HADA: Empower Database Experts with Data Dependencies

Daniel Lindner*
Hasso Plattner Institute
Potsdam, Germany
daniel.lindner@hpi.de

Jonas Baltruschat
Hasso Plattner Institute
Potsdam, Germany
jonas.baltruschat@hpi.de

Kisung Park
SAP Labs
Seoul, Korea
kisung.park@sap.com

Taehyung Lee
SAP Labs
Seoul, Korea
taehyung.lee@sap.com

Heesik Shin
SAP Labs
Seoul, Korea
heesik.shin@sap.com

Daniel Ritter
SAP
Walldorf, Germany
daniel.ritter@sap.com

Felix Naumann
Hasso Plattner Institute
Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Workload-driven data dependency discovery enables the use of previously unknown dependencies for query optimization. While greatly improving the performance of benchmark workloads, unsupervised systems cannot judge the genuineness of the discovered dependencies. Thus, it is not clear whether the dependencies can be expected to hold for any database instance, e. g., after data inserts or updates. We combine the dependency discovery approach of such systems with a user feedback loop to let domain experts and database administrators (i) decide which dependencies the database system can safely use for optimization purposes and (ii) assess the impact on the query plans. Our demonstration gives database experts and application developers full control over the use of dependencies and helps them improve the performance of their workloads running on SAP HANA Cloud. Furthermore, we present a new dependency-based optimization technique for this system.

PVLDB Reference Format:
Daniel Lindner, Jonas Baltruschat, Kisung Park, Taehyung Lee, Heesik Shin, Daniel Ritter, and Felix Naumann. HADA: Empower Database Experts with Data Dependencies. PVLDB, 19(12): 4502 - 4505, 2026.
doi:10.14778/3827998.3828050

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HPI-Information-Systems/HADA.

1 INTRODUCTION

Query optimization techniques that make use of data dependencies have been proposed over decades, as surveyed by Kossmann et al. [4]. While such techniques can improve database performance, applying them in real-world systems is challenging for multiple reasons [3, 5]. First, not all relevant dependencies are known or enforced by database management systems (DBMSs). Second, most of the dependency types, such as functional dependencies (FDs) or order dependencies (ODs), cannot even be passed to the DBMS's optimizer because SQL does not support to specify them [5]. Third, dependencies are sensitive to data modification: a dependency that is valid for a specific snapshot might be invalidated by any insertion, update, or deletion if it is not actively enforced.

Unsupervised dependency discovery and rewrite systems aim to address the first two challenges [3, 5, 6]. However, these systems cannot distinguish between real-world (*genuine*) and randomly occurring (*spurious*) dependencies, where the latter ones can be affected by changes in the data or the underlying data-generating application logic. We demonstrate a system that circumvents this issue by including database expert users (e. g., database administrators or application developers) in the loop for an interactive, supervised discovery. These users judge the dependencies' genuineness and select which dependencies can be passed to the optimizer as query hints. Furthermore, our system reveals to database admins which dependencies improve query plans, and how. While our approach could be extended to different DBMSs, we provide a solution for SAP HANA Cloud, showcase a new optimization technique, and present HADA,[1] the **HA**NA **D**ata dependency **A**ssistant. Thus, HADA can actively help in tuning a production database instance.

Interactively identifying *all* valid data dependencies in large tables with hundreds of columns and millions of rows is still infeasible even for state-of-the-art discovery algorithms [2]. Thus, HADA validates only *relevant* dependency candidates based on the workload, similar to previous work [3, 5]. Both the valid dependencies and the rewritten SQL queries enriched with the required hints are prompted to the users, who finally decide whether they want to make use of the exploited dependencies. In different panels, HADA visually presents differences in the query plans of the original query and the version with dependency knowledge, as well as a live performance comparison.

*(Part of) this work was conducted during an internship at SAP Labs Korea.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828050

[1] In Korean, "hada" (하다) means *make* or *do*.

HADA interacts with the database instance entirely via SQL: even dependency validation is performed on the database side using standard relational operators. This design allows for providing the system as a lightweight advisory service without setup overhead or data transfer. We use a novel dependency-based optimization technique in the SAP HANA Cloud in-memory DBMS [7] to showcase HADA, which we describe in Section 2. Section 3 presents the system's architecture and implementation. Finally, Section 4 sketches the user interaction with HADA using an application scenario to assess how query performance can improve.

2 EXAMPLE OPTIMIZATION: REPLACING SEMI-JOINS WITH TABLE SCANS

Many dependency-based query optimizations have been proposed before [4]. This section focuses on one of the of the rewrites supported by HADA: replacing semi-joins with table scans. First, we describe the existing implementation in SAP HANA Cloud. Second, we present data dependency-based extensions to this mechanism.

2.1 SAP HANA Cloud's Table Scan Semi-join

SAP HANA Cloud implements a specialized operator mainly used for equi semi-joins and reductions [8]. Conceptually, this *table scan semi-join* (TSSJ) operator performs an `IN` predicate on the reduced relation with all unique values of the reducer's join attribute. The operator also supports multiple conjunctive predicates, but executes them individually and intersects the result sets. Thus, tuples with value combinations that do not exist in the reducing relation can still qualify. When used to perform semi-joins that are not reductions (*genuine* semi-joins), this behavior limits the TSSJ's application to single-predicate equi semi-joins. However, the predicates of multiple TSSJs as well as further selections on the reduced relation can be condensed into a single TSSJ operator and fused into the general access-path selection. TSSJs are enumerated during cost-based optimization.

Executing this operator in SAP HANA Cloud's HEX engine [7] is efficient because the data is dictionary-encoded by default. Instead of an attribute's value, a column persists only the offset of the value in a sorted dictionary (*value ID*). The TSSJ operator qualifies rows from the reduced relation by evaluating value ID matches between the reduced and reducer relations. For each row, the TSSJ probes if the column's value ID is contained in the set of qualifying value IDs. These qualifying value IDs are created by looking up the reducer's side unique attribute values in the reduced relation's column dictionary or by translating them using lookup tables if the reducer side's column is also dictionary-encoded. Alternatively, the evaluation can exploit an existing index if index access is estimated as cheaper than a table scan.

The execution engine decides on the scan implementation during runtime based on the value ID set's characteristics: if there are only a few value IDs, it chooses a list and checks for list containment for each reduced relation's row. When there are more qualifying value IDs, the value ID set is represented as a bit vector that indicates if a value ID matches for each dictionary offset, enabling a direct lookup for each row. The engine also detects small, continuous ranges of value IDs and filters for tuples between ranges' lower and upper bound value IDs.

2.2 Data Dependency-based Extension

Data dependencies enable the application of two optimizations for the TSSJ operator: (i) a guarantee of arbitrarily sized, continuous join key ranges similar to [10] (join-to-predicate rewrite in [5]) and (ii) a generalization to multi-predicate equi-joins.

The first optimization targets joins with a single equality predicate. Assume there is an equality or range predicate on one (PK) input of the join and an order dependency between the join key and the predicate attribute. The order dependency guarantees that all join keys between the selection's minimal and maximal values qualify. A combination of a unique join key and an inclusion dependency guarantee that all tuples of the other (FK) side within that range find exactly one join partner. Thus, the original join is equivalent to a selection of join keys between the minimal and maximal value [5]. This rewrite also applies to conjunctions of selection predicates on the PK side as long (i) there exists an OD with the join key on the left-hand side and the selection attributes on the right-hand side and (ii) only the last attribute in the OD's left-hand side is part of a range predicate. Such queries are frequent for filters on dimension tables representing time units that are joined with fact tables. For instance, the *date_dim* table of the TPC-DS benchmark [11] contains one tuple for each date with a sequential key. Among others, each tuple has a field for the date itself, the year, the month, and the day of month. Every equality or range predicate on (i) the year, (ii) a range of months for a single year, (iii) days within a single month, or (iv) the date fulfills the properties above.

The second optimization targets multi-predicate equi-joins. Usually, genuine (semi-)joins cannot be replaced by TSSJ's with individual equality predicates that are sequentially applied to the attributes. Assume all join columns are part of an inclusion dependency. Then, all *value combinations* that are present in the data have a match in the referenced side. If one or more selection predicates on the referenced side depend on one of the join keys, a selection on them always selects or removes all tuples with the same value for that individual join key. A unique column combination on the join keys further guarantees that tuples qualify only once. Thus, the multi-predicate join can be transformed to a TSSJ that filters each join column individually.

3 HADA: A DATA DEPENDENCY-BACKED SQL PREPROCESSOR

This section overviews HADA's architecture, provides details on the dependency validation, and describes how dependencies are passed to the DBMS.

3.1 Overview

The approach of HADA is similar to previous work [3, 5]: the system parses query plans of SQL queries optimized without dependency knowledge, generates candidates for dependencies that can be used during optimization, validates these candidates, and applies valid candidates in the optimization process. The main differences compared to prior work are (i) that a human is in the loop to select genuine dependencies and (ii) that the discovery is decoupled from the DBMS and communicates entirely via the SQL interface. Thus, it can connect to any SAP HANA Cloud instance that supports the dependency-based optimizations and dependency passing.

Figure 1: Original SQL query and rewritten version with SQL hint that uses two selected dependencies (highlighted).

The starting point of the interactive dependency discovery is a user-provided query (see Figure 1). Analogously to previous work [5], HADA performs a rule-based analysis of the query plan, which is obtained via an `EXPLAIN` statement. The result of the analysis is a set of data dependency candidates that benefit optimization if they were valid. These candidates cover multiple techniques implemented in SAP HANA Cloud, such as dependent group-by reduction, join to semi-join rewrites, and the technique presented in Section 2.2. HADA issues SQL queries to the database instance to validate if these candidates hold on the data.

Users can select both valid and invalid dependencies to be passed to SAP HANA Cloud (Figure 2). Thus, they can simulate the impact of the dependencies if they were valid: violations of a meaningful dependency can point out a data quality issue, and the impact simulation shows the potential benefit of data cleaning. Having expert users in the loop also helps to judge the genuineness of the valid dependencies: dependencies can hold true by chance on a given database state and might be invalidated by changes to the data. However, users can decide whether the dependencies are meaningful and select only the ones that will remain valid in the future. To assist in the selection process, HADA scores the dependencies' genuineness with metrics similar to related work [9].

The selected queries are passed to the DBMS via SQL hints, so the optimizer can decide to choose a different, more efficient query plan. To assess the plan changes, HADA parses the different plan versions obtained with `EXPLAIN` statements and shows the plan graphs to the users. Furthermore, the users can start a live benchmark to compare the runtime of the to query versions. Together with the selected dependencies and the SQL queries, these performance numbers are shown in a history. Thus, users can test and execute the query with different combinations of dependencies to understand the impact of each individual dependency. Furthermore, HADA verifies that the query results are correct and logs the interaction with the database.

3.2 SQL-based Dependency Validation

Previous research focused on efficient and automatic dependency validation of many candidates inside the database [5]. With a human-in-the-loop setting and a single query, validation performance must still be responsive, but is less important. HADA aims for a decoupled solution: it can connect to any database instance that implements the supported optimization techniques without adapting the DBMS core to support internal validation. Thus, HADA also performs the dependency validation step via the SQL interface.

For an FD `zip` $\longrightarrow$ `city`, only one distinct `city` value and no further `NULL` values must occur per `zip`. We group by the determinant column `zip` and count the distinct dependent values `city` as well as

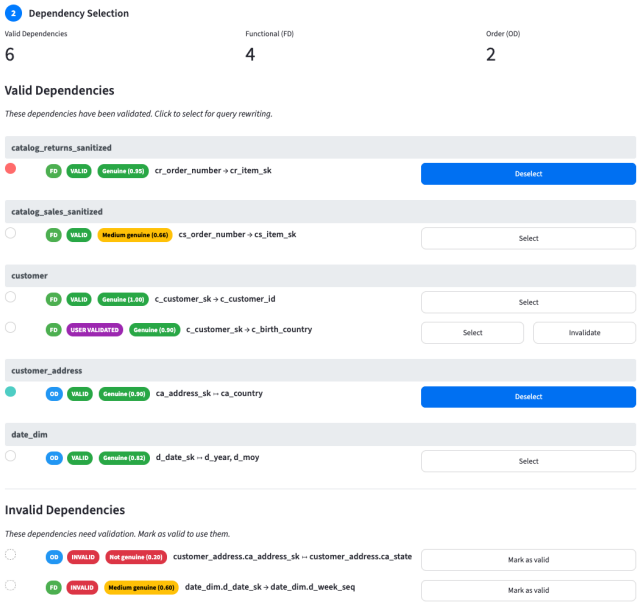

Figure 2: Overview of the dependency discovery. Users can select valid dependencies for optimization and even choose invalid, but beneficial candidates to simulate their potential.

the number of `NULL` values. Then, we select the maximum of these aggregated values. When there is at most one distinct dependent value per group and no `NULL`s, the FD is valid.

OD validation is more complicated because we must compare tuples with their predecessor respective to the ordering. Assume we want to validate the OD `date_key` $\mapsto$ [`d_year`, `d_month`]. First, we obtain the `d_year` and `d_month` values of the previous tuple with respect to ordering by `date_key` with a `leap` window function. Then, we perform a lexicographical comparison of the right-hand side attributes and the previous values using a `CASE` expression. If an aggregation of all comparison results is true, the OD is valid.

Unique column combinations (UCCs) can be validated with a simple distinct count query, and SQL-based validation of inclusion dependencies (INDs) has been proposed before [1, p. 57–60]. Furthermore, these two dependency types can be defined via unique/primary key or foreign key constraints that allow to skip the validation process at all. HADA generates and executes all required validation queries and groups multiple candidates with the same left-hand side in a single query to reduce the overhead. The relational operators used for the validation scale linearly or linearithmicly with the input, and the overall discovery never exceeded a few seconds on TPC-DS data with a scale factor of 10.

3.3 Passing Data Dependencies

HADA passes only the user-selected data dependencies to the optimizer as query hints. Thus, we make discovered dependencies transparent to the users, including dependencies that cannot (yet) be guaranteed by constraints. Additionally, explicitly passing individual dependencies allows the users to assess the impact of each dependency or combination of dependencies in isolation. SAP

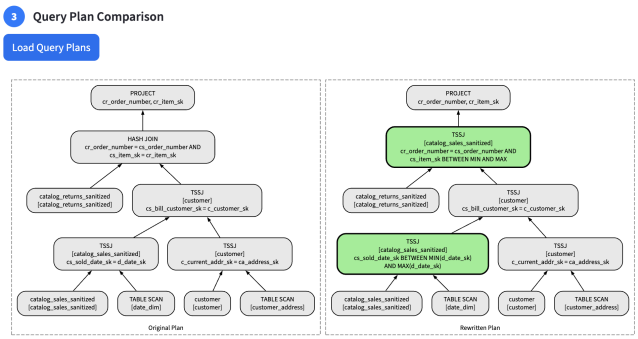

Figure 3: Visual comparison of the original query plan and the version using dependency knowledge. Differences are highlighted.

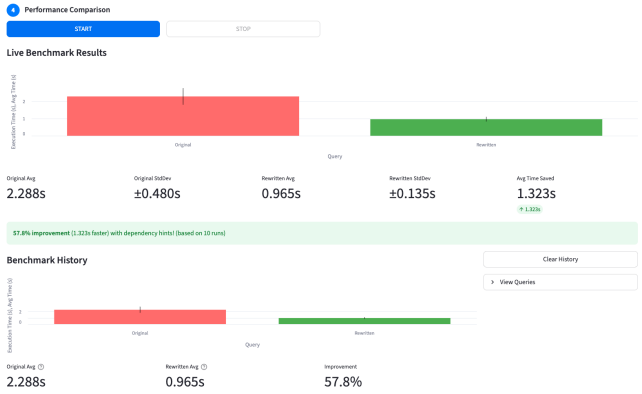

Figure 4: Live performance comparison of original query and version with dependency knowledge, including history.

HANA Cloud parses the given dependencies and passes them to its optimizer via metadata structures. By specifying the dependencies for individual queries, users can even exploit them if they are not valid for the entire relation, but only for tuples qualified by further selections (conditional dependencies). Furthermore, hints are tied to individual SQL queries, and there are no side effects for the remaining workload. In the future, data dependencies could be defined similarly to SQL constraints or other metadata objects, such as histograms or sketches. Specialized indexes could monitor or enforce the validity of FDs and ODs.

4 DEMONSTRATION SCENARIO

Users establish a live connection to a database instance in HADA's sidebar. The sidebar also includes optional plan cache loading to start from a cached statement. Furthermore, the sidebar allows users to choose which optimization techniques they want to test.

Users enter a SQL statement (1 *SQL Query Input* in Figure 1) and initiate the dependency discovery. The 2 *Dependency Selection* view (Figure 2) shows the resulting dependencies in two categories: valid dependencies are grouped by table. Invalid candidates are listed separately and can be marked as valid by the users to assess the system's capabilities with cleaned data. The respective tags for each dependency indicate the dependency type and the state of validity. Additionally, each dependency holds a tag that indicates its genuineness. To assist the users' judgment, a genuineness score calculated on the underlying data highlights whether a dependency is classified as *not genuine* (score < 0.4), *medium genuine* (score $\in [0.4, 0.7]$), or *genuine* (score > 0.7). Chosen dependencies are color-coded consistently across the interface. Next to the dependency list, the *Selection Summary* panel displays the currently selected dependencies and shows the hint that will be inserted into the query. After applying the rewrite, the query enriched with the SQL hint appears and is color-linked to the selected dependencies.

The 3 *Query Plan Comparison* view, shown in Figure 3, provides a side-by-side graph-rendering of the original and rewritten execution plans. Nodes that are changed in the rewritten plan are highlighted in green with bold borders to make structural changes identifiable at the operator level. Figure 4 shows the 4 *Performance Comparison* section, where users can execute a continuous benchmark that alternates between the original and rewritten SQL statements. The view reports average time and standard deviation for each query in a live bar chart and shows the average time saved. Each benchmark session is stored in a history to facilitate comparisons across different selection choices.

The 5 *Query Results Preview* displays the first 100 rows of both query results and a banner indicating whether the result sets match. Identical results are expected when users select valid dependencies: the hint affects only performance, not semantics. Finally, 6 *SQL Logs* expose the 20 most recent SQL operations. HADA discovers dependencies for multiple optimization techniques. Users can also modify the data and experience how that affects the dependencies.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling.* Morgan & Claypool Publishers.

[2] Tobias Bleifuß, Thorsten Papenbrock, Thomas Bläsius, Martin Schirneck, and Felix Naumann. 2024. Discovering Functional Dependencies through Hitting Set Enumeration. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 43:1–43:24.

[3] Jan Kossmann, Daniel Lindner, Felix Naumann, and Thorsten Papenbrock. 2022. Workload-driven, Lazy Discovery of Data Dependencies for Query Optimization. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 7 pages.

[4] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *The VLDB Journal* 31, 1 (2022), 1–22.

[5] Daniel Lindner, Daniel Ritter, and Felix Naumann. 2026. Unleashing Data Dependency-based Query Optimization. In *Proceedings of the International Conference on Extending Database Technology (EDBT).* 516–529.

[6] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2023. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. *PVLDB* 16, 6 (2023), 1208–1221.

[7] Norman May, Alexander Böhm, Daniel Ritter, Frank Renkes, Mihnea Andrei, and Wolfgang Lehner. 2025. SAP HANA Cloud: Data Management for Modern Enterprise Applications. In *Companion of the International Conference on Management of Data (SIGMOD).* 580–592.

[8] Manuel Mayr, Wolfgang Stephan, and Till Merker. 2022. Table scan predicate with integrated semi-join filter. U.S. Patent 12099490B2.

[9] Thorsten Papenbrock and Felix Naumann. 2017. Data-driven Schema Normalization. In *Proceedings of the International Conference on Extending Database Technology (EDBT).* 342–353.

[10] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Przemyslaw Pawluk, and Calisto Zuzarte. 2011. Queries on dates: fast yet not blind. In *Proceedings of the International Conference on Extending Database Technology (EDBT).* 497–502.

[11] Transaction Processing Performance Council. 2024. *TPC Benchmark DS.* Standard Specification Version 4.0.0.