Precision Tolerant Database Systems: Handling Imprecise Numerical Data

Hillary Hiu Ming Ho (Student Number: 7236670)
Mahmoud Saleh (Student Number: 7332038),
Ruiqi Zhang (Student Number: 7767547),
Tang Cheung NG (Student Number: 7543530),
Xuan Huyen Nguyen (Student Number: 7363382),
Yonax Lonard (Student Number: 7046170),

Supervisor: Dr. Janusz Getta
Faculty of Engineering and Information Sciences
School of Computing and Information Technology
University of Wollongong, Australia, 2023

***Abstract*—Relational databases ensure data consistency through the application of conditional and consistency constraints. When data insertion violates these constraints, the entire row is typically rejected to maintain the precision of the database. However, this approach results in the removal of not only uncertain data but also potentially useful information. This project explores an alternative to traditional database with constraints by attempting to preserve so-called imprecise data while ensuring robust database operations and query accuracy. The proposed precision-tolerant systems will be implemented as an extension of existing database management systems, addressing data storage, query handling, aggregation, and table merging tasks for data that violates Numerical Conditional Constraints (NCC). By prioritizing data retention, we anticipate several potential advantages, including cost savings in data cleaning and improved analytics capabilities, with the understanding of the associated trade-offs.**

I. Introduction

Codd's invention of relational databases in the 1970s resulted in a fundamental shift in data management [1]. This concept transformed information management by providing a structured and ordered method for storing and retrieving data. Relational databases easily manage complicated data sets by utilizing tables, rows, columns, and relationships. The relational table is still the most extensively used data model today, serving as the foundation for the vast majority of database management systems (DBMS) [2]. The market for relational database systems is large, reflecting their widespread use in business. When compared to previous data models, the relational model has several advantages, including a straightforward data representation and a streamlined technique for executing even the most complex queries.

After exploring the historical background and widespread usage of relational databases in today's DBMS, let us look at the common practice in preserving data integrity. Normally, database will implement constraints to retain it's relational and precision integrity in order to provide robust operation. Data rows will be checked for constraints violations prior to insertion, and will be rejected altogether if found. From the perspective of big data analytics, the presence of errors and noise becomes more prominent, given the vast amounts of data collected from various sources. Errors, including value omissions, misspellings, erratic formats, multiple entries for the same item, and violations of business and data integrity rules, frequently occur. The results of a recent survey on data science and machine learning (ML) confirms that handling incomplete data is among the primary obstacle faced by professionals working with data [3].

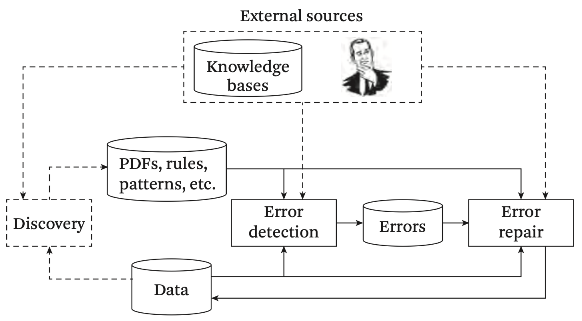
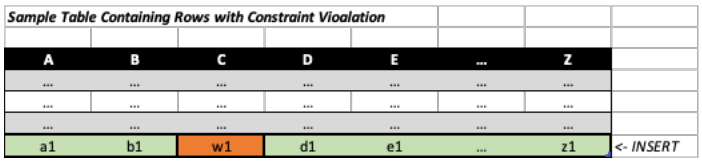


Fig. 1: Typical Data Cleaning Workflow with optional discovery step, error detection step and error repair step [3]

While data cleaning is crucial for businesses requiring high-accuracy representation data and obtain precise and useful results, its cost can take up the majority of database maintenance budget. Given the sheer volume of data being collected from various sources, it is inevitable that inconsistencies and rule violations will occur. In this case, manual data cleansing may be impractical due to its time-consuming nature and susceptibility to errors [4]. The main goal of the project is not the same with data cleaning objective, but rather to serve as a pre-processing and query handling solution that can operate reliably prior or without data cleaning.

II. Aims

The primary objective of the project is to design solution system that is robust to rule violations and able to retain as much information as possible. A single error within a multi-dimensional data record may not nullify the value of the entire row. As seen in fig. 2, a violation in cell colored red consequently reject all the potentially valuable information colored green. Our methods are not seeking to correct these errors or predict correct values, but giving the users the flexibility to decide what to do with them. The project aims to organise the precise information of the data along with the imprecise or uncertain components and allow users to specify the precision level during query. Our solution procedures should provide clear distinctions between the reliable and unreliable portions of the data, thereby enabling robust query processing and query aggregation. By ensuring that no information is lost during storage and processing, we can allow for subsequent data cleaning or forensic activities if required. We've identified existing gaps that can offer valuable opportunities for solutions, particularly in areas like big data analytics, where richer data is always more desirable, operational database management, where robustness is always a priority, and sensor data recording, which is known to be susceptible to noise and inaccuracies.



Sample Table Containing Rows with Constraint Vioalation

A	B	C	D	E	...	Z	
...	...	...	...	...	...	...	
...	...	...	...	...	...	...	
...	...	...	...	...	...	...	
a1	b1	w1	d1	e1	...	z1	<- INSERT

Fig. 2: Sample table showing a row with imprecise data or violation in one of the column, but contains much information which will be rejected altogether otherwise

There are numerous categories of violations that normally occur in a database. The most of the common ones are imposed by Integrity Constraints (IC) through the enforcement of data quality rules [5]. An example of an Integrity Constraint (IC) could involve two rows in a database sharing the same value for one characteristic (X) but different values for another (Y). A different example, domain constraints within an IC require certain attributes to fall within a specific range or category of values. For instance, a practical application of a domain constraint could be enforcing a minimum age of 18 for individuals to qualify for a driver's license. Establishing a thorough set of integrity constraints for each organization's policies is challenging and require domain experts to establish rule-based constraints. Furthermore, the application of the rule-based using IC language has limited expressiveness. For example, constraints are limited to single-row basis, unable to implement rules involving of multiple records. Since Integrity Constraints uniformly reject entire rows of data due to a single violation, we aim to explore the possibility of eliminating these constraints from the database and substituting them with an alternative method for storing the data in its entirety but retain the capability of working with data that meet the conventional IC.

In this project, we will specifically focus on data processing for storage and query handling data pertaining **Numerical Conditional Constraints (NCC)**. In general, NCC encompasses a broad scope that refers to restrictions imposed on numerical data within a database, it ensures that specific numerical calculations or values meet certain pre-established conditions. In the context of this project, the scope of NCC violations we cover are classified into three categories: range violations, dependency violations, and aggregation violations.

Range violations is normally categorized into domain constraints for numerical data. To illustrate, range violations occur when a numerical value falls outside an acceptable range, this range is defined by upper and lower bounds or specific threshold values, beyond which the numerical data is considered invalid or inappropriate. Ensuring that numerical values stay within their defined ranges is essential to prevent errors, inaccuracies, or unexpected outcomes in the system. In addition, dependency violations occur when a numerical value depends on a function of other numerical values, and the two values are not consistent or compatible. In such cases, the values should be interconnected logically, and any inconsistency may lead to data integrity issues or faulty computations. Lastly, aggregation violations occur when aggregated numerical values across records violates a logical rule or constraint. These constraints are often defined by specific logical expressions or rules that must be satisfied for the data to be considered valid.

NCC plays a vital role in diverse real-life applications, thus a case study approach using a business database is seen suitable for this project. Specifically, the proposed solution scope spans three primary areas: violation detection during data insertion, storage organization, and data handling for queries in SQL, including cases of table joins and aggregations. In each of these areas, we will explore and demonstrate how our proposed solutions can function effectively in handling imprecise data in the aforementioned NCC applications.

It is expected that the project's potential advantages can be applied to a wider range of other constraint violations and errors. Through the implementation of constraint checks and the proper organization of data records, we can continue database operation even with imprecise data. There is also again the potential for increased information retention from unrejected rows, which could lead to richer data sets and more insightful data analysis or for forensic use case. It is worth

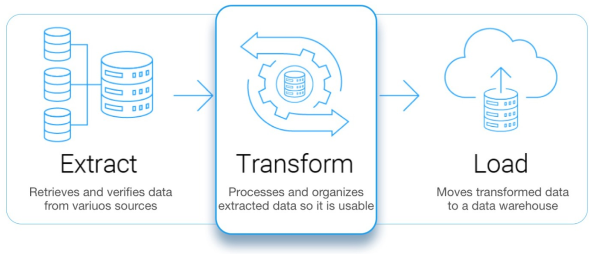

Extract
Retrieves and verifies data from variuos sources
Transform
Processes and organizes extracted data so it is usable
Load
Moves transformed data to a data warehouse

Fig. 3: ETL chart showing where the proposed solution will be deployed

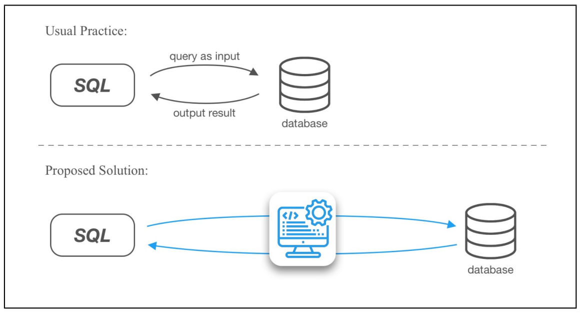


Fig. 4: Illustration to query processing solution

noting that these advantages could be particularly valuable for a wide range of groups and applications, including but not limited to academic researchers and businesses or organizations looking to leverage their data for strategic decisions. Another significant value could arise from cost savings related to data cleaning. This is because the system can continue to function robustly even in the presence of data violations. Organizations then have the option to implement data cleaning, repair, or processing procedures on top of the proposed solutions only as needed or according to their priorities. Lastly, by keeping all data records, it is potentially beneficial to various other use case such as error prevention studies, anomaly or fraud detection, forensics, etc.

In short, this project aims to leverage imprecise data, offering a fresh approach in handling violation of consistency constraints in relational databases. We seek to fill in the gap where traditional database with IC could not fill, and set a stepping stone for further study along this ”living with dirty data” approach. Through the creation of robust pre-processing procedures, it is aspired to improve violation coverage, and separately manage unclean or imprecise data, thus reducing information loss. Focusing on NCC violations, the proposed approach could then be generalized to other types of violations in order to be practical for real-life use. This approach stands to benefit a broad spectrum of stakeholders, from academic researchers to businesses seeking to make strategic decisions based on their data.

III. Literature Review

With the knowledge that real-life data is often dirty and can be costly to organisations, we perform some survey to the literatures that may relate to our approach. Error Detection is the first step in data processing. Numerous studies have been undertaken to identify data errors as violations using integrity constraints (ICs) [6, 7, 8, 9]. A violation occurs when certain data values, when combined, contravene these ICs, indicating them as incorrect. However, constraints don't specify which values within a breach are accurate or incorrect, making them less useful for reliable data correction. Recently, fixing rules [10] have been introduced to accurately pinpoint incorrect values, given sufficient evidence. Nonetheless, such technique needs customized fine-tuning and domain expertise for each use case while also opening the potential interference with other data quality rules. Data repairing is often the next step after error detection and many algorithms have been proposed [11, 12, 13]. Although not directly discussed in our project, our solutions should also serve as a good foundation for data repairing and cleaning afterwards when needed.

Only few researches have explored solutions that allow for the retention of imprecise data while ensuring robust database operation and query precision. We aim to explore novel methods and solutions in the areas of imprecise data storage and tolerant database systems.

Imprecise data storage refers to the storage and management of data that contains inherent imprecision or uncertainty. This type of data storage is becoming increasingly important as many applications do not require high-precision storage for every data structure. For example, in the case of a JPEG file, small errors in the payload data may be tolerable at the cost of some decoding imprecision [14].

One approach to handle imprecise data is the use of interval-valued coefficients in mathematical optimization. Mandal [15] considered imprecise data as a form of interval in nature and presents a solution procedure with imprecise coefficients. The method offers a way to handle imprecise data in optimisation problems. It has been applied to problems such as geometric programming with imprecise coefficients, where the imprecise nature of the data is taken into account during the optimization process. Another approach by Xu et al. [16] discussed precision-constrained approximate queries in wireless sensor networks for object tracking applications. In this method, it presents a scheme that trades answer precision for energy efficiency by keeping error-bounded imprecise location data at designated storage nodes. The degree of data impreciseness is controlled by a parameter called the approximation radius adjusted by an adaptive algorithm. A third approach from a recent study [17] presented a relational data model that can store, manage, and manipulate uncertain data with certain data together, analysed using SQL queries to ensure the completeness of the model. This method recognises that the importance of store and manage imprecise or inexact data is just equally important as for the certain data. This approach aligns with our research goal of creating a storage design and query procedure to retain as much information despite rule violations without the need of data cleaning, repair nor systematic prediction.

Another relevant area of research is the use of imprecise probabilities in engineering problems. Aliche et al. [18] discuss the concept of imprecise probabilities and refer to a reference that provides an overview of developments involving imprecise probabilities for solving engineering problems. While not directly related to tolerant database systems, this research highlights the use of imprecise data in modeling and decision-making processes and shows the importance of it. These approaches may provide frameworks and techniques for representing, querying, and analyzing imprecise data, enabling more flexible and robust data storage and management.

In conclusion, this section highlighted some related methods and solutions in the areas of imprecise data storage and tolerant database systems. We recognize the existing methods, such as geometric programming techniques, which provide a solution for managing vague data in optimization problems. Additionally, Byzantine fault-tolerant transaction processing deals with the challenge of tolerating random faults in duplicated databases. The concept of imprecise probabilities also plays a role in modeling and decision-making processes. Further research is needed to explore and develop specific solutions that address the retention of imprecise data in relational databases while maintaining robust database operation and query precision.

IV. Methodology

Our research will employ a case study approach to evaluate of the proposed solutions. Doing research by case study will involve in-depth examination and analysis of a specific case or a small number of cases within a particular context. This method is expected to help us understand the complexity of the problem and explore solutions with adjustable granularity. Case study research is known to be very useful during exploratory research and theory development phase. Our experiment will include several methods of building the imprecision tolerant database system which results will be measured against each other. Finally, we will discuss our findings and generalize the conclusion to a wider set of applications.

As specified previously, the scope of the experiment is limited to the **Numerical Conditional Constraints (NCC)** using real life scenarios and database schema. We define our steps as follow:

A. Problem Identification

Conventional database systems prioritize data integrity by enforcing constraints, rejecting entire data insertions upon constraint violations. However, with the surge in big data, it is common practice to source information from various origins which inadvertently introduce variations in data structures and organization, necessitating extensive data cleaning efforts and expenses. For various reasons, a datarow may contain one or more values that violates consistency constraints, causing data loss when whole rows are rejected due to isolated column violations. To address this, we aim to develop two key components: storage systems for databases with numeric constraint violations along with its violation information, and procedures for querying and representing information to users.

B. Case Study Formation

In this step, we will use a sample case-study with NCC violations to experiment with our solutions. Our reproduction of database is derived from a ride-hailing or online taxi scenario, where we store daily trip payment information in a database table called `Payment`. The columns in the payment will represent the information normally contained for business operational as well as analytical use. The columns of the table are explained as follows:

- `trip_id` - unique identifier of each data row. Also serves as primary key.
- `driver_id` - unique identifier of the driver providing the service. It is assumed to be a foreign key from the table ”Driver”.
- `Basefare` - numerical column containing the base amount charged for service rendered.
- `Surcharge` - numerical column containing the of surcharge to be added to the basefare.
- `Total` - numerical column containing the total amount of charges by adding basefare and surcharge values together.
- `Duration` - numerical column containing the duration of the trip in hours, to account for drivers’ work hours.

The sample database will have 3 main numerical conditional constraint (NCC) violations to handle across 4 numerical columns (`Basefare`, `Surcharge`, `Total`, `Duration`). We deliberately limit the scope of our research to only three types of violations at the moment, one of which is normally handled with the table constraint, where as the other two are categorized as Numeric Functional Dependency (NFD) [19]. We will elaborate more on them as follows:

1) Range violation : A range violation occurs when a data row contains a numerical value that lies beyond a specified range. Typically, this mechanism is employed to guarantee that the entered data remains within a known acceptable value range. In our specific case study, we make the assumption that both the `Basefare` and `Surcharge` values are non-negative decimal numbers each with a given maximum value. To maintain data integrity, we impose a range constraint of 0 to 100 for the `Basefare` column and 0 to 50 for the `Surcharge` column. Furthermore, as a consequence of the prior constraints, we also implement a range

restriction of 0 to 150 for the `Total` column.

TABLE I: Example of Range Constraints

Basefare	Surcharge	Total	Duration
x	y	-	-

Condition
$0 \leq x \leq 100$
$0 \leq y \leq 50$
$x + y \leq 150$

2) Dependency violations : A dependency violation occurs when a value that is derived from other columns does not match a given formula. In our example, the column `Total` contains numerical value produced as summation of values from `Basefare` and `Surcharge` from the same row. The dependency violation will be raised on all values when they are not consistent with this formula. We will also consider a dependency violation if one of the dependant subject is already marked with other violation.

TABLE II: Example of Dependency Constraints

DriverID	Basefare	Surcharge	Total
-	x	y	z

Condition
$x + y = z$

3) Aggregation violations : This violations occur when an aggregation of values across multiple records violates a certain rule. In our case, the sum of values in a specific column shall not exceed an acceptable range across multiple rows. Unlike range violations that focus on individual rows, aggregation violations may span the entire table. In our case, as an example, we establish a total aggregation value of 8 for the `Duration` column for each driver. This means that if a driver with a specific `driver_id` accumulates more than 8 hours of total trip duration in a day, we would flag all the corresponding rows from the same DriverID as aggregation violations, including the ones already stored. This is due our method not making any assumption regarding where the error value might be among all the aggregated values.

TABLE III: Example of Aggregation Constraints

DriverID	Basefare	surcharge	Duration
x_c	-	-	x_1
x_c	-	-	x_2
x_c	-	-	x_3

Condition
$x_1 + x_2 + x_3 \leq 8$

C. Experiment

After formalizing the case-study and building the sample database, we will then attempt to reproduce the NCC violations in each case-study. The experiment will explore algorithmic and logical approaches to address storage, query, and mathematical aggregation challenges within the scope of relational database implementation. For the scope of our experiment, MySQL is chosen to be the platform where we implement storage and handling solutions. The sample database will consist of 10,000 rows (see appendix 1) containing all types of NCC violations outlined above. We believe the 10,000 rows shall provide ample information and measurable values regarding the performance of each proposed solutions. In addition to the main insert statements, we will attempt to insert 10,000 rows including around 2,000 rows of only single type of violations, to examine the performance of every solution in handling each violation we cover.

Next we move on to query representation on the main 10,000 rows with all 3 types of NCC violations. We proceed to test each solution by running the same set of test queries. These queries cover a range of types, including basic SELECT statements, computational queries, and table join queries. This testing helps us evaluate the utility of each proposed solution and show how effectively they present information to users. Our experiment will not include parallel computing and distributed system in this study, leaving them open for future researches.

D. Output

For each proposed solution, we will observe the output across several key aspects and generate measurements using the following metrics:

1) **Coverage:** We'll determine how effectively each solution handles the three defined NCC violation types and assess the level of information each solution provides about these violations.
2) **Speed:** We'll evaluate the performance of each solution by processing the same 10,000 insert statements and executing a same set of test queries. Multiple iterations will be conducted, and the time taken by each solution will be compared to a baseline reference and among each other. The duration of execution will be determined by recording the start time before the script begins running and noting the end time when the execution is completed. Our precision level for each time duration is set to a tenth of a second.
3) **Storage:** After performing the storage organization for 10,000 rows, we will measure the storage it takes for each table. Again this values will be compared to each other alongside a baseline reference. For our case study, we use the mySQL data length value and its unit.
4) **Utility:** Lastly, we'll discuss the user benefits offered by each solution. Our experiment aims to yield both quantitative data, showcasing the amount of new information retained compared to the baseline database, and qualitative insights into the potential advantages users can gain as well as the robustness of the proposed database system.

It is anticipated that each solution will involve trade-offs, which will be assessed based on their utility and suitability for real-life applications. The specific solutions and their outputs from our case study, will then be generalized to other types violations and other applications in the discussion and conclusion part of this report.

V. Solution

In this experiment, we propose three separate solutions to effectively handle imprecise information. These solutions include algorithm to store and organize the data as well as procedures for future querying and user presentation. The three solutions are namely **"Imprecise Tagging Method"**, **"Precision Mapping Method"** and lastly **"Imprecise Table Method"**, and will be referred to as Solution A, B and C respectively. In each method, our main emphasis is on exploring the comprehensive and attainability of each solutions. We would run experiments and analyse the performance based on the pre-determined metrics and discuss their differences. We note that the solutions are preliminary and is designed to provide distinct level of trade-offs, while reserving optimization for future research.

A. Imprecise Tagging Method

The first method is to perform tagging on the numerical violations using two additional columns. The first column, called "imprecise_status," uses a binary value to tell us if there is any violations within each row. The second new column, "imprecise_tags," holds a list of codes that describe the level of precision for each numerical column. These precision codes are represented as numerical values, and they indicate the type of violation present in each column. For example, if a row has no violations, `imprecise_status` would be set to 0, and `imprecise_tags` might look like [0, 0, 0, 0] for four numerical columns, denoting no violation. Depending on the needs, we can customize these codes to handle different types of imprecision, which is only 3 for our specific case. This can be expanded to accommodate other types of violations by expanding the code length.

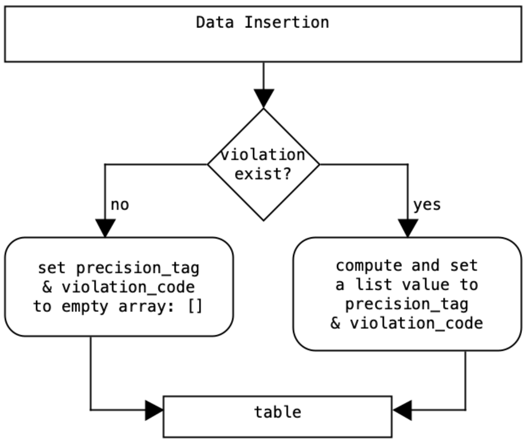

Data Insertion
violation exist?
no
yes
set precision_tag & violation_code to empty array: []
compute and set a list value to precision_tag & violation_code
table

Fig. 5: Flow chart of method A

First, we created a procedure to organize the insertion of data rows in mySQL or perform update if the primary key exists. The procedure will check for the three types of NCC violations and assign values to both `imprecise_status` and `imprecise_tags` columns. The procedure is able to perform error checking on every insertion and provide the imprecise tags for the columns of interests. To handle aggregation violation, the procedure will also update existing rows in the table if aggregation violation is discovered on a new row. This process require the SQL_safe_update to be set to 0. As illustrated in Table VII, the imprecise tag is capable of offering precision information at both the row and column levels while preserving the complete dataset within the database. To facilitate query processing, we have developed an additional procedure that accepts a standard query and an additional parameter indicating whether the user desires to include imprecise values. When users opt for precise information only, the query procedure examines the precision status, retaining what we refer to as "sure information" and replacing imprecise cells with null values in a temporary table. Conversely, if users specify otherwise, the procedure includes all available information to the temporary table and then process the query from the said table. This method has been designed to ensure that the maximum amount of information and violation details are retained for query processing.

TABLE IV: Solution A Example TABLE

Trip ID	Driver ID	Base Fare	Sur-charge	Total	Dura-tion	Imprecise Status	Imprecise Tags
1	DRIVER1	99	10	109	1.5	1	[0,0,0,3]
2	DRIVER2	40	10	50	0.5	0	[]
3	DRIVER1	99	-10	89	1.5	1	[0,1,2,3]
4	DRIVER1	99	10	100	1	1	[0,0,2,3]
5	DRIVER1	99	10	109	5	1	[0,0,0,3]

B. Precision Mapping Method

In the given dataset, there are multiple columns, each containing numerical values. Accompanying these data columns are corresponding `imprecise_mapping` columns, which are uniquely named based on the original data column labels. These `imprecise_mapping` columns are used to identify instances of rule violations within the dataset, akin to boolean masking.

For example, consider a dataset with various columns. If a rule violation occurs in a specific column for a particular row, the corresponding `imprecise_mapping` column for that row is marked with a 1, indicating the location of the violation. In contrast, the `imprecise_mapping` columns for other data columns in the same row will remain at 0. This approach enables precise tracking and identification of rule violations across different data columns within the dataset. We tested this solution by creating an insertion procedure to check for rule violations and then provide binary values to the corresponding `imprecise_mapping` columns. The second procedure is used to process query by taking the precision parameter specified by the user, to then read the precision mapping and filter the precise values into a temporary table where the query will be run.

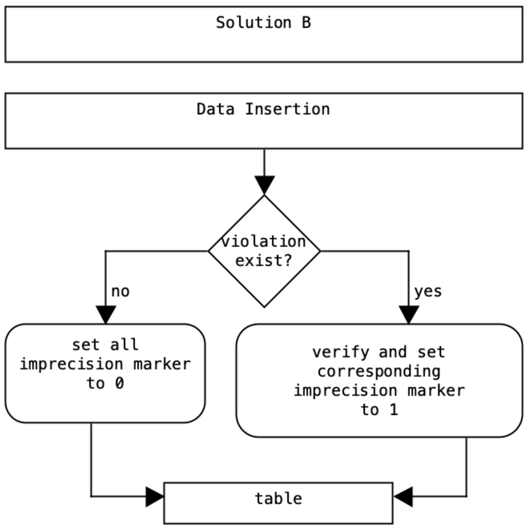

Solution B
Data Insertion
violation exist?
no
yes
set all imprecision marker to 0
verify and set corresponding imprecision marker to 1
table

Fig. 6: Flow chart of method B

TABLE V: Solution B Example TABLE

Trip ID	Driver ID	Base Fare	Sur-charge	Total	Dura-tion	Im-precise Base-Fare	Im-precise Sur-charge	Im-pre-cise To-tal	Im-pre-cise Du-ra-tion
1	DRIVER1	99	10	109	1.5	0	0	0	1
2	DRIVER1	130	10	140	1.5	1	0	1	1
3	DRIVER1	99	-10	89	1.5	0	1	1	1
4	DRIVER1	99	10	100	1.5	0	0	1	1
5	DRIVER1	99	10	109	2.5	0	0	0	1

In the presented data table, certain violations of predefined rules are detected, resulting in the activation of the imprecise columns. Specifically, on Trip 2 the `BaseFare` value exceeds the allowable range, causing the `Imprecise_BaseFare` column to turn to 1. Similarly, on Trip 3 a surcharge violation is identified, leading to the `Imprecise_Surcharge` column being set to 1. Due to the dependency between these columns, the `Imprecise_Total` column is also set to 1 in these cases, signifying an aggregate violation.

Furthermore, it is observed that the total hours for trips conducted by **Driver1** have exceeded the limit of the aggregation function, where no trip should be extended beyond 8 hours per day per driver. Consequently, for all rows pertaining to Driver1 the `Imprecise_Duration` column is uniformly set to 1 to indicate the violation of this rule. These imprecision mappings enable the precise identification of rule violations within the dataset.

In summary, we can provide a concise overview of the solution using a generalized table below as Table VI:

*Note: the imprecise value will be marked by **.*

TABLE VI: Solution B

ID	Group by	A	B	C (=A+B)	Ag-ge-ration	Im-precise A	Im-precise B	Im-precise C	Im-precise Aggr
1	VALUE1	#	#	#	#	0	0	0	1
2	VALUE1	**	#	#	#	1	0	1	1
3	VALUE1	#	**	#	#	0	1	1	1
4	VALUE1	#	#	**	#	0	0	1	1
5	VALUE1	#	#	#	#	0	0	0	1

C. Imprecise Table Method

The third method is to separate the imprecise data and store them in a imprecise table from the the original table. In this method, two tables are employed: one to store imprecise data and another to store precise data. The imprecise data table operates without any constraints, allowing for the accommodation of data that may not conform to predefined rules or accuracy standards. Conversely, the original table, reserved for precise data, upholds all the constraints inherent to the scheme's design, maintaining data integrity and consistency.

To determine the placement of each incoming data row, a dedicated procedure is invoked. This procedure assesses the row's precision status, as well as its relationships with other rows, and makes an informed decision regarding its placement in either the imprecise or precise table. By employing this approach, users can efficiently handle NCC data, which may have different levels of accuracy, while ensuring the overall structure of the database remains intact. This straightforward method is designed to optimize the trade-off between speed, storage usage, and information accuracy, among other considerations. However, to achieve this, we have chosen to compromise on column-level precision, separating an entire row when a violation is detected.

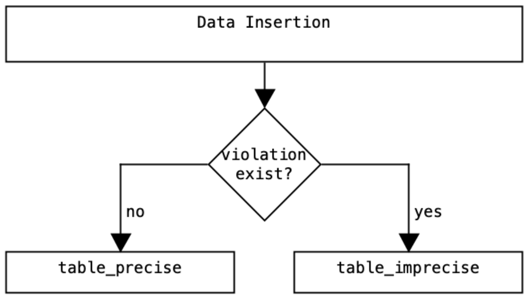


Fig. 7: Flow chart of method C

To process query, we create a procedure that concatenate the imprecise table to the precise table prior to running the query unless a precise parameter is specified.

VI. Results

To confirm the expected results and thoroughly assess the cost-effectiveness and data reliability of the three solutions mentioned (Solution A, Solution B, and Solution C), we

TABLE VII: Solution C Example TABLE

(a) Precise table

Trip ID	Driver ID	Base Fare	Surcharge	Total	Duration
1	DRIVER1	99	10	109	1.5
2	DRIVER2	40	10	50	0.5

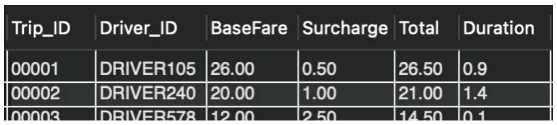
(b) Imprecise table

Trip ID	Driver ID	Base Fare	Surcharge	Total	Duration
3	DRIVER1	99	-10	89	1.5
4	DRIVER1	99	10	100	1

carried out an experiment using the MySQL platform. This experiment included a detailed comparison of these solutions with a reference database, which follows traditional relational database principles and maintains only precise data through constraints. Throughout our investigation, we took special care to examine the trade-offs that emerge within each performance metrics when emphasizing data retention.

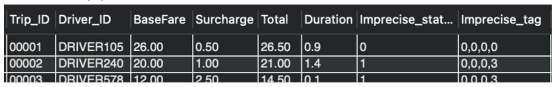
The assessment encompassed multiple performance metrics, including NCC violations coverage, speed, storage, and utility.

1) **NCC Violations Coverage**



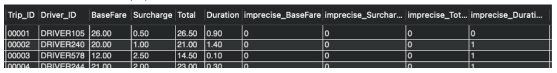
Trip_ID	Driver_ID	BaseFare	Surcharge	Total	Duration
00001	DRIVER105	26.00	0.50	26.50	0.9
00002	DRIVER240	20.00	1.00	21.00	1.4
00003	DRIVER578	12.00	2.50	14.50	0.1

(a) Reference Database - Select * Table



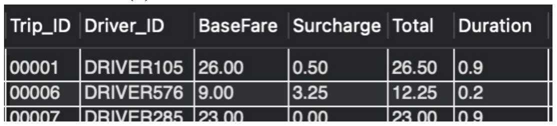
Trip_ID	Driver_ID	BaseFare	Surcharge	Total	Duration	Imprecise_stat...	Imprecise_tag
00001	DRIVER105	26.00	0.50	26.50	0.9	0	0,0,0,0
00002	DRIVER240	20.00	1.00	21.00	1.4	1	0,0,0,3
00003	DRIVER578	12.00	2.50	14.50	0.1	1	0,0,0,3

(b) Solution A - Select * Table



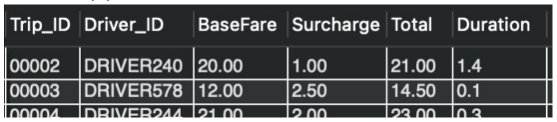
Trip_ID	Driver_ID	BaseFare	Surcharge	Total	Duration	imprecise_BaseFare	imprecise_Surchar...	imprecise_Tot...	imprecise_Durati...
00001	DRIVER105	26.00	0.50	26.50	0.90	0	0	0	0
00002	DRIVER240	20.00	1.00	21.00	1.40	0	0	0	1
00003	DRIVER578	12.00	2.50	14.50	0.10	0	0	0	1
00004	DRIVER244	21.00	2.00	23.00	0.30	0	0	0	1

(c) Solution B - Select * Table



Trip_ID	Driver_ID	BaseFare	Surcharge	Total	Duration
00001	DRIVER105	26.00	0.50	26.50	0.9
00006	DRIVER576	9.00	3.25	12.25	0.2
00007	DRIVER285	23.00	0.00	23.00	0.9

(d) Solution C - Select * Precise Table



Trip_ID	Driver_ID	BaseFare	Surcharge	Total	Duration
00002	DRIVER240	20.00	1.00	21.00	1.4
00003	DRIVER578	12.00	2.50	14.50	0.1
00004	DRIVER244	21.00	2.00	23.00	0.3

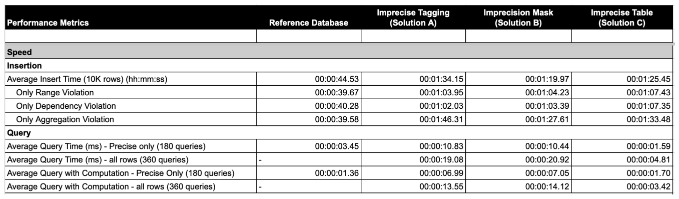
(e) Solution C - Select * Imprecise Table

Fig. 8: Violation Coverage Results

Note: these are partial capture of tables from each solution. The table sources can be found in Appendix B.

By inserting 10,000 rows of sample data that contain all types of NCC violations, the above experiment result was obtained. Solution A, Solution B, and Solution C all successfully exhibit the ability to capture range, dependency and aggregation violations, outperforming the reference database. As expected, Solution A offers detailed information on accurately showing the imprecise status and reflecting types of violation with imprecise tag, while Solution B also provides the imprecise status of each data cell. In Solution C, the rows with imprecise data are separately stored in imprecise table without additional labelling while precise data are kept in precise table.
Solutions A, B and C covers all three types of violations which significantly improve violations coverage compared to the traditional approach applied in reference database, with Solution A standing out for its detailed violation reporting.

2) **Speed**



Performance Metrics	Reference Database	Imprecise Tagging (Solution A)	Imprecision Mask (Solution B)	Imprecise Table (Solution C)
Speed				
Insertion				
Average Insert Time (10K rows) (hh:mm:ss)	00:00:44.53	00:01:34.15	00:01:19.97	00:01:25.45
Only Range Violation	00:00:39.67	00:01:03.95	00:01:04.23	00:01:07.43
Only Dependency Violation	00:00:40.28	00:01:02.03	00:01:03.39	00:01:07.35
Only Aggregation Violation	00:00:39.58	00:01:46.31	00:01:27.61	00:01:33.48
Query				
Average Query Time (ms) - Precise only (180 queries)	00:00:03.45	00:00:10.83	00:00:10.44	00:00:01.59
Average Query Time (ms) - all rows (360 queries)	-	00:00:19.08	00:00:20.92	00:00:04.81
Average Query with Computation - Precise Only (180 queries)	00:00:01.36	00:00:06.99	00:00:07.05	00:00:01.70
Average Query with Computation - all rows (360 queries)	-	00:00:13.55	00:00:14.12	00:00:03.42

Fig. 9: Performance Metrics - Speed

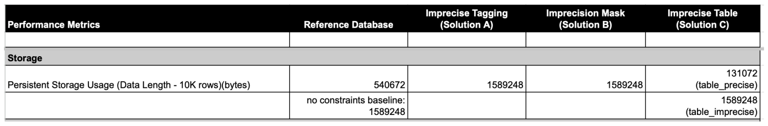
Fig 9 shows speed experiment results that are measured in seconds with format (hh:mm:ss) for the insert time and milliseconds (hh:mm:ss.ms) for the query time:

- **Average Insert Time:** The reference database boasts the fastest overall average insert time at around 44.53 seconds, while solutions (A, B, and C) that retain both precise and imprecise data incur higher overhead. However, Solution B only performs slightly better than Solution A and Solution C. Overall this trend is seen across all violations.
  - **Only Range Violation Insert Time:** The reference database performs the best, while Solution A exhibits a slightly better performance compared with Solution B and C.
  - **Only Dependency Violation Insert Time:** The reference database is the fastest, and Solution A still outperforms Solution B and Solution C in this category.
  - **Only Aggregation Violation Insert Time:** The reference database is still the fastest, Solution B is slightly faster than Solution A and Solution C.
- **Average Query Time**
  - **With Precise Data Only:** Solution C is significantly faster than Solution A, B and perform very close to the reference database.
  - **With All Rows:** Considering both precise and imprecise data, Solution A and B take longer time than Solution C.

- **Average Query with Computation - SUM, AVG, MIN, MAX**
  - **With Precise Data Only:** Solution C is the fastest again nearing the reference database, followed by Solution A and B having the same speed.
  - **With All Rows:** Considering both precise and imprecise data, Solution A and B still have the same performance, and Solution C is again the fastest among the three solutions.

The experiment's speed comparison demonstrates that the reference database excels in insert times and maintains a speed advantage in various scenarios because it directly rejected rows containing imprecise data. On the other hand, Solution A and B introduce imprecise status and tagging filtering, causing precise data queries to take longer than solution C. With its simpler imprecise row separation approach, Solution C consistently stands out as the fastest in general and computational query tasks, making it a valuable choice when query performance is critical, albeit not at column level. It should be noted that Solution A offers the same ability to perform row level precision and may perform similarly to Solution C in expense of some data retention.

3) **Storage**



Performance Metrics	Reference Database	Imprecise Tagging (Solution A)	Imprecision Mask (Solution B)	Imprecise Table (Solution C)
Storage				
Persistent Storage Usage (Data Length - 10K rows)(bytes)	540672	1589248	1589248	131072 (table_precise)
	no constraints baseline: 1589248			1589248 (table_imprecise)

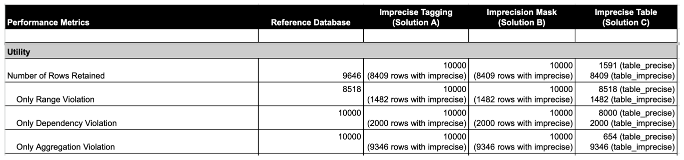
Fig. 10: Performance Metrics - Storage

Fig 10 shows storage experiment results that are measured by mySQL data length unit (bytes):

- **Persistent Storage Usage:** The reference database minimizes storage usage by rejecting rows containing imprecise data, resulting in a compact data length of 540,672 bytes. In contrast, Solution C takes up the most storage space for maintaining two tables, while Solution A and B also require significantly more storage, each with a data length of 1,589,248 bytes.
  However, if the reference database operates without constraints, it will keep all rows and maintain roughly the same storage requirements as Solution A and B.

The solutions (A, B, and C) that retain both precise and imprecise data all introduce overhead but offer benefits in terms of data completeness. The solutions does not demand substantially more storage space compared to the reference database without constraints, which means the cost of storage will only be correlated to the amount of data retained and not the method. Solution A maintains storage requirements similar to Solution B, even with more detailed violation information. This observation proves our assumption that, with the given sample dataset, there is clear trade-off between having richer data to storage space.

4) **Utility**



Performance Metrics	Reference Database	Imprecise Tagging (Solution A)	Imprecision Mask (Solution B)	Imprecise Table (Solution C)
Utility				
Number of Rows Retained	9646	10000 (8409 rows with imprecise)	10000 (8409 rows with imprecise)	1591 (table_precise) 8409 (table_imprecise)
Only Range Violation	8518	10000 (1482 rows with imprecise)	10000 (1482 rows with imprecise)	8518 (table_precise) 1482 (table_imprecise)
Only Dependency Violation	10000	10000 (2000 rows with imprecise)	10000 (2000 rows with imprecise)	8000 (table_precise) 2000 (table_imprecise)
Only Aggregation Violation	10000	10000 (9346 rows with imprecise)	10000 (9346 rows with imprecise)	654 (table_precise) 9346 (table_imprecise)

Fig. 11: Performance Metrics - Utility

Fig 11 shows utility experiment results that are measured in number of rows:

- **Number of Rows Retained:** The reference database, which enforces strict constraints, retains fewer rows (I.e., 9,646 out of 10,000 rows), discarding those with imprecise data. These remaining rows also still contain violations that the reference table is unable to cover. In contrast, all three solutions retain the entire dataset, preserving imprecise data.
  - **Only Range Violation:** When it comes to range violations, the reference database discards all 2,000 rows with violations, while all three solutions retain all rows.
  - **Only Dependency Violation:** For dependency violations, the reference database retains all rows due to inability to coerce this constraint. Similarly, all three solutions retain the entire dataset.
  - **Only Aggregation Violation:** The reference database retains most of the data, only discarding rows with single-row range-violations in the duration column, as opposed to aggregating multiple rows prior to evaluating the rule. This pose inconsistency issue, by keeping some potentially unsure data in the table. In contrast, all three solutions retain all rows.

All three solutions (A, B, and C) outperform the reference database in terms of retaining data. While the reference database rigorously enforces constraints and discards rows with imprecise data, it can not handle the dependency and aggregation violations correctly when performing single insertion. However, the three proposed solutions retain all data, including rows with violations. The result indicates the three proposed solutions successfully maximize data retention, which can be valuable for various purposes, such as data mining and analytics. From the perspective of operational database, we can observe that all solutions are able to operate at precise level with added violations coverage. They also have the capability to incorporate imprecise data when specified in the query parameter, a feature that standard database systems with constraint enforcement lack. Additionally, for data warehousing purposes, no data is forfeited while offering violation information for all rows. This approach significantly accelerates query processing when compared to the conventional method of iterating through all rows and conducting violation checks for each query.

VII. Discussion

A. Potential Advantages and Disadvantages

1) Reference Database
The Reference Database excels in terms of speed and data consistency, making it suitable for various business operations, including serving as a working and recording database. Our experimental results clearly demonstrate its ability to process data swiftly. However, the Reference Database has limitations in terms of data consistency. It retains the minimal amount of information and lacks error tolerance, which can be disadvantageous in scenarios where more data is preferred. Additionally, it cannot handle dependency and aggregation violations without the use of additional triggers or periodic verification. This serves as the foundation against which our solutions can offer its core value to users by enabling the retention of more information and maintaining precision integrity as needed. An alternative approach is to store all data within a conventional database without any constraints which is usable in record only database. This approach requires time-consuming violation checks process for every query and may not be suitable for operational use.
2) Imprecise Tagging Method (Solution A)
Imprecise Tagging (Solution A) outperforms in maintaining all available information, even the rows include error details, and provides imprecise status and tagging to support both column and row-level precision, making it well-suited for data analytics and operational use cases. The detailed error tags can also support customized filtering during query, for example when the user wishes to only ignore certain types of violation but not all.
Nevertheless, Solution A shows slower performance due to error checking and tagging during each insertion. The parsing of imprecise tags for every query also resulted in relatively slower speed as a major trade-off.
3) Precision Mapping Method (Solution B)
Precision Mapping (Solution B) also maintains all information while offering valuable error location details. It shares the ability to support both column and row-level precision and is particularly advantageous for analytics. Unfortunately, similar to Solution A, Solution B incurs a performance penalty due to error checking during every insertion. When performing query representation, it was expected to perform better due to its mapping design that allows for matrix operation. However, our experiment results using MySQL indicate almost identical result to solution A or negligible difference. Solution B is lacking in error type detail to solution A but retain column level precision over solution C.
4) Imprecise Table Method (Solution C)
Separate Imprecise Table (Solution C) stands out by preserving all data along with precision status information and favoring row-level precision. It also boasts a notable speed advantage, with insert and query time closer to our reference database system. However, Solution C lacks support for column-level precision since it does not store specific column location of the imprecise data. Depending on application, solution C may be seen as superior when speed is the utmost priority.

B. Limitations of Proposed Solutions

The experiment results highlight the data retention benefit provided by the three solutions, it is also important to note that achieving this retention comes at several costs:

- Lower Data Process Efficiency:
All three solutions (A, B, and C) require longer data processing time, especially during data insertion and some query processing, due to the need to handle imprecise data. This may not be suitable for real-time or high-throughput applications.
- Increased Storage Requirements:
All three solutions (A, B, and C) require significantly more storage space compared to a reference database with constraints. This increase in storage can be a limitation, especially for organizations with limited storage resources. As the cost of storage may change over time, this decision is left to the user consider.
- Potential Resource-Intensive Maintenance:
Each of the three solutions (A, B, and C) retains imprecise data, which implies that they might store inaccurate or conflicting information. Depending on the specific application, these solutions may not be entirely robust because they solely identify and isolate imprecise values without automated repair or cleaning capabilities. In cases where high data accuracy is essential, additional maintenance efforts may be required to effectively manage imprecise data. This could involve periodic error correction and continuous monitoring of data quality over time, which can be resource-intensive.

In short, the limitations highlight the trade-offs between data retention and operation performance. Solutions A, B, and C offer enhanced NCC violations coverage but generally introduce overhead in terms of insertion time, query time, and storage. The three solutions stand out for its detailed violation reporting, which may be valuable in certain use cases, but limitations may incur. The choice of a data handling technique should be based on the specific requirements of the application, balancing the need for data completeness and data accuracy against operation performance.

C. Comparison to Traditional Approaches – Data Cleaning and Data Prediction

When considering the merits of Imprecise Tagging, Precision Mapping, and Separate Imprecise Table (Solution A, B, and C) in data handling, it's valuable to compare them to other traditional approaches, such as data cleaning and data prediction. We assume the violation coverage can be expanded to include all other types of violations outside NCC that IC language can cover as well as some of the more complex ones. This implies there is no generalization issue for the methods when compared to the capability of the reference database system.

Data cleaning is a conventional approach to data management that prioritizes data precision by directly eliminating or correcting imprecise data. In the pursuit of precision, some data points or records may be deemed inconsistent or erroneous and consequently removed from the dataset. Therefore, it introduces the risk of data loss, especially when data standards are stringent. When compared to our proposed solutions, the data cleaning can be put off until necessary while having the database running on specified precision level according to the user specification.

On the other hand, data prediction offers an alternative strategy by estimating missing or imprecise data values using algorithms, prioritizing data completeness while maintaining accuracy. We have observed various attempt in this approach using same row information, or trained machine learning models. However, this highly depends on the accuracy and effectiveness of the prediction algorithm, which may not eliminate errors altogether into the data set if it is not properly tuned.

In contrast, in addition to preserving precise data, the three proposed solutions prioritize data retention and completeness by also preserving all imprecise data. Specifically, Solution C maintains imprecise information while providing row level selection, catering to robust operational use. Solution B extends this approach by retaining all data and offering error location information, which is particularly advantageous for analytics-driven applications. Solution A offers a step further in preserving all data along with each row's precision status and identifying error location and types, albeit with some trade-off in computational speed.

Essentially by retaining potentially valuable imprecise data, Solutions A, B, and C are able to minimize potential data loss and provide a more flexible and valuable approach to a variety of applications. These solutions can be paired with conventional data cleaning or auto-regressive model to repair the errors to further maximize the value added.

D. Potential Applications and Benefits

The discoveries made in evaluating Solutions A, B, and C have far-reaching implications for various applications and offer substantial benefits to a wider range of use cases in different aspects.

1) **Data Warehousing – Imprecise Data Analytics:** Solutions A, B, and C, designed to retain imprecise data, find their niche in data warehousing and analytics. By preserving a more extensive dataset that encompasses both precise and imprecise data, these solutions offer a valuable asset for data analysts and scientists. This approach enables the consolidation of data from multiple sources, resulting in a larger dataset that opens doors to the discovery of valuable insights and trends, even within imprecise data points. For instance, a retail company utilizing Solution C to store transactional data can benefit from including imprecise data such as incomplete customer information or irregular purchase records. This expanded dataset supports in-depth customer analytics and enhances decision-making. Furthermore, with its capacity to retain imprecise data and its error details, users can uncover insights related to error prevention, database design improvement and the development of automated solutions.
2) **Cost Reduction in Data Cleaning:** The proposed solutions can significantly reduce the cost of data cleanup efforts by enabling the system to work on precise data despite storing imprecise data. By adopting Solutions A, B, or C, organizations can redirect resources and focus data cleaning efforts on more critical data points. This optimized approach ensures that data cleaning efforts are focused where they matter most, potentially increasing efficiency.
3) **Operational Databases – Data Accuracy:** In operational databases, where real-time data processing and data accuracy are important, the reference database with strict constraints may be preferred. However, there are scenarios where operational databases require data completeness as well. For example, for research purposes, a healthcare company may implement Solution A or B alongside their operational database to retain incomplete historical patient data. Researchers can identify valuable data from imprecise status and tagging. These insights enhance research accuracy, contribute to long-term health trend analysis and advancements in medical research.
4) **Machine Learning and AI:** Machine learning and artificial intelligence applications can greatly benefit from solutions that preserve imprecise data. These solutions provide larger data sets, potentially leading to more robust model training and thus potentially better predictive capabilities. Our solutions may also offer a better method to segregate and handle noise points in input data. Nevertheless, it is crucial to exercise with caution when handling imprecision during model development and deployment to ensure that machine learning models can effectively account for and learn from imprecise data. Known regularization methods may help minimize the impact of data with identified imprecise characteristics.

VIII. Conclusion

As we strive to preserve the most amount of data, we inevitably face trade-offs concerning coverage, speed, storage, and depending on each scenario, utility. This report introduces three solutions that allow for the retention of imprecise data, albeit with some observed compromises in certain metrics. Moreover, each solution presents its own set of pros and cons, contingent upon its specific design.

From our experiment, the Imprecise Tagging Method (Solution A) emerges as the most optimal choice due to its multiple precision level and detailed violation information capability. Alternatively, if prioritizing speed is critical, the Imprecise Table method (Solution C) can be implemented, albeit at the expense of precision level capability. Lastly, while the Precision Mapping Method (Solution B) yields outcomes akin to Solution A in MySQL, it may have potential benefit for systems designed for efficient matrix computation. In conclusion, we affirm that operating with imprecise data is feasible, and robust performance can be achieved even in the presence of such data.

From our study, we studied the potential benefits and applications of these precision tolerant database systems in record management, operational database, data analytics and even forensics. Considering the trade offs, substantial benefits may arise from richer dataset, lower maintenance expense and availability of violation details. As a disclaimer, our solutions are tested to handle NCC through a single case study. It is by no means exhaustive within the NCC domain, let alone all types of violations. However, we conclude that the general design can serve as a good baseline to building more comprehensive solutions.

For future research, implementing the three solutions to real-time data and operation would allow us to gain a much deeper insights. This experiment serves as a foundational study and may benefit from further study in its design and method optimization. Conducting experiments on a larger dataset and diverse data types, incorporating various violation types covered in the integrity constraint language, along with more complex rules, could also prove immensely advantageous. Additionally, Hybrid approaches with other data cleaning or data repair methods may potentially offer reduced trade offs while retaining the benefits. Furthermore, a seamless interface for our storage and query solution can be developed to improve usability. This can be in a shape of intermediary application where users can input a form to modify the conditional functions or specify the types of violations and precision level for query processing. All these can be pursued in future work.

References

[1] E. F. Codd, "A relational model of data for large shared data banks," Communications of the ACM, vol. 13, no. 6, pp. 377–387, 1970.

[2] R. Ramakrishnan and J. Gehrke, "Database systems: Design, implementation, and management," McGraw-Hill, 2003.

[3] I. F. Ilyas and X. Chu, Data Cleaning. Association for Computing Machinery, 2019.

[4] F. Ridzuan and W. M. N. Wan Zainon, "A review on data cleansing methods for big data," Procedia Computer Science, vol. 161, pp. 731–738, 2019, the Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050919318885

[5] X. Chu, I. Ilyas, and P. Papotti, "Holistic data cleaning: Putting violations into context," 04 2013, pp. 458–469.

[6] M. Arenas, L. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases," in Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 1999, pp. 68–79.

[7] L. Bravo, W. Fan, and S. Ma, "Extending dependencies with conditions." in VLDB, vol. 7, 2007, pp. 243–254.

[8] J. Chomicki and J. Marcinkowski, "Minimal-change integrity maintenance using tuple deletions," Information and Computation, vol. 197, no. 1-2, pp. 90–121, 2005.

[9] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for capturing data inconsistencies," ACM Transactions on Database Systems (TODS), vol. 33, no. 2, pp. 1–48, 2008.

[10] J. Wang and N. Tang, "Towards dependable data repairing with fixing rules," in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014, pp. 457–468.

[11] G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David, "Modeling and querying possible repairs in duplicate detection," Proceedings of the VLDB Endowment, vol. 2, no. 1, pp. 598–609, 2009.

[12] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in Proceedings of the 33rd international conference on Very large data bases, 2007, pp. 315–326.

[13] W. Fan, S. Ma, N. Tang, and W. Yu, "Interaction between record matching and data repairing," Journal of Data and Information Quality (JDIQ), vol. 4, no. 4, pp. 1–38, 2014.

[14] J. Bornholt, R. Lopez, D. M. Carmean, L. Ceze, G. Seelig, and K. Strauss, "A dna-based archival storage system," ACM SIGARCH Computer Architecture News, vol. 44, pp. 637–649, 2016.

[15] T. Mandal, "Posynomial parametric geometric programming with interval valued coefficient," Journal of Optimization Theory and Applications, vol. 154, pp. 120–132, 2012.

[16] J. Xu, X. Tang, and W. Lee, "A new storage scheme for approximate location queries in object-tracking sensor networks," Ieee Transactions on Parallel and Distributed Systems, vol. 19, pp. 262–275, 2008.

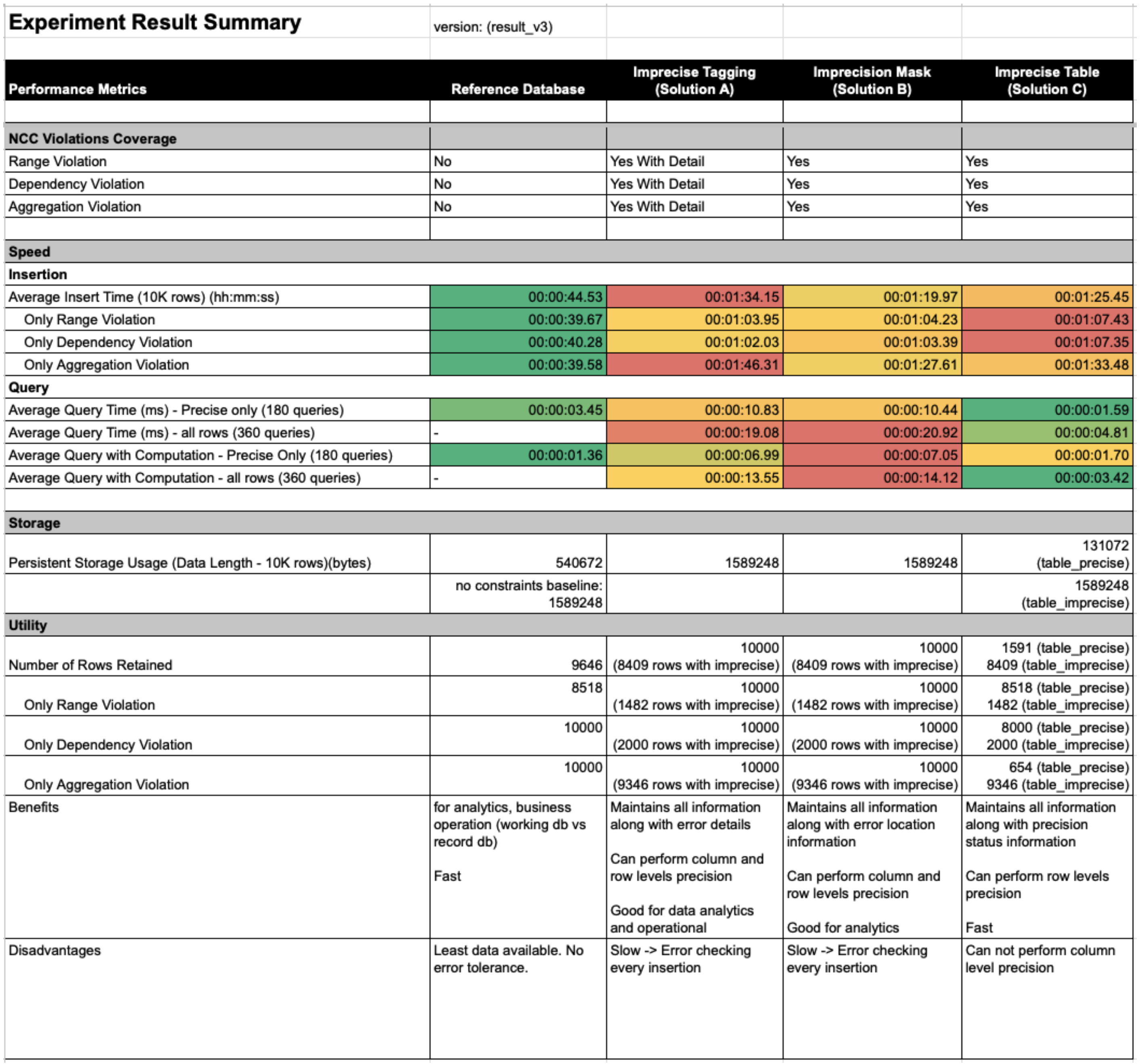
[17] S. Channar, "A relational data model for uncertain data," International Journal of Emerging Multidisciplinaries Computer Science Artificial Intelligence, vol. 1, pp. 46–55, 2022.

[18] A. Aliche, H. Hammoum, K. Bouzelha, and N. Hannachi, "Development and validation of predictive model to describe the growth of concrete water tank vulnerability with time," Periodica Polytechnica Civil Engineering, 2016.

[19] G. Fan, W. Fan, and F. Geerts, "Detecting errors in numeric attributes," Web-Age Information Management, p. 125–137, 2014.

APPENDIX

APPENDIX A
EXPERIMENT SUMMARY TABLE



Experiment Result Summary version: (result_v3)

Performance Metrics	Reference Database	Imprecise Tagging (Solution A)	Imprecision Mask (Solution B)	Imprecise Table (Solution C)
NCC Violations Coverage				
Range Violation	No	Yes With Detail	Yes	Yes
Dependency Violation	No	Yes With Detail	Yes	Yes
Aggregation Violation	No	Yes With Detail	Yes	Yes
Speed				
Insertion				
Average Insert Time (10K rows) (hh:mm:ss)	00:00:44.53	00:01:34.15	00:01:19.97	00:01:25.45
Only Range Violation	00:00:39.67	00:01:03.95	00:01:04.23	00:01:07.43
Only Dependency Violation	00:00:40.28	00:01:02.03	00:01:03.39	00:01:07.35
Only Aggregation Violation	00:00:39.58	00:01:46.31	00:01:27.61	00:01:33.48
Query				
Average Query Time (ms) - Precise only (180 queries)	00:00:03.45	00:00:10.83	00:00:10.44	00:00:01.59
Average Query Time (ms) - all rows (360 queries)	-	00:00:19.08	00:00:20.92	00:00:04.81
Average Query with Computation - Precise Only (180 queries)	00:00:01.36	00:00:06.99	00:00:07.05	00:00:01.70
Average Query with Computation - all rows (360 queries)	-	00:00:13.55	00:00:14.12	00:00:03.42
Storage				
Persistent Storage Usage (Data Length - 10K rows)(bytes)	540672	1589248	1589248	131072 (table_precise)
	no constraints baseline: 1589248			1589248 (table_imprecise)
Utility				
Number of Rows Retained	9646	10000 (8409 rows with imprecise)	10000 (8409 rows with imprecise)	1591 (table_precise) 8409 (table_imprecise)
Only Range Violation	8518	10000 (1482 rows with imprecise)	10000 (1482 rows with imprecise)	8518 (table_precise) 1482 (table_imprecise)
Only Dependency Violation	10000	10000 (2000 rows with imprecise)	10000 (2000 rows with imprecise)	8000 (table_precise) 2000 (table_imprecise)
Only Aggregation Violation	10000	10000 (9346 rows with imprecise)	10000 (9346 rows with imprecise)	654 (table_precise) 9346 (table_imprecise)
Benefits	for analytics, business operation (working db vs record db) Fast	Maintains all information along with error details Can perform column and row levels precision Good for data analytics and operational	Maintains all information along with error location information Can perform column and row levels precision Good for analytics	Maintains all information along with precision status information Can perform row levels precision Fast
Disadvantages	Least data available. No error tolerance.	Slow -> Error checking every insertion	Slow -> Error checking every insertion	Can not perform column level precision

Fig. 12: Experiment Summary Table

APPENDIX B
SQL SCRIPTS FILES

- Script for describing the process of the whole experiment step by step:
  - main.sql
- Script for inserting 10K rows containing all types of violations:
  - Insert_BaselineAll.sql
  - Insert_A_ScriptAll.sql
  - Insert_B_ScriptAll.sql
  - Insert_C_ScriptAll.sql
- Script for inserting 10K rows containing Range violation:
  - Insert_Baseline_Script_Range.sql
  - Insert_A_Script_Range.sql
  - Insert_B_Script_Range.sql
  - Insert_C_Script_Range.sql
- Script for inserting 10K rows containing Dependency violation:
  - Insert_Baseline_Script_Dependency.sql
  - Insert_A_Script_Dependency.sql
  - Insert_B_Script_Dependency.sql
  - Insert_C_Script_Dependency.sql
- Script for inserting 10K rows containing Aggregation violation:
  - Insert_Baseline_ScriptAgg.sql
  - Insert_A_ScriptAgg.sql
  - Insert_B_ScriptAgg.sql
  - Insert_C_ScriptAgg.sql
- Non-computational queries:
  - Precise queries:
    - baseline_precise.sql
    - A1_precise.sql
    - B1_precise.sql
    - C1_precise.sql
  - Precise + Imprecise queries:
    - A1_all.sql
    - B1_all.sql
    - C1_all.sql
- Computational queries:
  - Precise queries:
    - baseline_computational.sql
    - A2_precise.sql
    - B2_precise.sql
    - C2_precise.sql
  - Precise + Imprecise queries:
    - A2_all.sql
    - B2_all.sql
    - C2_all.sql