2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 978-1-6654-9590-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-SEIP55303.2022.9793914

An Empirical Study on Quality Issues of eBay's Big Data SQL Analytics Platform

Feng Zhu[†,*], Lijie Xu[‡,*], Gang Ma[†], Shuping Ji[§], Jie Wang[#], Gang Wang[†], Hongyi Zhang[†], Kun Wan[†], Mingming Wang[†], Xingchao Zhang[†], Yuming Wang[†], Jingpin Li[†]

[†]eBay Inc. [‡]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[§]University of Toronto [#]Ant Technology Group

[†]{fenzhu, ganma, gwang3, hongyzhang, wakun, mingmwang, xingczhang, yumwang, jingpli}@ebay.com
[‡]xulijie@iscas.ac.cn, [§]shuping@msrg.utoronto.ca, [#]jagger.wj@antgroup.com

ABSTRACT

Big data SQL analytics platform has evolved as the key infrastructure for business data analysis. Compared with traditional costly commercial RDBMS, scalable solutions with open-source projects, such as SQL-on-Hadoop, are more popular and attractive to enterprises. In eBay, we build *Carmel*, a company-wide interactive SQL analytics platform based on Apache Spark. *Carmel* has been serving thousands of customers from hundreds of teams globally for more than 3 years. Meanwhile, despite the popularity of open-source based big data SQL analytics platforms, few empirical studies on service quality issues (e.g., job failure) were carried out for them. However, a deep understanding of service quality issues and taking right mitigation are significant to the ease of manual maintenance efforts. To fill this gap, we conduct a comprehensive empirical study on 1,884 real-word service quality issues from *Carmel*. We summarize the common symptoms and identify the root causes with typical cases. Stakeholders including system developers, researchers, and platform maintainers can benefit from our findings and implications. Furthermore, we also present lessons learned from critical cases in our daily practice, as well as insights to motivate automatic tool support and future research directions.

CCS CONCEPTS

• **Information systems → Data management systems**; • **Software and its engineering → Software reliability**.

KEYWORDS

Big Data, SQL on Hadoop, Open Source, Empirical Study

ACM Reference Format:

Feng Zhu, Lijie Xu, Gang Ma, Shuping Ji, Jie Wang, Gang Wang, Hongyi Zhang, Kun Wan, Mingming Wang, Xingchao Zhang, Yuming Wang, Jingpin Li. 2022. An Empirical Study on Quality Issues of eBay's Big Data SQL Analytics Platform. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513034

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513034

1 INTRODUCTION

SQL analytics platforms play as the fundamental infrastructure for decision support and business intelligence in many organizations. With the ever-increasing data volume, the widely adopted solutions have been evolving from traditional costly commercial RDBMS systems to scalable big data architectures, which have the advantages of scale-out, elasticity and high availability. Over the past few years, a growing number of organizations have turned to rely on Hadoop[1] ecosystem to implement enterprise grade applications.

A common practice is to use Hadoop as the central data repository for different data sources. Many Hadoop-based frameworks are developed to manage and run deep analytics to gain actionable insights, including analysis over semi-structured data, as well as relational-like SQL processing over structured data. For enterprises, SQL processing always gains significant attention, as many data management tools rely on SQL interface and many users are familiar with it. As a result, a number of new big data SQL analytics systems (also known as *SQL-on-Hadoop* [5]) have increased significantly, such as Hive[2] and Spark SQL [1].

Compared with traditional RDBMS-based warehouse solutions, big data SQL analytics platform is distinguished by its fundamentally different architecture and deployment, including the interoperability of multi-layer [11] (e.g., *hardware cluster layer*, *distributed storage layer*, *resource management layer*, *distributed processing layer*), highly distributed, redundant, and elastic data repositories. Rather than being a siloed and centralized data repository like a solitary Oracle, a Hadoop cluster generally consists of anywhere from tens to thousands of nodes. This group of machines works in tandem to appear as a single entity, much like a mainframe, but with much lower capital expense and operating cost. On the other hand, in terms of software maintenance and evaluation, big data SQL analytics platforms exhibit unique characteristics, due to the necessary collaborative activities with open source community.

In eBay, we have built *Carmel*, a company-wide SQL-based analytics platform to serve all of eBay's *interactive* query analysis traffic. *Carmel* is a typical SQL-on-Hadoop based on Spark, the de facto industry standard for big data processing. To guarantee the service quality, we have made plenty of customization efforts, including well-defined incident escalation process, bug fix, performance optimization, tool support, metric monitoring, as well as the close collaboration with Apache Spark community.

*Corresponding authors
[1]Apache Hadoop: https://hadoop.apache.org
[2]Apache Hive: https://hive.apache.org

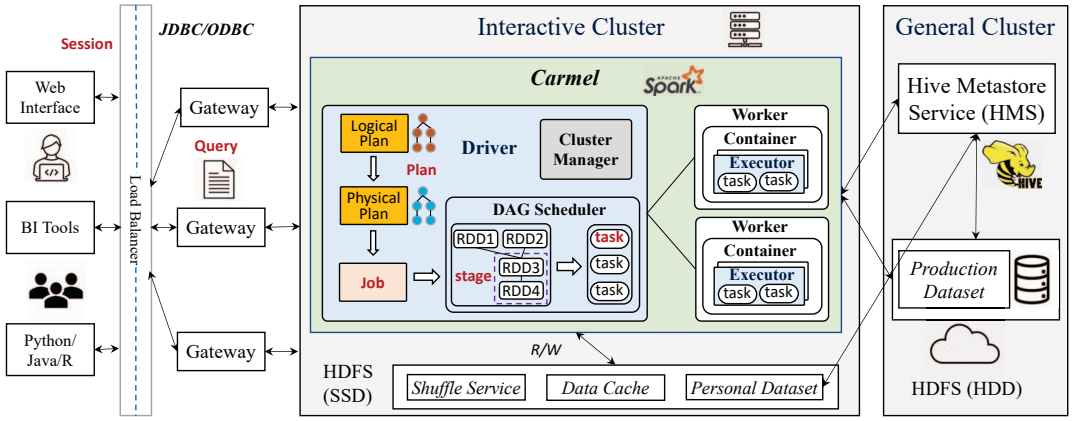

Figure 1: *Carmel* platform architecture and deployment.

Despite the popularity of open-source based big data SQL analytic platforms, to the best of our knowledge, few studies were carried out for them, including the service quality, maintenance and evolution. We believe that it is important to understand the characteristics of service quality issues and root causes, so that stakeholders including system developers, platform maintainers, and researchers can benefit from the findings and implications. To fill this gap, we conduct a comprehensive empirical study on 1,884 tracked issues and try to answer the following research questions.

- **RQ1**: What are the common symptoms of quality issues?
- **RQ2**: What are the common root causes of quality issues?
- **RQ3**: What efforts researchers can do to mitigate the issues?

Contributions. (1) We present the first comprehensive study on service quality issues of open-source based big data SQL analytics platform. Our work can help better understand the issues in similar platforms and provide mitigation guidelines. (2) We perform in-depth root cause analysis and obtain some interesting findings that have rarely been discussed before, such as abnormal query transformation and big data components mismatches. Our findings not only facilitate engineers to diagnose and fix the issues, but also motivate automatic tools for the ease of manual maintenance. (3) We propose 5 research topics (also pain points in practice) on reliability and performance, which require knowledge in software engineering, system architecture, and databases.

2 BACKGROUND: *CARMEL* IN EBAY

In the past, eBay had years of investment in Teradata as its business intelligence solution for company-wide data analytics. However, with the continuously increasing data volume, the solution from a third-party service vendor like Teradata not only exposes limitations on scalability and flexibility, but also becomes more expensive.

History. In 2018, eBay started the *Carmel* project, a big data SQL analytics platform. *Carmel* is based on Apache Spark and aims to undertake the workloads from Teradata. One challenge we ever confronted was how to empower the SQL execution engine with matching performance and stability of the previous system. Though *Carmel* can naturally inherit the scalability and flexibility of Spark, there is a big performance gap between the previous commercial software and the open-source Spark SQL engine. For example, a query with multiple JOIN operators might only execute for a few seconds on Teradata, while the same query could take several minutes on the new SQL-on-Hadoop engine, especially when multiple users execute different queries concurrently. To reduce this gap, we made significant optimizations and customizations, such as data caching, bloom filter join, view-based access controls, dynamic predicates pushing down, range join algorithm and so on. *Carmel* also offers the capacity to integrate with complementary open-source systems to provide better analytics functions and user experience.

Architecture. Figure 1 depicts *Carmel*'s platform architecture. *Gateway* is the platform's access point. It is lightweight and compatible with the Hive thrift protocol. *Gateway* is responsible for client connection, authentication and traffic distribution. Business intelligence tools, such as Tableau and MicroStrategy, can use JDBC and ODBC protocols to connect with *Carmel* through *Gateway* to run SQL commands. Spark adopts the master-slave architecture containing one central coordinator and multiple distributed worker nodes. The central coordinator is called *Driver* and it communicates with all the workers. Each worker node consists of one or more *Executors*. *Carmel* provides service by running many Spark Servers in Hadoop YARN cluster. Organizations in eBay domain have dedicated YARN queues to execute their respective workloads as a way to get rid of resource contention. When a Spark server is started, a number of executors will be allocated and started in the queue. The thrift server and executors are *long-running* services that keep processing all SQL requests coming to that queue. *Carmel* runs in a *Interactive Cluster*, which utilizes SSD disk to speedup data IO performance. The table metadata is stored in a shared Hive metastore, which resides on a separate *General Cluster*. The metadata are accessible by Spark driver and executors.

Workload lifecycle. The workload lifecycle in *Carmel* can be described from high-level session to low-level task. Each connection establishes a **session**, which contains a number of SQL queries. A **query** will be transformed into a **query plan**. The transformation is completed by the parser, analyzer and optimizer. A query plan contains one or more Spark jobs. A **job** can be divided into a number of **stages** based on the shuffle boundary. Each stage can be further divided into a number of **tasks** based on RDD data partitions.

RDD is an abstracted primitive in Spark for efficient distributed computing. A task is the smallest work unit for Spark executors.

Carmel uses about 4,000 cluster nodes to manage and process hundreds of petabytes of data. There are over 1,200 distinct users, who submit about 300,000 SQL queries every day. The execution of 80% queries can be finished within 30 seconds. *Carmel* has already evolved as a fundamental infrastructure in eBay and help saving tens of millions of dollars every year. The platform's outstanding performance is a key factor in the smooth roll out of Hadoop ecosystem across eBay. As we continue leveraging data to power eBay's tech-led re-imagination, building our own in-house solution puts us in the driver's seat for ongoing enhancements and innovations.

3 STUDY METHODOLOGY

We analyze 1,884 real-world service quality issues occurred on *Carmel* in a whole year, a.k.a., from August 2020 to July 2021. It should be clarified that our research focuses on the query engine (i.e., the green colored part in Figure 1), which is the kernel of a big data SQL analytics platform. For those relied components (e.g., Zookeeper), we will not dive into their internal mechanism details. Hence, we also use *Carmel* "system" to refer in particular to the *Carmel*-Spark system in the rest of this paper.

Data Collection. We track all the noteworthy issues as tickets in eBay's JIRA system. These issues could be reported by users, detected by different monitoring tools, identified & promoted by developers, or directly triggered by incident escalation. A JIRA ticket contains all information about an issue, including but not limited to background, priority, labels and comments. We develop scripts to extract and analyze all tickets data through JIRA API. For a small number of complex issues, we also refer to the history emails, slack conversations, and audit logs as complementary information.

Issue Classification & Root Cause Identification. *Carmel* has a suite of auxiliary tools (e.g., monitoring systems, regression testing system, query execution history server, audit log system), and a well-defined escalation process to track and investigate service quality issues on 24/7 basis. For example, *Carmel*'s audit log system pre-defined about 40 error message patterns with frequent keywords (e.g., "IO exception") to automatically extract and categorize job failures. *Carmel*'s monitoring system monitor more than 100 metrics (e.g., execution time, IO inputs, etc.) in real time.

Step 1: Symptom classification. Firstly, it is straightforward to get the major symptoms from auxiliary tools and customer's report, including (1) whether the job runs successfully? (2) is the performance satisfactory? (3) are the query results correct? Then the on-call engineer investigates the audit log system and history server to determine the further detailed classification. Therefore, this step is semi-automatic and conducted in daily maintenance.

Step 2: Labeling and root cause analysis. It is a manual process accomplished by the corresponding on-call engineer in every week. In practice, the root cause analysis of each issue is completed within 3 days. Then the on-call engineer send a mail to team with the issue details, including the issue summary, impact, event timeline, corrective measure and preventive measures.

Step 3: Team review. To reduce bias and misunderstanding, on-call engineers share the issues occurred in previous week and the *Carmel* team (made up of 6 senior and 2 expert software engineers) reviews these issues every week. In case of disagreement, the team will create more in-depth discussions. After that, the team builds weekly reliability report. Moreover, as a necessary maintenance activity for the opensource-based software, *Carmel* engineers keep close interactions with Spark community to ensure the right understanding and fix approaches for those common issues.

Table 1: Platform quality issue symptoms.

Category	Symptom	Count	Ratio
Job Failure	Threshold triggered	209	11.09%
	OOM exception	92	4.88%
	Memory Full GC	68	3.61%
	Parse exception	41	2.18%
	Analysis exception	56	2.97%
	Planning error	87	4.62%
	IO exceptions	244	12.95%
	Task timeout	96	5.10%
	Broadcast failure	61	3.24%
	Other trivial cases	133	7.06%
	Subtotal	**1087**	**57.7%**
Job Slowness	Resource shortage	274	14.54%
	High CPU usage	77	4.09%
	High I/O usage	239	12.68%
	Scheduler stuck	37	1.96%
	Stage & task retry	39	2.07%
	Other trivial cases	38	2.02%
	Subtotal	**704**	**37.36%**
Wrong Result	Incorrect data	**97**	**4.94%**
Total	**1884**		**100%**

Threats to Validity. (1) *Internal threats to validity.* Manual study is indispensable for classification. In this process, subjectiveness would arise during root cause reasoning. For some classifications, we may ignore to highlight some rarely-happen but important issue patterns that fall in the "Other" category. Moreover, some issues may be also ignored to classified into more than one categories. In our study, we minimize subjectivity by considering both JIRA labels, email subjects and slack conversations in complementary. (2) *External threats to validity.* Though *Carmel* is a typical big data SQL analytics system, its deploy mode and workload characteristics may be different from other systems. For example, the dominant workloads for some enterprises are batch-oriented and scheduled periodically, the corresponding issue categories may have different percentages. Besides, other computing engines (e.g., Hive) may adopt different architectures which introduce different issues. In addition, the issues caused by internal features like access control, may be difficult to generalize to other systems.

4 RQ1: SYMPTOMS OF QUALITY ISSUES

According to different symptoms, the platform issues can be classified into three categories, as shown in Table 1.

(1) Job Failure. Many queries fail with exceptions or errors, which can be appeared in different phases and components. As depicted in Figure 1, *Carmel* leverages Spark for query planning and execution. The driver is responsible for parsing, analyzing, optimizing and transforming the query to physical plan, and then schedule

tasks inside the plan, whereas the executors are responsible for running the tasks (one thread per task) in a distributed cluster. Job failures can occur in both driver side and executor side.

The most two prominent job failure symptoms are IO exceptions (12.95%) and threshold exceeded exceptions (11.09%). IO exceptions include a wide range of data loading, writing, and shuffling related issues. For example, heavy data shuffling spills large volumes of data into disk and usually produces disk IO related issues. As a shared service for multi-users, to guarantee the system availability, *Carmel* adds a variety of thresholds in different system modules, including restrictions on data size for join key, broadcast table size, directory item size, download size limits and so on. When these thresholds are triggered, users will get the corresponding error message. The failure symptoms in the query transformation phase, including parsing (2.18%), analyzing (2.97%) and planning (4.62%) are also common and easy to be understood by users. Memory-related failures totally occupy 8.49% percentage, including memory full GC (3.61%) and OOM exceptions (4.88%). Almost all memory full GC occurred in driver side, while OOM exceptions exist both in driver side and executor side. In practice, we also pay much attention to memory-related failures with comprehensive monitoring, diagnosing, and discussion. Moreover, the job can fail as broadcast failures (5.89%), task timeout (5.10%) and other trivial cases.

(2) Job Slowness. Many queries run successfully, but users suffer from its low performance. One scenario is the abnormal long execution time that contradicts common sense, for example, a simple query that counts a table with only a hundred of rows takes nearly one hour. Another scenario is the execution of a recurring query takes relatively long time than before. For example, we often notice users' complains like "*my query can always finish within 10 minutes, why it spends more than 30 minutes today?*"

In the platform side, the most prominent (14.54%) symptom is the resource shortage, especially when the total cluster workload is busy at peak periods (e.g., Monday morning). Too many queries run concurrently, and part of them have no choice to wait for resource allocation. The second most (12.68%) symptom is high IO usage, which covers a wide range of data reading, writing, and shuffling related issues. For example, when a query scans huge tables, we observe explosive IO usage growth. The symptom of high CPU usage is also common (4.09%) in practice. There are CPU intensive functions (e.g., regex pattern match) that involve complex calculations. Meanwhile, system defects (e.g., unimplemented code generation for some operators) can also impact the CPU usage. Besides, job slowness can also manifest as scheduler stuck (1.96%), stage or task retry (2.07%) and other trivial cases.

(3) Wrong Result. Some queries produce wrong results, which can be characterized as incomplete data or incorrect values. Wrong results adversely affect user experience and business, especially those involving payment and finance. Different with job failures and slowness, some wrong results may be identified after several months since they occurred. In our practice, issues falling in this symptom always attract attentions with high priority.

Finding 1: The platform quality issues manifest as more than 15 main symptoms and can be classified into three categories, i.e., *Job Failure* (57.7%), *Job Slowness* (37.36%) and *Wrong Result* (4.94%), indicating the diversity of issues in production environment.

5 RQ2: ROOT CAUSES OF QUALITY ISSUES

After identifying the root causes of these service quality issues listed in Table 1, we get the following finding.

Finding 2: The root causes can be classified into three primary categories: 14.54% issues are caused by *User Side Faults*, 46.02% caused by *System Internal Defects* in *Carmel*-Spark during the query planning and execution, and 39.44% caused by *Platform Component Mismatches* among different open-source components.

5.1 User Side Faults

Each job includes a *SQL query*, several system-related *configurations*, and the dataset to be analyzed. User side faults refer to the faults in these three user-defined items. As detailed in Table 3, user side faults include *SQL anti-patterns* (73.16%), *improper configurations* (20.22%), and others like *platform misuse* (6.62%).

5.1.1 ***SQL Anti-Patterns.*** An anti-pattern (AP) refers to a design decision that is intended to solve a problem, but incurs other problems by violating fundamental design principles. The Anti-Patterns in SQL queries can lead to convoluted logical and physical database designs, thereby leads to job failures or performance degradation.

(1) Wrong code intention. Users wrote "correct" queries with correct grammar but wrong intention, leading to unexpected execution (e.g., data explosion) or results. For such cases (e.g, Figure 2), we interact with users and find that the root causes are caused by users' error-prone programming habits.

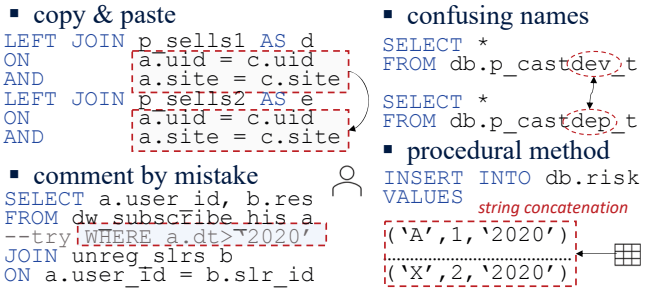

Figure 2: Four typical cases of wrong code intention.

- *Copy & paste.* Users copy and paste similar code snippets in their scripts but they forget to change the table names or column names, leading to wrong results.
- *Confusing names.* Many *table* and *view* (virtual table) names are very similar, users may misuse each other, leading to wrong semantic results or runtime exceptions.
- *Comment clauses by mistake.* Users comment some operators by mistake. For example, the *WHERE* operator was edited mistakenly by a user, which leads to both wrong result and low performance (caused by a large table scan without filter).
- *Procedure programming habits.* SQL language is declarative with relational operators. In case 4, the logic can be simply and efficiently implemented with a SELECT clause. However, the user writes a script to concatenate all rows into the query, generating a 500MB string that is directly rejected by our platform.

(2) Inefficient SQL patterns. Users write inefficient queries that leads to low job performance, as demonstrated in Figure 3.

Table 2: Root cause categories of the quality issues.

	Job Failure	Job Slowness	Wrong Result	Total
User Side Fault	132	115	27	**274(14.54%)**
System Internal Defect	483	338	46	**867(46.02%)**
Platform Component Mismatch	472	251	20	**743(39.44%)**
Total	**1087(57.7%)**	**704 (37.36%)**	**93 (4.94%)**	**1884 (100%)**

Table 3: Root causes of the user side faults.

Root Cause	Count		Ratio
SQL Anti-Pattern	Wrong code intention	39	14.34%
	Inefficient SQL pattern	112	41.18%
	Inefficient data model	32	11.76%
	Non-determinism	16	5.88%
	Subtotal	**199**	**73.16%**
Mis-configuration	Improper parameters	55	20.22%
Other trivial cases	Platform misuse etc.	18	6.62%
Total	**272**		**100%**

CPU-consuming function

```
SELECT CASE
   WHEN get_json_object(s, '$.a')='1'…
   WHEN get_json_object(s, '$.a')='2'…
FROM t
```

IO-consuming query

```
SELECT 'A' AS ID
FROM t WHERE x = 'XA'
UNION ALL
SELECT 'B' as ID
FROM t WHERE x = 'XB'
```

Redundant window computation

```
SELECT a
FROM (
SELECT row_number()…AS rn
……) WHERE rn = 1
```

Redundant join

```
SELECT max(t1.a)
FROM t1 LEFT JOIN
 t2 ON t1.id = t2.id
```

Cartesian product

```
SELECT *
FROM t1 JOIN t2
ON t1.id like t2.id
```

Figure 3: Five typical cases of inefficient SQL patterns.

- *CPU-consuming user-defined functions.* For complex computation, users can utilize UDF (builtin or user-defined functions) in SQL queries, such as the *get_json_object* in case 1 for processing JSON data. When the UDFs are CPU intensive and invoked many times, they will degrade the job performance.
- *IO-consuming query.* Some queries that read the same table multiple times can be combined to reuse table scan to reduce IO cost and avoid generating too many tasks.
- *Redundant computation.* In the query example, only the first record is retrieved after calculating the row number. The redundant computation of window clause degrades performance.
- *Redundant operator.* Similar to the *JOIN* in case 4, some operators are redundant and can be simply removed.
- *Cartesian product.* Joining tables without efficient join condition generates time-consuming Cartesian product. It should be avoided with best rewriting efforts.
- *Redundant data.* Some special values that represent meaningless data in business are not filtered and participate in query processing. Another typical scenario is the condition on different types. For example, "$a.id = b.id$", in which "$a.id$" is string and "$b.id$" is double. When the implicit type coercion fails, it will produce massive *NULL* values, resulting in data explosion and data skew.

Although the query planning tries best to handle more scenarios, there are still cases out of the coverage. For example, the CPU-consuming function pattern is trivial but difficult to be identified.

(3) Non-determinism. Queries may generate inconsistent results at each execution. Apart from some non-deterministic functions like *random()*, current shuffle implementation can also introduce uncertainty in group row order.

```
SELECT a, b
FROM (SELECT row_number()
 OVER (PARTITION BY c ORDER BY a) AS rn FROM t)
WHERE rn = 1
```

For example, the above query first groups rows by column '*c*' and then orders the rows in each group by column '*a*'. It achieves the same result when running on a single machine for many times. However, when running in a cluster for many times, the orders of the rows in each group are different since the rows from different nodes are shuffled into one node by network without order. This unordered shuffle leads to inconsistent results.

(4) Inefficient data model design. To improve the big data management and query performance, *Carmel* provides some advanced data models (formats) for data storage and efficient queries, including partitioned tables, buckets, views, etc. Users sometimes forget to use them or inappropriately use them, leading to low job performance. For example, one user created a table without defining which columns can be partitioned. It may work well when data size is small. However, with the data increase, scanning the whole table reads a large number of files, leading to poor performance.

Finding 3: More than 70% user-side faults caused by SQL anti-patterns, including code smells, low performant queries, ambiguous operators, and inefficient data mode design. For error detection, the difficulty centers on ferreting out wrong business semantics from normal queries with correct syntax (i.e., can be successfully parsed and executed). Besides, the pain point is the lack of tools and guidelines for users, especially those who have less experience.

5.1.2 ***Improper job configurations.*** Built on Spark, users can specify job parameters, including *resource-related configurations* such as the memory sizes of driver and executors, as well as *dataflow-related configurations* like partition number. However, it is difficult to set and tune these configurations, especially before running the jobs, even for experienced engineers. In practice, many users prefer to get recommended parameters from open forums like StackOverflow. In many cases, the "answer" may be not applicable to their specific jobs, leading to job failure or performance degradation.

Case: Out-of-memory error due to improper data broadcast threshold. We have a configuration-related issue occurred in October 2020, when the user submitted a query to join two tables. One is a 700GB large table, whereas another one is a 1GB table.

Table 4: Root causes of system internal defects.

Root Cause	Detailed causes	Count	Ratio
Abnormal & Sub-optimal query plan transformation	Rule issue related	43	4.96%
	Rule execution order	14	1.61%
	Suboptimal plan	144	16.61%
	Inaccurate statistics	29	3.34%
	Codegen related	11	1.27%
	Subtotal	**241**	**27.79%**
Inefficient runtime data management	Input data	82	9.46%
	Intermediate data	155	17.88%
	Output data	15	1.73%
	Subtotal	**252**	**29.07%**
Large metadata	Job & task meta info	**56**	**6.46%**
Internal component design limitation	Shuffle framework	97	11.19%
	Job DAG Scheduler	44	5.08%
	Job Thrift Server	29	3.34%
	Subtotal	**170**	**19.61%**
Other trivial causes	New feature bug, etc.	148	17.07%
Total	**867**		**100%**

Normally, Spark will transform this query to sort-merge based *JOIN* for joining two tables. To improve query performance, the user searched suggestions[3] on Stackoverflow and tried to make it as a broadcast based *JOIN*, based on the assumption of "big table join small table". After enlarging the broadcast threshold to 500MB (note: the default value is 100MB in our platform), Spark diver tried to buffer and broadcast the 1GB table, leading to the OOM error.

Apart from the resource- and dataflow-related configurations, users sometimes specify *wrong client properties*, i.e., account/security/privilege related configurations. For example, users may wrongly configure the resource queue name, certificate key or service account when they interact with the *Carmel* server.

Finding 4: About 20% user-side faults caused by improper user-side configurations. It is error-prone especially when the configurations relate to resource usage or dataflow processing behaviors.

Implication: It is practical to enable the system to anticipate and defend against configuration errors. A model to evaluate the configuration's effectiveness is beneficial to address the challenges.

5.1.3 ***Others.*** There is also a small number (totally 18 issues) of user side faults related to data privileges (e.g., accessing illegal tables), mismatched (ODBC and JDBC) client versions, and platform misuse (e.g., creating a new table on existing table paths).

5.2 System Internal Defects

Carmel leverages Spark system to transform and execute user-submitted jobs (i.e., queries, data, and configurations) in a distributed cluster. In case of computation/data/memory-intensive jobs, some Spark internal components, including job (DAG) scheduler, query planner, and distributed execution runtime framework, can suffer from errors or performance degradation due to internal defects, as depicted in Table 4.

[3]Tuning broadcast threshold: https://tinyurl.com/y2f8rvnw

5.2.1 ***Abnormal/Suboptimal query plan transformation.*** After receiving the SQL query, Spark's query planner (in the driver) will transform the query to a physical plan that contains executable stages and tasks as shown in Figure 1. This query transformation is sophisticated with four steps [1] as follows.

- *Logical planing*: Spark parses SQL as *Unresolved Logical Plan*, and then verifies the plan with catalog (i.e., metastore of entities, e.g., tables, columns etc.) to generate *Analyzed Logical Plan*.
- *Logical plan optimization*: Spark defines a batch of rules and apply them iteratively to derive *Optimized Logical Plan*.
- *Physical planning*: Spark then takes the optimized logical plan and generates *Physical Plan*. During this phase, Spark decides which algorithm must be used for every operator. For example, the decision on which join algorithm must be used, whether sort-merge based *JOIN* or broadcast-based hash *JOIN*, is made at this stage. The cost-based optimization framework[4] leverages data statistics (e.g., row count, max/min values, etc.) to choose the better plan. However, outdated statistics can lead to sub-optimal query plans. AQE[5] (*A*daptive *Q*uery *E*xecution) tackles with such issues by re-optimizing and adjusting query plans based on runtime statistics collected in the process of query execution.
- *Code generation* [8]: Once the best physical plan is chosen, Spark generates Java code that can be compiled and run on each node.

We found that there are a few defects in both the rule-based *logical planning* and *cost-based physical planning*. As a result, the query planner generates abnormal or suboptimal query plans, leading to job failures, performance degradation, or wrong results.

Case 1: Abnormal plan due to the incomplete rule. Optimization rule logic bug generates wrong result. Some optimization rules can bring regressions with incorrect computation logic. For example, we ever picked the implementation to optimize "*IN*" predicates[6] , leading to incorrect result. From the predicate "*a in (1, 2, 4)*", we can infer predicate "$a >= 1 \&\& a <= 4$". But the optimization ignores the reverse situation, i.e., predicate "*a not in (1, 2, 4)*" cannot infer the predicate "$a < 1 \&\& a > 4$".

Case 2: Abnormal plan due to wrong rule order. In a job failure case, the generated *JOIN* plan has different left and right side partition numbers. *EnsureRequirements* rule should be invoked before *RemoveRedundantSorts* rule[7]. The *RemoveRedundantSorts* rule uses SparkPlan's output partitioning information to check whether a sort node is redundant. Since some operators require left and right partitioning to have the same number of partitions, which is not necessarily true before applying the *EnsureRequirements* rule. Consequently, if the *RemoveRedundantSorts* rule is invoked before the *EnsureRequirements* rule, a query can fail with exceptions.

Case 3: Abnormal plan due to rule conflict. Rule conflicts between *PaddingAndLengthCheckForCharVarchar* and *ResolveAggregateFunctions*[8]. *ResolveAggregateFunctions* is a hacky rule and it calls 'executeSameContext' to generate a 'resolved agg' to determine which unresolved sort attribute should be pushed into the 'agg'. However, after we add the *PaddingAndLengthCheckForCharVarchar* rule which will rewrite the query output, thus, the

[4]Cost Based Optimization: https://issues.apache.org/jira/browse/SPARK-16026
[5]AQE: https://issues.apache.org/jira/browse/SPARK-33828
[6]IN optimization: https://issues.apache.org/jira/browse/SPARK-32792
[7]Rule order issue: https://issues.apache.org/jira/browse/SPARK-33472
[8]Rule conflict: https://issues.apache.org/jira/browse/SPARK-34003

'resolved agg' cannot match original attributes anymore. It causes some dissociative sort attributes to be incorrectly pushed in.

Case 4: Sub-optimal plan because DPP rule does not work with AQE. DPP[9] (i.e., Dynamic Partition Pruning) is the optimization to add dynamic partition pruning filter for partitioned table. We ever found about 4 cases, in which DPP can not be applied with AQE, leading to exception by huge scan of sub-optimal plan without partition filter. We reported the issue[10] and fixed it collaboratively.

Finding 5: There are 27.79% issues in system internal defects caused by abnormal/suboptimal query plan. Big data SQL planner suffers from correctness and efficiency problems, including rule's correctness/completeness/integrity guarantee, inefficient cost model, and inaccurate statistical runtime data.

Implication: More comprehensive check on query plan integrity is necessary to ensure the correctness. It is also significant to improve the adaptive policy to cover more scenarios.

5.2.2 ***Inefficient runtime data management.*** Current big data frameworks like Spark use the data parallelism paradigm [22] to execute jobs. The key features of this paradigm are (1) dividing input/intermediate data into small partitions and launching tasks to process them in parallel, (2) leveraging data shuffle mechanisms like *hash-based* shuffle to exchange intermediate data like intermediate computing results among tasks, (3) leveraging in-memory data structures and cache mechanisms to improve the IO performance. Since the runtime input/intermediate/output data of tasks are dynamically changing, unexpected large runtime data can lead to job failures or high resource consumption. Although there are some resource control mechanisms such as spilling large shuffled data onto disk and cache replacement, big data frameworks still suffer when there are unexpected data sizes or distributions.

Case 1: Large input data with massive small files. A small file refers to a data file that is much smaller than the HDFS's default block size (e.g., 128 MB). There are about 15 cases that have millions of small files. These small files are generated from different data sources or the resultant of data processing jobs. Small files not only degrade the job performance due to more disk seeks, but also incur a substantial task scheduling overhead.

Case 2: Large intermediate data with data skew. About 20 cases with *JOIN* queries suffer from OOM errors caused by the data skew. Joining big tables needs to shuffle large data among tasks. Some tasks will receive much more data than others, if there are data skew like skewed key distribution. As a result, these tasks need to aggregate more data in memory and tend to suffer from long execution time or OOM errors. In order to alleviate this problem, Spark's skew join optimization[11] can detect data skew from the statistics of shuffle data and split the skewed partitions into smaller sub-partitions. However, this optimization only works for basic *sort-merge* join, and cannot be used for other cases such as join with bucket tables and join with aggregation.

Case 3: Large output data. Some queries tend to generate large unexpected output data during data processing. For instance, Cartesian *JOIN* queries need to perform Cartesian product on two tables. As a result, each row will be outputted many times, leading to the explosion of output data. In one case, joining one 10MB table with one 15MB table leads to more than 500GB output data, which exceeds the user's space quota size with IO exception.

Finding 6: Nearly 30% of system internal defects are caused by inefficient runtime data management. Big data framework suffers from both memory and I/O pressures brought by *unexpected* large runtime data, such as small files and data skew.

5.2.3 ***Inefficient job metadata management.*** To facilitate job scheduling and monitoring, Spark keeps various job metadata in driver, including lineage information (i.e., the dependencies among tasks), dataflow/resource metrics of each stage and each task, RPC messages, etc. In case of heavy workloads, there will be large volumes of job metadata. Since the driver is a single JVM process, large job metadata will lead to OOM errors, IO exception or job hanging.

Case: large task event metadata caused full GC. In September 2020, we encountered a heavy task event metadata related full GC[12], in which a query held over 100GB memory. Through the heap dump investigation, we found a stage contained dozens of thousands of tasks and each task held thousands of file names. Thus, the scheduler held millions of task event metadata. All accumulator objects will use Spark listener events to deliver to the event loop and even a full GC can not release memory.

Finding 7: Inefficient job metadata management can impact the system's reliability and performance. There are totally 56 cases suffer from large metadata under heavy jobs.

Implication: It is possible to leverage runtime features, approximate algorithms or compression techniques to design memory-efficient data structures. For example, redundant metadata fields can be marked as *soft* or *weak* references in JVM. We can also design distributed schedulers for scheduling large numbers of tasks.

5.2.4 ***Component design limitation.*** Apart from the abovementioned defects, big data frameworks like Spark are also confronted with common problems in system internal component mechanisms that involve scalability, reliability, and performance.

Case 1: Aggressive fault tolerance strategy. We ever encountered 3 cases with a big number (i.e., 3000) for retrying failed tasks, resulting in at least four times slowdown. These cases trouble users whether to kill & re-submit the query or just waiting for the result, and occupies too much resources. After investigation, we found the root cause is Spark's endless retry logic[13]: "*we should ideally differentiate these task statuses so that they don't count towards the failure limit*". That is, several special task types (e.g., speculative commit tasks) may step into endless retry scenarios.

Case 2: Scalability limitations and inefficiency of shuffle framework. We found 93 job slowness issues harmed by heavy shuffle operators, including window clauses, *SORT* and *JOIN* operator. With the rapid growth of data size and scale of deployment, shuffling is becoming a bottleneck of further scaling the infrastructure. Current shuffle framework provides a good balance of fault-tolerance and performance, the fast growth of workloads poses defects on reliability (e.g., all-to-all node connections), IO efficiency (e.g., billions of small random disk reads) and scalability (i.e., the

[9]DPP: https://issues.apache.org/jira/browse/SPARK-11150
[10]DPP with AQE: https://issues.apache.org/jira/browse/SPARK-30186
[11]Skew join optimization: https://issues.apache.org/jira/browse/SPARK-29544
[12]Accumulators full GC: https://issues.apache.org/jira/browse/SPARK-32994
[13]Task retry: https://issues.apache.org/jira/browse/SPARK-8167

Table 5: Root causes of platform component mismatches.

Root Cause	Count	Ratio
Incorrect interface assumption	454	61.10%
Insufficient service federation	158	21.27%
Cross-component mis-configuration	57	7.67%
Non-reproduced	23	3.10%
Others	51	6.86%
Total	**743**	**100%**

reduction of average block size). There are also many efforts on optimizing the shuffle framework, such as Facebook's Casco[14].

Finding 8: There are about 20% of system issues that can be attributed to the architecture design limitations of Spark's internal components, such as scheduler, shuffle service, and etc.

Implication: Fault tolerance, scalability, and performance are popular and long-standing topics in big data frameworks. There is no universal solution for all scenarios. It is more practical to combine platform-specific characteristics (e.g., workload distribution, data properties, etc.) with advanced techniques to improve the scalability, fault-tolerance and performance of the platform.

5.3 Platform Component Mismatches

Carmel glues multiple open-source software components together and orchestrates them as a big data SQL analytics platform. It relies on multi-layer services. Any mismatches among them, as depicted in Table 5, can lead to runtime errors.

5.3.1 ***Incorrect interface assumption.*** Interfaces are the main bridges among different components to integrate. When the workloads in production environment violates the interface assumptions, component mismatch issues occurred.

Case 1: The lack of corner case handling. Spark blacklist logic cannot perfectly handle the case of YARN RM restart. There were more than 50 jobs failed due to the YARN resource manager restart. After investigation, we found the Spark's check logic problem would add all worker nodes into the blacklist, but actually there's no black listed nodes. However, the current solution[15] in Apache Spark community is still a partial alleviation.

> *"Adding double check *** can not actually resolve the issue. When active RM switches *** will be less than some small value. If it equals to like 1, then it will trigger the *** check."*
> *"Not sure what the status of that PR is, so I will merge this and attach to the JIRA as a partial fix."*

Case 2: Unlimited space quota assumption. More than 40 queries failed with "not enough space" when writing data, even user's space quota is enough. Spark calls Hadoop's output interface[16] to write data into HDFS. When writing data, each task will pre-request block-sized (e.g., 256MB) storage with replication factor. For example, 10,000 tasks will try to acquire $256MB * 3 * 10000 = 7.3TB$, even when the real result size is only 1MB. Consequently, if the directory remaining space is less than 7.3TB, the query will fail with the "fake" quota exceeded exception. However, space quota management is common in production, violating of the assumption of "infinite space of user's data directory".

Finding 9: The majority (61.10%) of the platform component mismatch issues are caused by incorrect interface assumptions, including corner cases, restrictions, data presentation (e.g., encoding scheme and case sensitivity) and so on. Whether the interface assumption is suitable for production environment is not well anticipated in open source software.

Implication: Extensive integration testing with efficient approaches to narrow down the state space are essential to find out the incorrect assumptions about the meaning, units, or boundaries of the data being passed between components.

5.3.2 ***Insufficient service federation.*** There are 158 (21.27%) issues caused by component service down. Among these cases, most errors (93) are related to Hadoop service, including NameNode failure, YARN resource manager restart, etc. There are 25 errors caused by Hive metastore and 9 errors caused by Zookeeper. There are also 5.06% MySQL-related errors. The remaining issues are caused by other inner-built components in eBay.

Although platform environment errors generally lead to a great quantity of failures, most of them can be monitored and fixed quickly. In May 2021, we encountered an incident with the highest priority. The HMS service blocked nearly 1,000 queries for more than 6 hours. Although we had two HMS servers for balance, one was crashed and the other was stuck due to heavy read and write access. This impressive incident urged us to realize the importance of service federation. We have now built up 10 HMS servers.

5.3.3 ***Cross-component mis-configuration.*** Configuration mismatches are among the most urgent but thorny problems in software reliability. Improper configuration across different components in *Carmel*, such as HDFS data storage, can lead to job failures, performance degradation, and wrong results.

Case 1: Component mis-configuration leads to job failures. In December 2020, Hadoop HDFS developers updated the configurations for HDFS mount point table, leading to the failures of more than 100 Spark jobs in *Carmel*. After investigation, we found that when writing data, *Carmel* relies the Hadoop's *rename* operation to move data between HDFS paths. This operation is only allowed within the same mount point, conflicting with the newly updated mount point configuration.

Case 2: Risk Hadoop configuration that produces incompleteness result. Some configurations are used to enable performance optimizations while sacrificing data correctness to a certain extent, such as the file output committer algorithm[17] during writing data. If it is configured as version V2, each task will move output into the final directory concurrently, saving a lot of time for the driver when job is committing. However, if the job fails, partial data will be inserted, leading to an incomplete result.

Finding 10: About 7.67% of the platform component mismatch issues are caused by cross-component mis-configurations.

Implication: The mis-configuration problem becomes more prominent in big data ecosystem, in which components are configurable with hundreds of knobs.

[14]Casco: https://tinyurl.com/t73f3rmj
[15]YARN RM restart: https://issues.apache.org/jira/browse/SPARK-29683
[16]SQLHadoopMapReduceCommitProtocol: https://tinyurl.com/fkyb44y9
[17]Risk configuration: https://issues.apache.org/jira/browse/SPARK-36121

5.3.4 ***Non-reproduced.*** Some issues are difficult to reproduce for root cause diagnostic due to "elastic" or "dynamic" mechanisms. For example, YARN allocates resource containers for Spark executors and recycles them again after executor tasks finished. Consequently, even for the same job, the runtime generally differs on data splits, cluster nodes and hardware environments. We ever encountered a confusing JVM fatal error about memory crush, the detailed logs of which are deleted by YARN container. We also failed when try to reproduce it with more than 100 times stimulated workloads. There is a similar case[18] discussed in community. However, the ticket is also closed and marked as "*Cannot Reproduce*".

> *"Thanks for the bug report, but I think we couldn't move it forward because no developer could reproduce this issue based on the given information. So, I think the reporter needs more work to narrow down the issue and make other developer understood. At least, we need a simple query and test data to reproduce it."*

Finding 11: The elastic architecture advocated by big data ecosystem (especially the resource management systems like YARN) can also make some issues (totally 23) difficult to be reproduced.

5.3.5 ***Others.*** There is also a small number (totally 51 issues) of platform component mismatch related with component privilege (e.g., HDFS directory owner change), cluster environment (e.g., network sentinel rules), and etc.

6 RQ3: LESSONS AND INSIGHTS

From the root causes, we can see that there are many research opportunities to improve the reliability and performance of big data analytics platform like *Carmel*. We summarize our lessons with pain points and insights for the future research directions.

(1) SQL anti-pattern detector for big data. Section 5.1 indicates that users may unknowingly introduce anti-patterns (e.g., improper data model usage, etc.) that violate fundamental design principles. In practice, it is manual-intensive and time-consuming to diagnose and fix these anti-patterns. For example, we spend plenty of time to "guess" wrong code intention cases, analyze query intention and interact with users. Despite of many efforts like *SQLCheck* [3], there is not any tools for big data analytics platform in production, which share unique characteristics in data splitting, data model, distributed execution, and etc. For the ease of system maintenance, an automatic and intelligent anti-pattern detector can free both users and engineers from these pain points.

(2) Effective query plan transformation. Section 5.2 demonstrates the performance and correctness problems in the query planner. For the correctness (e.g., completeness, conflict, etc), there is no feasible approach or practice in industry for rule semantic expression and validation. Current practice relies on developers' professional knowledge. A feasible potential solution is to enforce more correctness validations. The performance of the plan is decided by query optimizers, the most popular research topic in database area. Current sophisticated approaches like XX system [12] advocate the direction of leveraging more adaptive policies that are learned systematically from the data and workloads. It is possible to take advantage of historical execution metrics and design advanced dynamic models with machine learning techniques.

[18]JVM fatal error: https://issues.apache.org/jira/browse/SPARK-29767

(3) Automatic configuration. Section 5.1 and Section 5.3 reveal the mis-configuration pains in user-side and platform cross-component. How to quantitatively evaluate the impact of configuration change and guide the configuration tuning are long-standing problems. The main challenge is that the runtime dataflow and resource usage are dynamically changing and determined by many factors such as data properties, resource usage, etc. In practice, we restrict some resource-related parameters in user side. As for the platform side, the cross-component configuration maintenance relies heavily on developer's experience. A potential solution is to build a dynamic model to quantify the relationship among the configurations, dataflow, and resource usage. Dynamic means that the model needs to capture the runtime varying dataflow information and resource usage for model accuracy.

(4) Efficient data and resource management. In Section 5, we can see that large runtime data and resource management defects can lead to job failures and slowness. Current data and resource management policies are simple and threshold-based. For example, if the shuffled data in the memory achieves the threshold, Spark will spill it onto disk. High threshold raises the risk of memory-related issues like OOM, while a low threshold brings heavy burden on disk I/Os. How to effectively manage the runtime data as well as the resource is a practical problem. The main challenge lies in the trade-offs among performance, reliability, and fairness. According to the root cause analysis, it is possible to design more advanced data and resource managers such as memory-efficient data structures for intermediate/output/cached data, common data skew handlers, more precise memory usage estimator and predictor [16], etc.

(5) Advanced replay and diagnosis tools. Big data platform's flexible architecture determines that the issues can occur among multi-layers and multi-components, aggravating the difficulties of root cause identification and issue fix. The challenges lie in: (1) necessity of deep knowledge on all the layers, (2) the lack of reproducible mechanism (e.g., the error in Section 5.3.4). In our daily maintenance activity, the pain point is that more than 30% system internal errors (e.g., full GC) require developers to trace back from low-level abnormal items (e.g., dumped memory objects) to high-level query. We need more advanced replay and diagnosis framework that enables horizontally tracking among different components, and vertically tracing the errors back to wrong operators, improper configurations, and etc.

7 RELATED WORK

Empirical studies on big data systems. Over the years, there have been many empirical studies on data-intensive scalable computing systems (DISC). Dinu and Ng [4] analyzed Hadoop behavior under failures of compute nodes and found that a single failure can result in unpredictable system performance. Xiao et al. [17] conducted a study on commutativity, nondeterminism, and correctness of data-parallel programs and revealed interesting findings that non-commutative reduce functions lead to five bugs. Ding et al. [21] investigated 198 failures occurred on Cassandra, HBase, HDFS, MapReduce, and Redis. They found that almost all failures require only 3 or fewer nodes to reproduce, and the majority of catastrophic failures could easily be prevented by performing simple testing on

the error handling code. Li et al. [10] studied the failure characteristics of Microsoft Scope jobs, and revealed that exceptional data and mismatched data schema are the major source of job failures, rather than code logic. Xu et al. [18] studied 123 real-world OOM errors in Hadoop and Spark applications. Their findings revealed that most OOM errors (64%) are caused by memory-consuming user code. They designed a memory profiler [19] and two types of quantitative rules to diagnose OOM errors. The work of Zhou et al. [23] is most closely to ours. Among the 210 real service escalations of Microsoft big data system, they found 21.0% issues were caused by hardware faults, 36.2% are caused by system side defects, and 37.2% are due to customer side faults. They also discussed practical mitigation solutions such as refining customer code. However, our study is not limited to commercial systems and provides insights that have not been discussed in their work, such as SQL anti-patterns, query transformations, and distributed execution.

Failure diagnosis and fixes. Researchers have also studied failure mitigation approaches, including debugging, profiling, tuning and testing tools for big data systems. BigDebug [6] simulates breakpoints and on-demand watchpoints to facilitate users to pinpoint a crash-inducing record and determine the root causes of errors. BigTests [7] adopted white-box testing approach to reason about the internal semantics of UDFs in data-intensive scalable computing system (DISC). Their evaluation shows DISC applications are often significantly skewed and inadequate in terms of test coverage. The tool can minimize test data to achieve interactive and fast local testing big data analytics. Bertty et al. [2] proposed TagSniff model based on tag and sniff primitives, to simplify data debugging for dataflows. They also developed a tool called Snoopy to support both online and post-hoc debugging modes. Hui et al. [9] summarized six cache-related bug patterns and proposes *CacheCheck* to automatically detect cache-related bugs by analyzing the execution traces in Spark applications. To detect DBMS bugs, Rigger and Su devised a series of novel approaches, including PQS [15], NoRec [13] and TLP [14]. We can leverage and extend these testing approaches to help us discover data quality issues. *TaintStream* [20] automatically injects taint tracking logic into the data processing scripts. It achieves accurate cell-level taint tracking with a precision of 93.0% and less than 15% overhead.

8 CONCLUSION

This paper presents a comprehensive study on the service quality issues of eBay's big data SQL analytics platform. We demonstrate the detailed issue symptoms and identify the main root causes, including user side faults, system internal defects and platform component mismatches. We further discuss lessons learned and insights. We believe our work can not only help alleviate the pains in big data platform maintenance, but also inspire research interests and motivate candidate research directions.

9 ACKNOWLEDGMENT

We are grateful to all the anonymous reviewers for their insightful feedback. We also thank Hongjiang Zhang, Yang Jiang, Lantao Jin and Yanghong Zhong for their work in building *Carmel*. Lijie Xu is supported by the Youth Innovation Promotion Association at CAS.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, and et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD)*. 1383–1394.
[2] Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, and et al. 2019. Tagsniff: Simplified big data debugging for dataflow jobs. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*. 453–464.
[3] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. 2020. SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*. 2331–2345.
[4] Florin Dinu and TS Eugene Ng. 2012. Understanding the effects and implications of compute node related failures in hadoop. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. 187–198.
[5] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. 2014. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1295–1306.
[6] Muhammad Ali Gulzar, Matteo Interlandi, and et al. 2016. Bigdebug: Debugging primitives for interactive big data processing in spark. In *38th International Conference on Software Engineering (ICSE)*. IEEE, 784–795.
[7] Muhammad Ali Gulzar, Shaghayegh Mardani, and et. al. 2019. White-box testing of big data analytics with complex user-defined functions. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 290–301.
[8] Kazuaki Ishizaki. 2019. Analyzing and optimizing java code generation for apache spark query plan. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 91–102.
[9] Hui Li, Dong Wang, and et al. 2020. Detecting cache-related bugs in Spark applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 363–375.
[10] Sihan Li, Hucheng Zhou, Haoxiang Lin, and et al. 2013. A characteristic study on failures of production distributed data-parallel programs. In *35th International Conference on Software Engineering (ICSE)*. IEEE, 963–972.
[11] Sourav Mazumdar and Subhankar Dhar. 2015. Hadoop as Big Data Operating System–The Emerging Approach for Managing Challenges of Enterprise Big Data Platform. In *First International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 499–505.
[12] Parimarjan Negi, Matteo Interlandi, and et al. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 2557–2569.
[13] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1140–1152.
[14] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[15] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 667–682.
[16] Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. 2021. AutoExecutor: Predictive Parallelism for Spark SQL Queries. *Proc. VLDB Endow.* 14 (2021), 2855–2858.
[17] Tian Xiao, Jiaxing Zhang, and et al. 2014. Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In *36th International Conference on Software Engineering (ICSE)*. 44–53.
[18] Lijie Xu, Wensheng Dou, and et al. 2015. Experience report: A characteristic study on out of memory errors in distributed data-parallel applications. In *26th International Symposium on Software Reliability Engineering (ISSRE)*. 518–529.
[19] Lijie Xu, Wensheng Dou, Feng Zhu, Chushu Gao, Jie Liu, and Jun Wei. 2018. Characterizing and diagnosing out of memory errors in MapReduce applications. *Journal of Systems and Software* 137 (2018), 399–414.
[20] Chengxu Yang, Yuanchun Li, Mengwei Xu, Zhenpeng Chen, Yunxin Liu, Gang Huang, and Xuanzhe Liu. 2021. TaintStream: fine-grained taint tracking for big data platforms through dynamic code translation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 806–817.
[21] Ding Yuan, Yu Luo, and et al. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 249–265.
[22] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. [n.d.]. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, April 25-27, 2012*. 15–28.
[23] Hucheng Zhou, Jian-Guang Lou, and et al. 2015. An empirical study on quality issues of production big data platform. In *37th International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 17–26.