Demo of VisBooster: Accelerating Tableau Live Mode Queries Up to 100 Times Faster

Qiushi Bai, Sadeem Alsudais and Chen Li

University of California, Irvine, CA 92697, USA

Abstract

We propose a middleware-based query rewriting framework called VisBooster to accelerate visualization queries formulated by Tableau in its live mode. VisBooster intercepts SQL queries by customizing JDBC drivers used by Tableau and uses rules to rewrite the queries to semantically equivalent yet more efficient queries. The rewriting rules are designed by data experts who analyze slow queries and apply their domain knowledge and optimization expertise. We demonstrate that VisBooster can accelerate visualization queries formulated by Tableau up to 100 times faster.

Keywords

tableau, visualization, middleware, query rewrite

1. Introduction

As a powerful way for users to gain insights from data quickly and intuitively, visualization is becoming increasingly important in the Big Data era. With rich functionalities, Tableau [1] is one of the most popular tools adopted by data analysts to visualize data [2]. Tableau provides two modes called *Extract mode* and *Live mode* [3] for users when connecting to data sources. The extract mode [4] stores a snapshot of the data once and updates the snapshot periodically. The live mode uses a JDBC (Java Database Connectivity) driver to connect to a database and supports in-situ data visualization by formulating SQL queries executed by the database. Often users prefer the live mode for several reasons. As an in-situ visualization paradigm, this mode does not have a long delay for the initial data-extraction process before the users consume the data. Since every frontend interaction using Tableau is a query to the underlying database, users will see the latest results. Also, with the computation happening inside the database, users do not need much additional computing resources for the visualization task. However, the notoriously poor performance of the live mode is a main obstacle preventing users from enjoying these various benefits [3]. In particular, Tableau purely relies on the database to answer queries, and the database cannot guarantee the best performance for all queries due to the difficulty of query optimization [5]. In our experiments (as discussed in Section 4), the query latency can be as much as tens of seconds, which significantly impairs the productivity of users [6].

In this demo, we show a middleware-based query-rewriting framework called VisBooster to solve this performance problem. As shown in Figure 1, VisBooster is between the Tableau application and a backend database. It treats both layers as black boxes, so it requires no code modification to them. It is especially useful in the cases where such changes are not possible. By replacing the original JDBC driver provided by the database vendor with a lightly customized driver, VisBooster intercepts SQL queries formulated by Tableau, and uses rules to rewrite them to semantically equivalent yet more efficient SQL queries. To develop rewriting rules for a given database, VisBooster adopts a human-involved approach. First, a database expert analyzes slow queries to identify rewriting rules, then introduces these rules to a query rewriter. The rewriter then applies these rules on new queries formulated by Tableau. In this way, the database expert continuously improves the performance of queries by identifying more slow queries and introducing new rewriting rules. As an example, we use VisBooster to visualize 30 million tweets on top of a PostgreSQL database using a commodity laptop. For specific visualization queries formulated by the Tableau live mode, VisBooster makes these queries up to 100 times faster. In particular, for a query that takes 30 seconds originally, VisBooster generates a new query that runs within 0.3 seconds.

2. Related Work

Recently, various techniques have been proposed to accelerate visualization queries.

Pre-computation-based approaches. Datacube techniques [7, 8] that predefine cube intervals cannot support visualization queries with arbitrary numerical range con-

Published in the Workshop Proceedings of the EDBT/ICDT 2022 Joint Conference (March 29-April 1, 2022), Edinburgh, UK

qbai1@uci.edu (Q. Bai); salsudai@uci.edu (S. Alsudais); chenli@ics.uci.edu (C. Li)

https://qiushibai.wordpress.com/ (Q. Bai); https://chenli.ics.uci.edu (C. Li)

0000-0001-8736-5042 (Q. Bai); 0000-0003-3928-690X (S. Alsudais); 0000-0001-8015-6870 (C. Li)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3135/bigvis_short5.pdf

ditions, while VisBooster does not have such restrictions on query shapes. View-based techniques such as [9, 10] that utilize materialized results to accelerate queries require additional storage overhead. Prefetching-based techniques such as [11, 12] utilize cached results to accelerate visualization queries. VisBooster is orthogonal to these techniques, and it can be used together with them to further improve query performance.

Query rewriting-based approaches. Bao [13] is a recent technique that learns to choose a good hint for a query to accelerate query execution. The proposed VisBooster framework is not limited to any specific type of rewriting rules. For rewriting rules that involve only query hints, VisBooster can also use Bao as part of the rewriting process. In addition, database vendors provide query rewriting functionalities (e.g., PostgreSQL [14] and MySQL [15]). These functionalities allow users to define their own rewriting rules, and the database will automatically use these rules to rewrite queries. However, these functionalities only support cases where a materialized view can be used, while the proposed VisBooster can support more rewriting cases where a more efficient plan is desired.

Compared to these existing techniques, VisBooster has the following uniqueness:

- It adopts a human-in-the-loop approach that fully leverages human intelligence, including domain-specific knowledge and database-optimization expertise.
- It treats both the application (Tableau) and the backend database as black boxes and requires no code modification. It can significantly improve the user experience in the Tableau live mode without changing the existing software and services.
- It is a general framework that supports many types of rewriting rules such as predicate transformation, query hints, and etc.

3. VisBooster Overview

Figure 1 shows the architecture of VisBooster. Using the Tableau live mode on top of a database, a frontend user interacts with a dashboard of visualization worksheets (e.g., line charts, bar charts, scatterplots, choropleth maps, etc.) to analyze data. For each frontend interaction (such as clicking, dragging, and zooming), Tableau formulates a SQL query (denoted as Q) for each visualization worksheet and sends the query to the database through a JDBC driver (step 1). Without VisBooster, the JDBC driver just forwards the original query Q to the database. The database executes the query and returns the result back to the JDBC driver (step 4). Tableau then consumes the result and produces the final visualization (step 5).

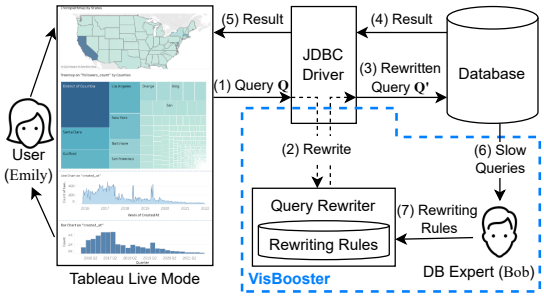

Figure 1: Overview of the VisBooster framework.

With VisBooster, after step 1, the modified JDBC driver invokes the Query Rewriter to rewrite Q to a semantically equivalent but more efficient SQL query, denoted as Q' (step 2). The new query Q' is sent to the database for execution (step 3). A challenge of the rule-based query rewriter is that for a given query Q, it needs to decide what rewriting rules should be applied to generate a new query Q'. In the VisBooster framework, a database expert analyzes the log of slow queries from the backend database (step 6). By using the knowledge about the database, the expert identifies rewriting rules and adds them to the query rewriter (step 7). With the rewriting rules applied to new visualization queries and more slow queries identified by the expert, the framework can iteratively improve the performance of queries.

4. Demonstration Scenarios

In this section, we present several scenarios to demonstrate the efficacy of the VisBooster framework for various visualization queries on different databases, including PostgreSQL and MySQL.

4.1. Case 1: Accelerating choropleth map queries on PostgreSQL

In case 1, suppose Emily is a data analyst who wants to study the temporal and spatial distributions of social media discussions about `iPhone` using a table of 30 million tweets. She builds a dashboard in Tableau connected to PostgreSQL in the live mode.

The dashboard in Figure 2 consists of three modules. On the top is a textual filter that allows users to input a keyword and filters tweets containing the keyword as a substring in their texts. A state-level choropleth map in the middle shows a geo spatial distribution of the filtered tweets grouped by states. A quarter-level bar chart at the bottom shows a temporal distribution of the filtered tweets grouped by quarters. Both the choropleth map and the bar chart change as the user further explores the results. For example, after the user types in `iphone`

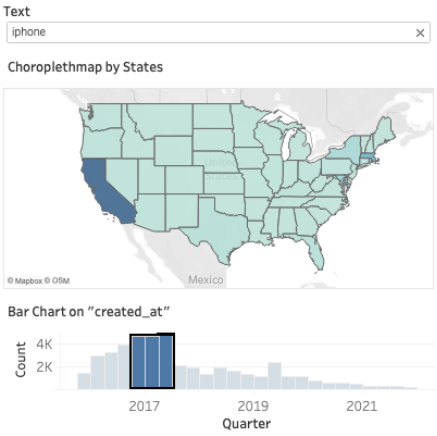

Figure 2: An example dashboard (including a text filter, a choropleth map, and a bar chart) on a Twitter dataset using PostgreSQL.

in the input box, the bar chart's first three quarters in 2017 attract Emily's attention due to their higher number of results than other quarters. She selects those quarters in the bar chart to zoom into the tweets published within the time range. As Figure 2 shows, tweets are selected using two conditions: text contains `iphone` and published within the first three quarters of 2017. The choropleth map is redrawn based on the newly filtered results. Tableau formulates a corresponding SQL query for each interaction and sends it to the database to compute the visualization result. The corresponding query for the current choropleth map is shown in Figure 3.

```
SELECT SUM(1) AS "cnt:tweets",
       CAST("state_name" AS TEXT) AS "state_name"
FROM "tweets"
WHERE CAST( DATE_TRUNC('QUARTER',
                  CAST("created_at" AS DATE))
        AS DATE) IN (
          (TIMESTAMP '2017-01-01 00:00:00.000'),
          (TIMESTAMP '2017-04-01 00:00:00.000'),
          (TIMESTAMP '2017-07-01 00:00:00.000'))
AND STRPOS(CAST(LOWER(
                  CAST(CAST("text" AS TEXT) AS TEXT))
              AS TEXT),
              CAST('iphone' AS TEXT) ) > 0
GROUP BY 2;
```

Figure 3: Original query Q_1 for the choropleth map. (Red shows bottleneck clauses.)

Rewriting rule 1: Removing type casting. Although there is a B+ tree index on the attribute `created_at` in the filtering expression `DATE_TRUNC('QUARTER', "created_at")`, PostgreSQL generates a physical plan that does a sequential scan instead of using the index, resulting in a long execution time of 32 seconds, as shown in Figure 5(a). The long delay of the dashboard queries significantly impairs Emily's productivity, so she asks a database expert, Bob, for help.

After analyzing the original query sent to PostgreSQL and the physical plan, Bob realizes that those type-casting expressions (e.g., `CAST("created_at" AS DATE)`) prevent PostgreSQL from choosing a more efficient index-scan plan. The reason why Tableau adds those type-casting expressions is to prevent computational overflow errors [16]. However, in this case, with the knowledge about the underlying data, Bob can remove those type-casting expressions without worrying about such failures. Thus, he introduces the following rewriting rule:

$$\text{Rule-1: } \texttt{CAST(}\ \langle exp \rangle\ \texttt{AS}\ \langle type \rangle\ \texttt{)} \Rightarrow \langle exp \rangle\,.$$

After applying this rule on Q_1, we have the new query Q_1' in Figure 4.

```
SELECT SUM(1) AS "cnt:tweets",
       CAST("state_name" AS TEXT) AS "state_name"
FROM "tweets"
WHERE DATE_TRUNC('QUARTER', "created_at") IN (
          (TIMESTAMP '2017-01-01 00:00:00.000'),
          (TIMESTAMP '2017-04-01 00:00:00.000'),
          (TIMESTAMP '2017-07-01 00:00:00.000'))
  AND STRPOS(LOWER("text"), 'iphone') > 0
GROUP BY 2;
```

Figure 4: The rewritten SQL query Q_1' with Rule-1 applied on Q_1. (Blue shows the modifications.)

As expected, PostgreSQL generates a much more efficient plan using the B+ tree as shown in Figure 5(b), which takes 10 seconds to compute the results.

Rewriting rule 2: Replacing substring match with LIKE. Seeing the 10s response time, Emily is still not satisfied with the performance for the interactive visualization frontend. After a closer look at the available indexes, Bob finds that a trigram index on the "`text`" attribute in PostgreSQL supports wildcard filtering predicates such as `LIKE` and `ILIKE`. However, PostgreSQL fails to use this index because the wildcard predicate formulated by Tableau is `STRPOS(LOWER("text"), 'iphone') > 0`, which is equivalent to "`text` `ILIKE` `'iphone'`. To address this issue, Bob introduces another rewriting rule:

$$\text{Rule-2: } \texttt{STRPOS(LOWER(}\ \langle exp \rangle\ \texttt{)}, \texttt{'}\langle literal \rangle\texttt{'}\ \texttt{)} > 0 \Rightarrow \langle exp \rangle\ \texttt{ILIKE '\%}\langle literal \rangle\texttt{\%'}.$$

With both rules applied to Q_1, we obtain a new rewritten query Q_1'' shown in Figure 6. For the new query, PostgreSQL chooses to use both indexes as shown in

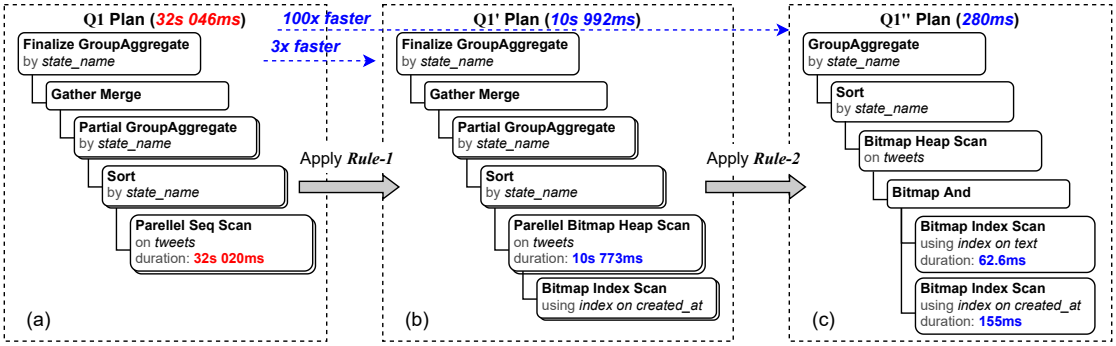

Figure 5: Query plans and execution times of (a) the original Tableau formulated SQL query Q_1, (b) the rewritten query Q_1' by applying Rule-1, and (c) the rewritten query Q_1'' by applying both Rule-1 and Rule-2. Q_1'' is semantically equivalent to Q_1 with a 100 times faster performance. (Red shows Q_1's bottleneck and blue shows Q_1' and Q_1'''s improvement.)

```
SELECT SUM(1) AS "cnt:tweets",
       CAST("state_name" AS TEXT) AS "state_name"
FROM "tweets"
WHERE DATE_TRUNC('QUARTER', "created_at") IN (
        (TIMESTAMP '2017-01-01 00:00:00.000'),
        (TIMESTAMP '2017-04-01 00:00:00.000'),
        (TIMESTAMP '2017-07-01 00:00:00.000'))
  AND "text" ILIKE '\%iphone\%'
GROUP BY 2;
```

Figure 6: The rewritten SQL query Q_1'' with both Rule-1 and Rule-2 applied on Q_1. (Blue shows the modifications.)

Figure 5(c). It takes 0.28 seconds to compute the results, which is 100 times faster than the original Q_1.

Rewriting Rule 3: Adding hints. Emily continues to explore the dataset with more interactions. She notices that the iphone-related tweets present a very low frequency in recent months. To further investigate the reason, she expands the bar chart to a monthly level and selects the recent 7-month bars in late 2021 as a new temporal filter (shown in Figure 7).

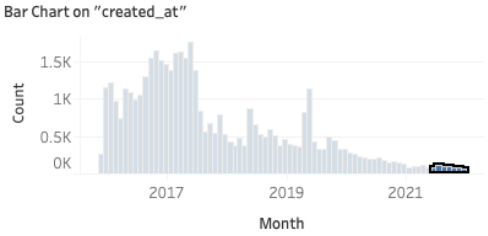

Figure 7: A temporal filter that selects tweets published within the last 7 months in 2021.

The corresponding SQL query is Q_2. Figure 8 shows the rewritten query Q_2'' after applying both Rule-1 and Rule-2 to Q_2.

Interestingly, PostgreSQL only uses the trigram index

```
SELECT SUM(1) AS "cnt:tweets",
       CAST("state_name" AS TEXT) AS "state_name"
FROM "tweets"
WHERE DATE_TRUNC('MONTH', "created_at") IN (
        (TIMESTAMP '2021-06-01 00:00:00.000'),
        (TIMESTAMP '2021-07-01 00:00:00.000'),
        (TIMESTAMP '2021-08-01 00:00:00.000'),
        (TIMESTAMP '2021-09-01 00:00:00.000'),
        (TIMESTAMP '2021-10-01 00:00:00.000'),
        (TIMESTAMP '2021-11-01 00:00:00.000'),
        (TIMESTAMP '2021-12-01 00:00:00.000'))
  AND "text" ILIKE '\%iphone\%'
GROUP BY 2;
```

Figure 8: The rewritten SQL query Q_2'' with both Rule-1 and Rule-2 applied on Q_2. (Blue shows the modifications.)

```
/*+ BitmapScan(tweets idx_tweets_monthly_created_at) */
SELECT SUM(1) AS "cnt:tweets",
       CAST("state_name" AS TEXT) AS "state_name"
FROM "tweets"
WHERE DATE_TRUNC('MONTH', "created_at") IN (
        (TIMESTAMP '2021-06-01 00:00:00.000'),
        (TIMESTAMP '2021-07-01 00:00:00.000'),
        (TIMESTAMP '2021-08-01 00:00:00.000'),
        (TIMESTAMP '2021-09-01 00:00:00.000'),
        (TIMESTAMP '2021-10-01 00:00:00.000'),
        (TIMESTAMP '2021-11-01 00:00:00.000'),
        (TIMESTAMP '2021-12-01 00:00:00.000'))
  AND "text" ILIKE '\%iphone\%'
GROUP BY 2;
```

Figure 9: The rewritten SQL query Q_2''' with all three rules applied on Q_2. (Blue shows the modifications.)

on `text` in the physical plan but fails to use the intersection of both indexes on both `text` and `created_at`, resulting in a long execution time of 6.5 seconds. After an empirical study, Bob finds a pattern that whenever the number of operands in the `IN` operator for the filtering condition on the `created_at` attribute exceeds six, Post-

greSQL always picks one of the two available indexes for scanning instead of doing an intersection after two index scans. In addition, for the specific Q_2, using the B+ tree index on `created_at` is more efficient than using the index on `text`. Thus, Bob introduces the third rule, which adds a hint to Q_2 to suggest PostgreSQL to use the B+ tree on `created_at`. The new rewritten query Q_2''' is shown in Figure 9.

The execution time of query Q_2''' is 2.5 seconds, more than two times faster than the previous rewritten query Q_2''. These examples show that by rewriting queries formulated by the Tableau live mode, VisBooster can significantly reduce the response time of visualization queries and improve the user experience.

4.2. Case 2: Accelerating Queries on MySQL

In case 2, we use a similar example as case 1 to show how the proposed framework accelerates visualization queries formulated by Tableau in its live mode connecting to a MySQL database. We first show that a rule similar to Rule-1 for PostgreSQL also applies to MySQL, i.e., removing type-casting expressions in the filtering clauses can help the database generate an index-based plan. However, for MySQL, the syntax of type-casting expressions formulated by Tableau differs from that for PostgreSQL. In the demonstration, we will use a scatterplot query to show how the database expert Bob introduces a rule to remove type-casting. Different from PostgreSQL, MySQL does not support trigram indexes. Thus for such queries, a rule similar to Rule-2 does not help for MySQL. This observation also shows the value of human involvement in the proposed framework because human knowledge about a specific database should be considered in the life cycle of query rewriting.

5. Open Problems

There are open research challenges related to using rewriting rules in VisBooster to solve the scalable in-situ data visualization problem in this middleware architecture. For example, how to help the database experts identify the rewriting rules? How and when to apply them (e.g., different query hints should be applied to different queries)? In what order should the rules be applied? We plan to address these challenges in our future work.

References

[1] C. S. et al., Polaris: A system for query, analysis, and visualization of multidimensional relational databases, IEEE Trans. Vis. and Comput. Graph. 8 (2002) 52–65.

[2] slintel, Tableau software market share, 2021. URL: https://www.slintel.com/tech/data-visualization/tableau-software-market-share, last accessed 2022-01-25.

[3] Tableau, Tableau online tips: Extracts, live connections, and cloud data, 2021. URL: https://www.tableau.com/about/blog/2016/4/tableau-online-tips-extracts-live-connections-cloud-data-53351, last accessed 2022-01-25.

[4] W. et al., An analytic data engine for visualization in tableau, in: SIGMOD 2011, Athens, Greece, June 12-16, 2011, ACM, 2011, pp. 1185–1194.

[5] G. Lohman, Is query optimization a "solved" problem?, ACM SIGMOD Blog. ACM Blog (2014).

[6] Z. Liu, J. Heer, The effects of interactive latency on exploratory visual analysis, IEEE Trans. Vis. Comput. Graph. 20 (2014) 2122–2131.

[7] C. A. Pahins, S. A. Stephens, et al., Hashedcubes: Simple, low memory, real-time visual exploration of big data, IEEE Trans. Vis. Comput. Graph. 23 (2017) 671–680.

[8] A. C. et al., The case for interactive data exploration accelerators (ideas), in: HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016, ACM, 2016, p. 11.

[9] S. A. et al, Automated selection of materialized views and indexes in SQL databases, in: VLDB 2000, September 10-14, 2000, Cairo, Egypt, Morgan Kaufmann, 2000, pp. 496–505.

[10] J. Goldstein, P. Larson, Optimizing queries using materialized views: A practical, scalable solution, in: SIGMOD 2001, Santa Barbara, CA, USA, May 21-24, 2001, 2001, pp. 331–342.

[11] L. B. et al, Dynamic prefetching of data tiles for interactive visualization, in: SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016, ACM, 2016, pp. 1363–1375.

[12] L. Battle, Behavior-driven optimization techniques for scalable data exploration, Ph.D. thesis, MIT, Cambridge, USA, 2017.

[13] R. M. et al., Bao: Making learned query optimization practical, in: SIGMOD '21, China, June 20-25, 2021, ACM, 2021, pp. 1275–1288.

[14] Oracle, Postgresql 14: Chapter 41. the rule system, 2021. URL: https://www.postgresql.org/docs/current/rules.html, last accessed 2022-01-25.

[15] Oracle, Mysql 8.0: Using the rewriter query rewrite plugin, 2021. URL: https://dev.mysql.com/doc/refman/8.0/en/rewriter-query-rewrite-plugin-usage.html, last accessed 2022-01-25.

[16] A. V. et al., Get real: How benchmarks fail to represent the real world, in: DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018, ACM, 2018, pp. 1:1–1:6.