Analyzing Query Execution for Integrity Constraint Violation Detection

Alessandro Neves dos Santos[1], **Eduardo H. M. Pena**[1]

Federal University of Technology - Paraná (UTFPR)[1]
Campo Mourão, PR – Brazil

alesan.2000@alunos.utfpr.edu.br, eduardopena@utfpr.edu.br

Abstract. *Data consistency ensures the validity and integrity of data representing real-world entities. Denial constraints (DCs) generalize various integrity constraints, providing a powerful way to define rules that ensure data consistency. This work analyzes the capabilities of relational database management systems (RDBMSs) to detect DC violations in different metrics. We explore various SQL patterns for measuring DC violations and evaluate the performance of multiple RDBMSs with extensive experiments, highlighting potential performance improvements, choke points, and limitations when using them.*

1. Introduction

Data consistency refers to the validity and integrity of the data representing real-world entities. In this context, denial constraints (DCs) provide a powerful way to define rules that ensure data consistency since they are a formalism that generalizes several types of data dependencies, including functional and order dependencies. Typically, measuring the inconsistency of a database regarding a set of DCs is achieved by detecting DC violations, which are sets of tuples that do not comply with the DCs.

Several studies have used constraint-based techniques for data error detection and data repairing [Chu et al. 2016]. [Fan et al. 2008] study conditional functional dependencies to capture inconsistencies using SQL. HoloClean [Rekatsinas et al. 2017] and NADEEF [Dallachiesa et al. 2013] are well-known data cleaning systems that use PostgreSQL to detect and repair data errors, leveraging the DC formalism to define integrity rules. [Pena et al. 2021] proposes a standalone tool to detect violations of DCs, showing that some RDBMSs fall short in performance depending on the database size and DC predicates. However, given RDBMS's availability and ease of use, data professionals often opt for them instead of standalone tools. Also, since the proposal of the tool, some RDBMS have updated the available algorithms, which are now more capable of delivering faster results [1]. Thus, this paper revisits relational DBMSs for handling constraint-based queries. Our study includes exploring various SQL query patterns that facilitate the measurement of different facets of DC-related inconsistencies. Additionally, we analyze the performance of several RDBMSs, examining various query processing engines.

Our contributions are as follows: we review different DC queries for violation detection studied in previous works, along with new derivations using alternative clauses (e.g., `GROUP BY` or `EXISTS`), evaluating the potential performance improvements or

[1]https://duckdb.org/2022/05/27/iejoin.html

drawbacks associated with their use; we conduct extensive experiments on diverse combinations of datasets, RDBMSs, and DCs, measuring the execution time of over 9000 query executions and analyzing their general behavior, highlighting possible choke points and limitations, including bugs, encountered when using the different RDBMSs.

2. Background

Consider a relation instance r with schema R, and predicates $p : t.A \ \theta \ t'.B$, where $A, B \in R$; θ is a comparison operator in $\{=, \neq, <, \leq, >, \geq\}$; and t, t' is a pair of distinct tuples of r. A DC φ specifies a predicate conjunction that must not be true given any tuple pair of r. The DC φ is expressed as: $\varphi : \forall t, t' \in r, \neg(p_1 \wedge \ldots \wedge p_m)$, and it is satisfied if, for all pairs of the table, at least one of the predicates of φ is false. Intuitively, tuples violating DCs point to inconsistent and, thus, problematic value combinations in the database.

Computing DC violations can be achieved using intuitive SQL queries, but different RDBMSs may process these queries in distinct ways. Factors such as the query processing model, selectivity of DC predicates, and the join algorithms used for DC predicates can significantly impact processing costs.

Query Processing Models. Query processing models define how RDBMSs handle and execute queries. The iterator model (aka, Volcano model) processes queries tuple by tuple using iterators and operators that extract data from its child operators in a pipeline. It is memory efficient, but potentially incurs the function call overhead, as operations are called for each tuple. The materialization model processes intermediate results by fully computing and storing them before passing to the next operator, reducing re-computation, but possibly requiring substantial memory usage. The vectorization model improves CPU cache utilization and reduces function call overhead by processing batches or vectors of tuples at a time, balancing the granularity of tuple-at-a-time processing and the efficiency of batch operations. The compilation model translates queries into low-level machine code before execution, allowing specific query handling that eliminates interpretation overhead and can leverage advanced CPU features [Kersten et al. 2018].

Predicate Selectivity. The predicate selectivity, i.e., how many pairs of tuples satisfy a given predicate, can significantly affect query processing cost. High-selectivity predicates, such as equalities, usually return a small subset of tuples, reducing the data for further stages in a pipeline. In turn, low-selectivity predicates, such as inequality comparisons ($\neq, <, \leq, >, \geq$), can return a large portion of the table, increasing the computational cost as more data needs to be scanned, filtered, and possibly joined.

Join Algorithms. Execution planners must choose the most suitable join algorithms for efficient predicate evaluation. Although algorithms such as Nested-Loop Join, Sort-Merge Join, and Hash Join can be used in typical cases, each provides specific advantages and disadvantages depending on the dataset, size, and predicates evaluated. For specific cases, there are algorithms that can offer more efficiency. For example, IEJoin, a recently proposed join algorithm, is already being used by some RDBMSs (e.g., DuckDB). It can process a pair of range predicates more efficiently than traditional join algorithms.

3. Proposed Evaluation

We exploit the rich predicate structure of DCs to explore various SQL query patterns. By applying these SQL queries to the DCs, we compute different metrics for data incon-

sistency. Each metric provides a different perspective on the consistency of the dataset. Table 1 presents the SQL queries and respective metrics obtained by executing them. The metrics have been studied in previous works dealing with discovering DC violations [Pena et al. 2021], data cleaning and repairing [Fan et al. 2008, Rekatsinas et al. 2017] or inconsistency measures [Livshits et al. 2021].

Metric m_1 provides the set of inconsistent pairs of tuples, which point to tuples that conflict with each other regarding a DC, i.e., the DC violations. Queries Q_1 and Q_3 compute m_1. The m_2 metric provides the number of violations existing in the dataset, where the inconsistency is measured in a single number with query Q_2. The m_3 metric provides the set of tuples that participate in any violation, that is, the set of tuples existing in any of the pairs obtained for the m_1 metric. Queries Q_4 and Q_5 capture m_3 . The m_4 metric informs whether any violation exists in the dataset and is obtained when query Q_6 returns a non-empty result. The metrics m_1 and m_3 are useful for DC-based data cleaning methods. For instance, due to the anti-monotonic property of DCs (i.e., inconsistency cannot increase by deleting tuples), a simple way to remove violations is to delete the tuples obtained for m_3 metric [Livshits et al. 2021]. We evaluated a pair of queries to obtain metrics m_1 and m_3, enabling queries with (potentially) different execution plans that might lead to faster query execution. We include the condition t.ID $\neq$ t'.ID to ensure that only distinct tuples are used to compare the DC predicates.

Query	SQL Statement	Metric obtained
Q_1	SELECT t.ID, t'.ID FROM r t JOIN r t' ON p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID	m_1
Q_2	SELECT COUNT(*) FROM r t JOIN r t' ON p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID	m_2
Q_3	SELECT t.ID, t'.ID FROM r t, r t' GROUP BY $t.A_1$, ..., $t.A_m$, $t'.B_1$, ..., $t'.B_m$, t.ID, t'.ID HAVING p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID	m_1
Q_4	SELECT t.ID FROM r t JOIN r t' ON p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID UNION SELECT t'.ID FROM r t JOIN r t' ON p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID	m_3
Q_5	SELECT t.ID FROM r t WHERE EXISTS (SELECT 1 FROM r t' WHERE p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID) UNION SELECT t'.ID FROM r t' WHERE EXISTS (SELECT 1 FROM r t WHERE p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID)	m_3
Q_6	SELECT t.ID, t'.ID FROM r t JOIN r t' ON p_1 AND ... AND p_m AND t.ID $\neq$ t'.ID LIMIT 1	m_4

Table 1. SQL queries used to detect data inconsistencies regarding DCs.

RDBMS Systems. We used RDBMSs that use different techniques for query processing. PostgreSQL (v16.2), MySQL (v8.3.0), and SQLite (v3.37.2) use the iterator model. DuckDB (v0.8.1) takes advantage of the vectorization model and implements IEJoin. Umbra (2023-11-14) implements just-in-time query compilation. RDBMS-1 and RDBMS-2 are community/developer versions of commercial RDBMSs. RDBMS-1 uses a hybrid approach between an iterator and a vectorization model, and RDBMS-2 mainly uses the

iterator model. We do not evaluate RDBMSs that use the materialization model (e.g., MonetDB) since previous work has shown critical issues related to memory usage for queries aimed at discovering DC violations, particularly for large datasets and range and non-equality predicates [Pena et al. 2021].

4. Experimental Evaluation

Our experiments include datasets from open-access sources (i.e., Kaggle and NYC OpenData) and repositories of previous work. For the DCs, we use the DC mining algorithm from [Pena et al. 2022]. We inspected the discovered results and select a few DCs for our tests. Table 2 presents the datasets and DCs selected. We start with clean datasets and then generate datasets with different noise levels (DC violations). We use the noise generator algorithm (CONoise) described in [Livshits et al. 2021], which inserts random DC violations by adjusting tuple values to disrupt the dataset consistency, repeating the process until reaching the desired amounts of noise. We analyze the execution plans of the queries and measure the execution time (arithmetic mean of three executions, fetching all resulting tuples) for each combination of dataset, DC, RDBMS, and noise level. Queries that exceed four hours were interrupted and disregarded, as well as those that result in errors or are not supported by the RDBMS being evaluated. The datasets produced, DCs, results, and scripts used are publicly available [2].

Table 2. Datasets and DCs.

	Name	Size	DC
φ_1	Brasil Exp	10 000 000	$\neg(\texttt{t.ncm} = \texttt{t'.ncm} \land \texttt{t.unid} < \texttt{t'.unid} \land \texttt{t.via} > \texttt{t'.via})$
φ_2	Dob Job	2 182 000	$\neg(\texttt{t.job} = \texttt{t'.job} \land \texttt{t.borough} \neq \texttt{t'.borough} \land \texttt{t.lot} > \texttt{t'.lot})$
φ_3	Measures	1 318 000	$\neg(\texttt{t.coolant} > \texttt{t'.coolant} \land \texttt{t.stator} < \texttt{t'.stator} \land \texttt{t.pm} = \texttt{t'.pm})$
φ_4	Tax	999 900	$\neg(\texttt{t.city} \neq \texttt{t'.city} \land \texttt{t.zip} = \texttt{t'.zip})$
φ_5	Loan Default	255 000	$\neg(\texttt{t.age} > \texttt{t'.age} \land \texttt{t.income} = \texttt{t'.income} \land \texttt{t.amount} = \texttt{t'.amount} \land \texttt{t.term} < \texttt{t'.term})$
φ_6	Lineitem	152 000	$\neg(\texttt{t.quantity} = \texttt{t'.quantity} \land \texttt{t.tax} = \texttt{t'.tax} \land \texttt{t.discount} < \texttt{t'.discount} \land \texttt{t.price} > \texttt{t'.price})$
φ_7	Salaries1	100 000	$\neg(\texttt{t.base_pay} > \texttt{t'.base_pay} \land \texttt{t.overtime_pay} > \texttt{t'.overtime_pay} \land \texttt{t.other_pay} > \texttt{t'.other_pay} \land \texttt{t.total} < \texttt{t'.total})$
φ_8	Salaries2	100 000	$\neg(\texttt{t.total_pay} > \texttt{t'.total_pay_benefits} \land \texttt{t.total_pay_benefits} > \texttt{t'.total_pay})$
φ_9	Yellow Taxi	100 000	$\neg(\texttt{t.pickup_datetime} > \texttt{t'.dropoff_datetime} \land \texttt{t.dropoff_datetime} < \texttt{t'.pickup_datetime})$
φ_{10}	Flights	91 000	$\neg(\texttt{t.origin} = \texttt{t'.origin} \land \texttt{t.destination} = \texttt{t'.destination} \land \texttt{t.flights} > \texttt{t'.flights} \land \texttt{t.passengers} < \texttt{t'.passengers})$

We used a server with the following configuration for running the queries: Intel Xeon E5-2620 2.10Ghz Processor (4 cores), 20GB of RAM, 100GB of SDD (CT2000MX500SSD1), OS Ubuntu 22.04.3 LTS (5.15.0-101-generic kernel).

Increasing number of violations. Figure 1 shows the results for the RDBMSs running the different queries for datasets with increasing ratios of violations (measured by m_3 metric). In general, DuckDB and Umbra perform better than the other RDBMSs, which suggests that some design choices, such as advanced parallel processing and just-in-time query compilation employed by Umbra, as well as vectorized execution used by DuckDB, help the underlying RDBMSs outperform the others [Raasveldt and Mühleisen 2019, Neumann and Freitag 2020]. In cases where the IEJoin algorithm is leveraged as a join operator (i.e., φ_7, φ_8 and φ_9), DuckDB can approach or even surpass the performance of Umbra, providing significantly superior performance compared to other RDBMSs. For instance, on φ_9 for Q_1, DuckDB performs 28x faster than Umbra and more than 2300x faster than other RDBMSs; similar results occur in φ_8. Although Umbra generally achieves a lower execution time than DuckDB for the other datasets and DCs, this

[2] `https://github.com/Alessandro-Neves/SQL-DC-violations-detection`

difference tends to decrease or even reverse when the number of violations in the dataset increases, due to the time spent fetching the results produced for m_1 and m_3 metrics.

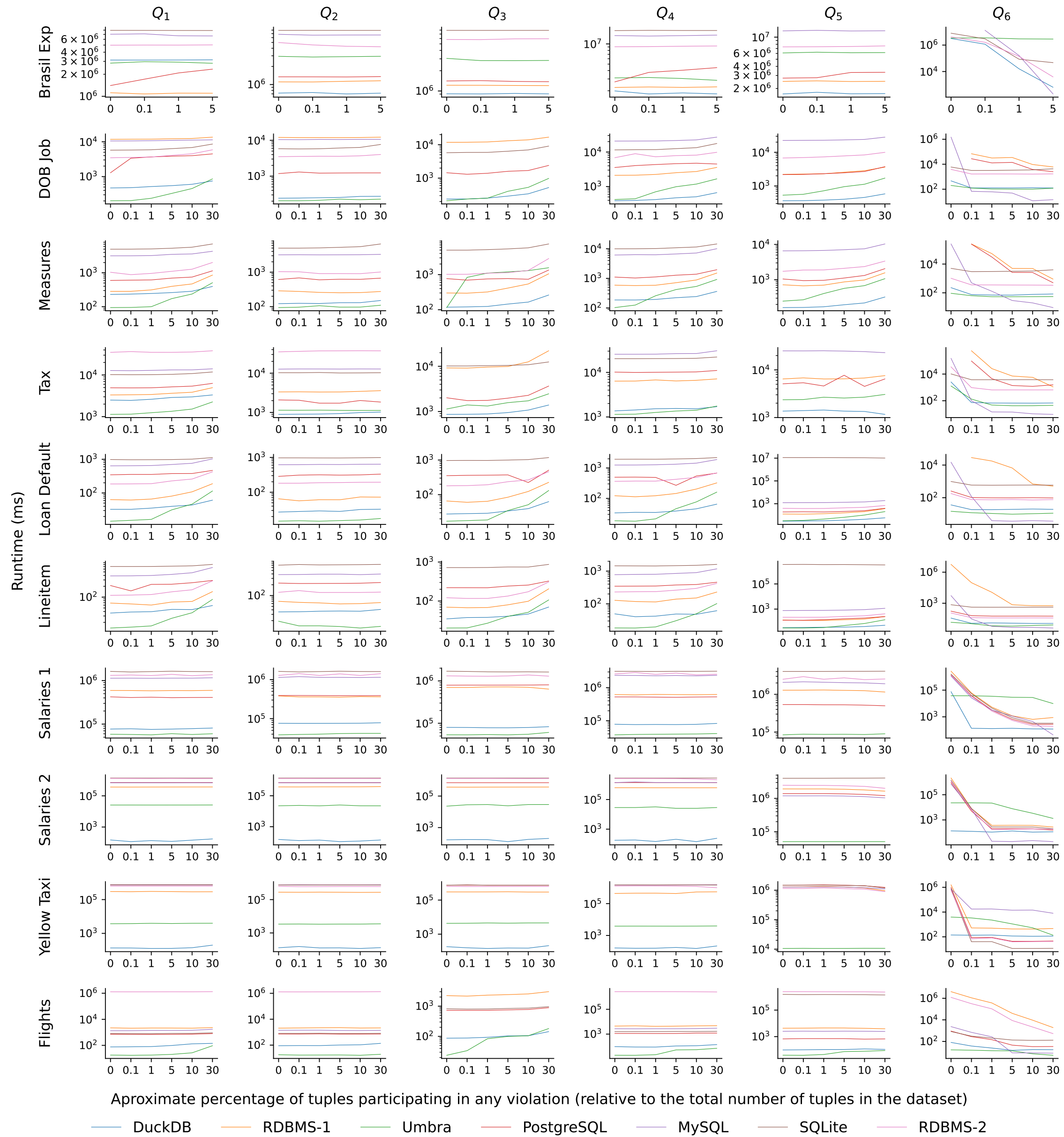

Figure 1. Runtime comparison for datasets with fixed size and increasing noise.

As seen for queries Q_6, when the goal is to verify the existence of violations, PostgreSQL, MySQL, RDBMS-1, and RDBMS-2 can significantly reduce their processing time on datasets with violations, especially when the dataset has a high noise level. MySQL stands out in these cases, even outperforming DuckDB. In this case, the iterator model execution may produce partial results early and finish due to the LIMIT clause, significantly reducing query processing time. After the first results are generated, processing the remaining tuples is unnecessary. However, it leads to poor performance when the dataset is clean since it must evaluate all tuples to confirm no violation exists, resulting in multiple iterations and re-evaluations of join conditions, adding substantial overhead in

the query processing, and significantly increasing processing time.

Increasing number of rows. For the next experiment, we focus on DC φ_1, which is more computationally demanding. We generated datasets with increasing sizes, and the amount of tuples that participate in any violation is fixed to nearly 1%. As seen in Figure 2, the results show the same behaviors observed in the first experiment, with the processing time growing significantly as the dataset size and number of violations increase (except in Q_6, due to the use of LIMIT), for instance, for the results in Q_4, the difference in execution time between the best and worst performing RDBMS is 122 seconds for the dataset with 1M tuples. For the largest dataset, with 10M tuples, this difference increases to more than 3 hours, DuckDB takes 1826 seconds to complete the same task that MySQL takes 13291 seconds. We run the experiment for DC φ_7 and observed similar trends.

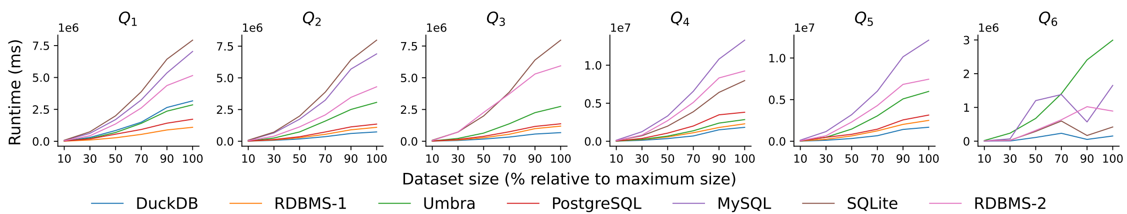

Figure 2. Runtime for φ_1 with progressive size and noise fixed at nearly 1%

Discussion. The CPU usage during query processing indicates that when executing Q_1 on φ_1, DuckDB predominantly uses only one processor core during query processing, in contrast to Q_3, DuckDB uses the four available processor cores. However, both return the same results and are executed on the same dataset; this behavior is observed even in reduced versions, as shown in Figure 2. DuckDB also performs faster in Q_3 compared to Q_1 for φ_2, φ_3, and φ_4. However, in these cases, the query execution time is too short to observe how the processor cores are used during the query execution. The SQLite presents an intense reduction in performance when processing Q_5 for all evaluated datasets and DCs, frequently exceeding the established execution time limit. A critical issue (i.e., bug) observed is that DuckDB produces incorrect results by returning all tuples in the dataset as participating in any violation (m_3) for query Q_5 (which uses the `EXISTS` clause) when the Nested-Loop Join is employed for semi-join operations (i.e., φ_7, φ_8, and φ_9), in this case, the execution time decreases to just a few tens of milliseconds. For all other query executions in DuckDB and other RDBMSs, the results presented the expected values.

5. Conclusion and future work

This study investigated the performance of executing SQL queries for DC violation detection across several RDBMSs. Our experiments show that some strategies, such as query compilation or vectorized execution, perform better in most cases. Also, they show potential bottlenecks in commonly used RDBMSs. This preliminary evaluation focuses on the vanilla/out-of-the-box versions of RDBMSs without any fine-tuning or indexes. As observed in previous studies, the indexing space for DCs can be extensive and not always result in better-performance plans. However, a deeper view of fine-tuning and indexing offers excellent opportunities for future work, as well as integrating the best-performing systems with data cleaning solutions that currently use the less-performant systems.

References

Chu, X., Ilyas, I. F., Krishnan, S., and Wang, J. (2016). Data cleaning: Overview and emerging challenges. In *SIGMOD*, page 2201–2206.

Dallachiesa, M., Ebaid, A., Eldawy, A., Elmagarmid, A., Ilyas, I. F., Ouzzani, M., and Tang, N. (2013). Nadeef: a commodity data cleaning system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 541–552, New York, NY, USA. Association for Computing Machinery.

Fan, W., Geerts, F., Jia, X., and Kementsietsidis, A. (2008). Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2).

Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A., and Boncz, P. (2018). Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222.

Livshits, E., Kochirgan, R., Tsur, S., Ilyas, I. F., Kimelfeld, B., and Roy, S. (2021). Properties of inconsistency measures for databases. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1182–1194, New York, NY, USA. Association for Computing Machinery.

Neumann, T. and Freitag, M. J. (2020). Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

Pena, E. H. M., de Almeida, E. C., and Naumann, F. (2021). Fast detection of denial constraint violations. *Proc. VLDB Endow.*, 15(4):859–871.

Pena, E. H. M., Porto, F., and Naumann, F. (2022). Fast algorithms for denial constraint discovery. *Proc. VLDB Endow.*, 16(4):684–696.

Raasveldt, M. and Mühleisen, H. (2019). Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA. Association for Computing Machinery.

Rekatsinas, T., Chu, X., Ilyas, I. F., and Ré, C. (2017). HoloClean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201.