C: Performance Optimization Prioritization
[4, 6]
D: Local Database Performance on Mobile Application
[8, 9]

Changes: All small circles have been placed inside the big circle since they all study database-access performance antipatterns (AP). Compared to the previous version, this version drops 2 papers that are not so relevant to the big topic; [5, 13] have been added to A after carefully reading the literature; [5] has been added to B, too.