Castle: Causal Cascade Updates in Relational Databases with Large Language Models

Anonymous ACL submission

Abstract

This work introduces ***Castle***, the first framework for schema-only cascade update generation using large language models (LLMs). Despite recent advances in LLMs for Text2SQL code generation, existing approaches focus primarily on SELECT queries, neglecting the challenges of SQL update operations and their ripple effects. Traditional CASCADE UPDATE constraints are static and unsuitable for modern, denormalized databases, which demand dynamic, context-aware updates. ***Castle*** enables natural language instructions to trigger multi-column, causally consistent SQL UPDATE statements, without revealing table content to the model. By framing UPDATE SQL generation as a divide-and-conquer task with LLMs' reasoning capacity, ***Castle*** can determine not only which columns must be directly updated, but also how those updates propagate through the schema, causing cascading updates — all via nested queries and substructures that ensure data confidentiality. We evaluate it on real-world causal update scenarios, demonstrating its ability to produce accurate SQL updates, and thereby highlighting the reasoning ability of LLMs in automated DBMS.

1 Introduction

Relational Database Management Systems (RDBMS) are the foundation of modern information systems, providing reliable storage and efficient retrieval for critical business data. Natural language interfaces to databases, such as Text2SQL approaches, have enabled users to pose complex questions and retrieve answers via generated SQL queries (Zhong et al., 2017; Xu et al., 2017; Yu et al., 2018; Guo et al., 2019; Wang et al., 2019; Scholak et al., 2021). However, these efforts have primarily focused on generating retrieval-focused (SELECT) queries.

To create a complete natural language interface, it is essential to also generate SQL UPDATE commands. While accuracy challenges persist (Yao et al., 2025; Pourreza and Rafiei, 2023), recent work by Li et al. (2024) has pushed towards a more comprehensive Text2SQL framework, incorporating a broader set of SQL commands, including SQL UPDATEs. Nevertheless, a significant challenge arises with cascade updates, where a change in one record requires automatic propagation of modifications to related records, causing a "ripple effect" in the database, particularly in high-performance denormalized databases (Kimball and Ross, 2013; Balmin and Papakonstantinou, 2005) (see Table 1). The advent of massively distributed systems and real-time analytics has increasingly led designers to adopt **denormalized** schemas (Kimball and Ross, 2013), where relational dimensions are flattened to reduce expensive join operations and meet performance targets. For instance, consider a denormalized database of soccer players: in a denormalized database, the table of players also have records about the clubs, such as the club's name and coach. After a player joins a new club (update club name), their coach name needs to be updated, but this information resides in the table of clubs. Thus, we need to update corresponding table entities in a cascading fashion.

This work addresses the task of improving Text2SQL UPDATEs, with a focus on **cascade updates**. We create two cascade update benchmarks using public datasets to test the ability of Text2SQL methods to issue correct update commands under cascades over more than 1 million records. We then introduce ***Castle***, a new framework designed to enable large language models (LLMs) to generate SQL update commands that execute intended modifications and automatically handle causal-driven cascade updates securely and efficiently. A key challenge is uncovering real-world causal relationships between updated fields and other fields. Our approach also prioritizes preserving data confidentiality by utilizing nested query construction in-

Scenario	Denormalized Schema	Normalized Schema
SELECT Query	✓ No joins required	✗ Requires multi-table joins
Update w/ Cascade	✗ Hard to trace and inconsistent updates risks	✓ Easy with relational foreign keys

Table 1: Comparison between denormalized and normalized schemas in practice, symbols indicate relative advantages (✓) and disadvantages (✗).

stead of table data augmented generation. Our experimental results demonstrate the effectiveness of our framework, achieving up to 85% correct updates in our benchmark tasks, consistently outperforming the best baselines, which reach at most 80% correctness, and are often much lower (down to 52%) in complex scenarios.

Our main contributions are as follows:

- ***Castle*** is the first framework tailored specifically for SQL cascade update operations. It treats SQL cascade updates as causal reasoning tasks. With nested structured subqueries, ***Castle*** generates update commands without exposing raw table data, mitigating privacy risks inherent in current table-augmented approaches.
- We propose two datasets for cascading updates that are 100% based on causal relationships from the real world with more than 1 million records.
- ***Castle*** is the first work that systematically studies and evaluates the LLM-assisted SQL Trigger management and generation.

2 *Castle*

Research Question Modern database designs often exhibit performance-driven redundancies, which complicate update operations. Specifically, we are interested in the question: **Can Large Language Models generate accurate cascade update queries correctly given only the database schema?** This research question gives rise to two fundamental challenges:

Identifying Update Targets (C1). When generating `UPDATE` queries, it is essential to determine the specific columns that require modification. From the perspective of LLMs, this involves not only identifying the target column(s) specified in the natural language instruction but also recognizing potential related updates to other columns, which can vary on a case-by-case basis.

Determining Update Values (C2). After identifying the columns to update, the cascade update operation must determine the new values to assign to the corresponding columns. Since these values are not explicitly provided in the natural language instructions, they may need to be inferred from other data entities, posing a significant challenge for LLMs.

Motivating exampling using our Soccer Transfer task. In order to illustrate the challenges, we first introduce our Soccer Transfer dataset schema in Appendix A. Consider the instruction: "Lionel Messi has transferred from Barcelona (code: fc-barcelona) to Paris Saint-Germain (code: fc-paris-saint-germain), update his information." In response to this instruction, an LLM should not only update the columns directly mentioned (e.g., updating the club_name from "Barcelona" to "Paris Saint-Germain"), but also infer and update causally related columns. For instance, it should update the coach_name from "Ronald Koeman" to "Mauricio Pochettino", reflecting the change in team affiliation. This example highlights the need for LLMs to capture complex causal relationships within the data schema to generate accurate and comprehensive updates.

Proposed Method. To enable LLMs to perform causally-driven cascading updates from natural language instructions, without sending table content data to models that compromise data confidentiality, in this section, we introduce ***Castle*** (see Figure 1), a multi-stage workflow addressing the aforementioned challenge via divide-and-conquer chain-of-thoughts (DC-CoT) with zero-shot samples. ***Castle*** orchestrates the generation and execution of SQL `UPDATE` queries from natural language instructions, maintaining data consistency via causal reasoning and robust query construction. The entire process is detailed in Algorithm 1 and described in what follows.

2.1 Skeleton: Identifying Columns for Update

As the first critical step to update the data in tables, ***Castle*** needs to accurately identify the columns to be updated from a natural language instruction. The expected result of this step is the skeleton

Algorithm 1 WORKFLOW OF C.A.S.T.L.E.

Require: Natural language update instruction x, table schema S

1: **C. Column Identification:**
2: Based on S, extract directly mentioned target column(s) C_{direct} and table from x
3: **A. Attribute Dependency Analysis:**
4: Use schema S and reasoning over C_{direct} to infer causally dependent columns C_{cascade}
5: **S. Subquery Planning:**
6: **for** each $c \in C_{\text{cascade}}$ **do**
7: Generate subquery q_c to retrieve correct value for c based on S
8: **end for**
9: **T. Trigger Maintenance:**
10: **for** each derived aggregate column $c \in C_{\text{cascade}} \cup C_{\text{direct}}$ **do**
11: Check trigger for maintaining derived c
12: **if** trigger is missing **then**
13: Generate SQL trigger t_c (via schema-based causal reasoning)
14: Deploy trigger t_c into the database to maintain real-time consistency
15: **end if**
16: **end for**
17: **L. Logical Query Composition:**
18: Combine C_{direct} values from x and valid subquery q_c into final SQL UPDATE query q
19: **E. Execution:**
20: Execute UPDATE query q on target data table.

of SQL UPDATE queries. ***Castle*** distinguishes between two types of columns from the database table schema:

Columns to be directly updated (C_{direct}). Using the given table schema and natural language instructions from general users, ***Castle*** identifies explicitly mentioned columns that need to be updated with data provided by the users. For example, like shown in Figure 1, given the instruction "Lionel Messi has transferred club from Barcelona (code: fc-barcelona) to Paris Saint-Germain (code: fc-paris-saint-germain)" the directly related columns `club_name` and `club_code` are explicitly identified by ***Castle***. This procedure corresponds to Line 1 in Algorithm 1.

Causally-dependent columns to be cascade updated (C_{cascade}). ***Castle*** applies the causal reasoning capabilities of LLMs via structured instruction to identify implicitly affected columns in each transaction (e.g., UPDATE) in the database. For instance, updating a player's club may also require updating dependent columns such as this player's `coach` or `competition`, as depicted in Figure 1. This procedure corresponds to Line 3 in Algorithm 1.

After the identification of the columns to be directly updated and/or cascade updated, a skeleton of the SQL UPDATE query is ready as shown in Code 1. Now the question remains to be "what to update".

```
UPDATE player_record
SET
    "club_name" = 'Paris Saint-Germain',
        -- directly update
    "club_code" = 'psg', -- directly
        update
    ...
    "stadium_name" = ?, -- causally-
        dependent column
    "competition_country" = ?, --
        causally-dependent column
    ...
    "foreigners_percentage" = ?, --
        aggregate and derived column
    "squad_size" = ? -- aggregate and
        derived column
    ...
WHERE
    "player_code" = 'lionel-messi';
```

Code 1: An example of generated SQL UPDATE skeleton for table `player_record`. Question marks here serve as placeholders for later subqueries or trigger maintenance.

2.2 Subquery Planning: Evidence-grounded Updates

After identifying the columns to update, ***Castle*** proceeds to handle causally-dependent updates securely through structured subquery planning. Instead of exposing actual table content data to LLMs, ***Castle*** only provides the table schema along with system instructions as input to LLMs integrated with the system. For each causally dependent column ($c \in C_{\text{cascade}}$) that needs to be updated, the LLM generates SELECT subqueries to fetch the correct value to update. Each subquery expects a result cardinality of at most one, guaranteeing legal and precise updates without data exposure, shown as an example in the dashed box in the middle of the right column of Figure 1.

Moreover, prior to query composition and execution, each generated subquery undergoes syntax checking to ensure its cardinality and compatibility with database-specific dialect rules, explicitly validating clauses like LIMIT (e.g., PostgreSQL

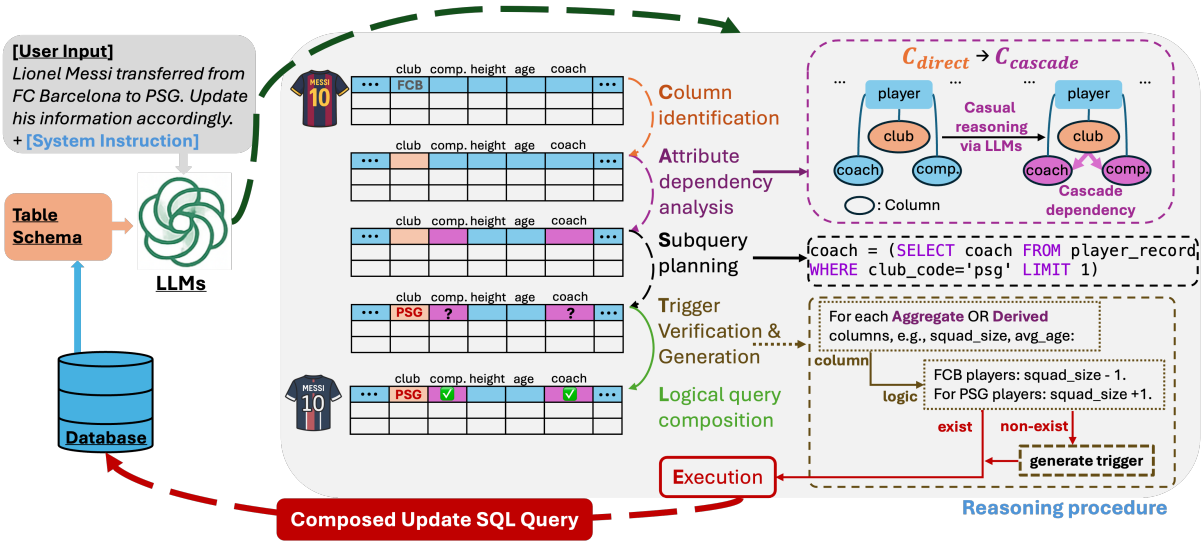

Figure 1: A sample workflow of ***Castle*** illustrates how a single transaction with natural language instructions as input is completed as a cascading update SQL transaction, incorporating the reasoning procedures of LLMs alongside the database.

and MySQL) or `TOP` (e.g., Microsoft SQL Server), thereby minimizing syntax-related runtime errors.

2.3 Trigger Verification and Generation

Aggregate and derived columns (e.g., counts, averages, or percentage metrics) or materialized tables in relational databases often reflect precomputed summarized information crucial for fast analytical retrieval (Jugel et al., 2016) in Business Intelligence (BI). When underlying data changes occur (e.g., player transfers or retail records change), traditional ETL-based methods may rely on scheduled batch data recomputing to maintain data consistency. ***Castle*** addresses this through automatic verification and generation of SQL `TRIGGER`, which efficiently maintains these derived metrics in real-time upon data modification events. This mechanism provides robustness, consistency, and efficiency for event-driven causal cascade updates.

Trigger Verification. Given columns identified either directly or via causal reasoning ($C_{\text{cascade}} \cup C_{\text{direct}}$), ***Castle*** first checks existing SQL Triggers from the database. Checking the existence and syntax of triggers based on the current schema from the system's maintained triggers table/view.

Trigger Generation. If the verification reveals missing triggers, ***Castle*** utilizes the schema-based causal reasoning capability of LLMs to dynamically generate efficient SQL trigger scripts with functions to maintain data consistency from events. The triggers are designed explicitly to accurately reflect updated derived (or aggregate) metrics like `squad_size`, `average_age`, and `foreigners_percentage`. Such one-time effort can automatically and consistently propagate changes without further manual intervention, thereby enhancing data integrity across the database.

In ***Castle***, generated triggers are subsequently deployed to the database and seamlessly integrated into transaction workflows. This proactive trigger management significantly reduces runtime computational overhead and LLM token usages, but also guarantees consistency for aggregate and derived data metrics in databases.

2.4 Compatibility with other systems

Castle is designed to be broadly compatible with industry standard DBMS such as PostgreSQL [1], MySQL [2], Microsoft SQL Server [3], and so on. On the one hand, it runs purely at the schema level, without requiring system-specific modules, extensions, and configurations (e.g., indices, paging, caching). Thus, it can be seamlessly integrated into existing data infrastructure without requiring modifications or additional extensions to the underlying database engine.

[1] https://www.postgresql.org/
[2] https://www.mysql.com/
[3] https://www.microsoft.com/en-us/sql-server

3 Experiments

We run our experiment on PostgreSQL 17 hosted on Neon [4], a serverless database platform built on AWS Aurora Postgres, with different LLMs integrated in the system workflow, including ChatGPT-4o [5], LLaMA-3.1-8B-Instruct [6] and Qwen2.5-7B-Instruct [7] as SQL generators.

3.1 Dataset

To evaluate our approach to causally driven cascade updates in structured relational databases, we utilize two real-world datasets:

Dataset	Size	#Columns	Data Type
Soccer Transfer	1M+	28	Date, Text, Numeric
UCI Retail	541K	12	Date, Text, Numeric

Table 2: Overview of relational datasets used in our experiment evaluation.

Soccer Transfer Dataset. A complete and comprehensive dataset recording over a million football player appearances and transfers worldwide yearly, including personal details, club and national team affiliations, transfer histories, and performance statistics. The relational structure allows users to model complex dependencies. Our ground truth of one year's update information is based on the following year's player record. We compare every pair of adjacent year records and find out those players who changed their club, such difference provides the club update information for us to extract the fact from the later year and evaluate LLM's ability to perform causal cascade update on the earlier year's record [8].

UCI Online Retail II Dataset. This dataset contains over half a million transactional records from a UK-based online retailer to worldwide customers, covering sales and returns over two years. Each record includes attributes like invoice number, product code, quantity, invoice date, unit price, customer ID, and country (Chen, 2012). We augment this dataset by: (1) Deriving the **Quarter** from the invoice date, corresponding to the quarter report in a materialized view (table) for faster data retrieval. (2) Mapping **Country** to **Region** for geographical analysis.

3.2 Evaluation Metrics

In retrieval-focused Text2SQL, one of the metrics is Execution Accuracy, which measures how close the results of the generated SQL query are to the ground truth results. In MultiSQL (Li et al., 2024), the evaluation metric for update operations in databases is state comparison, which directly compares the whole content of two database states (before and after update operation), returning a binary result of 0 (different) or 1 (same). Unlike the metric in MultiSQL, according to our workflow design, while performing database update operations, the first question is to identify the data to update.

Thus, for the update operations, given the causal and ripple-effect nature of cascade updates, we evaluate the model's reasoning capability of the causal cascade update scenario through **recall** (how many of the truly needed updates were found), with breakdown into those directly updated columns and cascade updated columns. This metric assesses the model's ability to holistically reason about multiple column dependencies within and outside the data table, quantifying how many of the truly needed updates were identified after the LLM reasoning procedure, and corresponds to evaluating the proposed "where to update" challenge in this work.

$$\text{Recall} = \frac{|\delta_{\text{Identified Cell to Update}}|}{|\Delta_{\text{Total Cell Requiring Updates}}|} \quad (1)$$

On the other hand, after we quantify the columns identified by LLMs to update, we also need to know the proportion of correctly updated causal columns among those targeted for update by the model. For example, after identifying columns to update, if one model fails to update all required columns, it has two kinds of errors: Type I (unnecessary update) errors and Type II (missed update) errors, either could happen when conducting updates to the database. The **F1-score** summarizes both aspects, indicating the model's overall effectiveness in both identifying and accurately updating causally dependent columns.

Besides, correctness is the final requirement in our task settings, where correct results would justify the usability of our proposed workflow. Thus, we introduce cell-wise correctness (CC), which evaluates fine-grained correct rate on how accurately the model updates each individual cell (i.e.,

[4] https://www.neon.tech/
[5] https://openai.com/index/hello-gpt-4o/
[6] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
[7] https://huggingface.co/Qwen/Qwen2.5-7B-Instruct
[8] https://www.kaggle.com/datasets/davidcariboo/player-scores

Method \ Model	ChatGPT-4o	LLaMA-3.1-8B-Instruct	Qwen2.5-7B-Instruct
***Castle*(w/o data)**	**99.96 | 89.39 | 80.82**	**99.49 | 88.62 | 79.45**	**96.32 | 87.58 | 77.45**
Multi-SQL (w/ data)	99.25 | 88.62 | 79.45	95.50 | 82.93 | 70.76	90.41 | 84.01 | 72.22
Baseline (w/ data)	99.18 | 88.35 | 79.00	99.32 | 88.19 | 78.76	96.10 | 83.41 | 71.98
Method \ Model	**ChatGPT-4o**	**LLaMA-3.1-8B-Instruct**	**Qwen2.5-7B-Instruct**
***Castle*(w/o data)**	**95.93 | 86.45 | 83.55**	**91.25 | 84.32 | 81.63**	**89.23 | 84.03 | 81.90**
Multi-SQL (w/ data)	93.68 | 85.83 | 81.68	79.87 | 71.83 | 73.39	81.34 | 73.47 | 75.90
Baseline (w/ data)	92.15 | 84.76 | 80.58	75.78 | 69.12 | 65.13	74.89 | 72.09 | 64.63

Table 3: Evaluation of update performance across models with methods on **Soccer Transfer** and **Retail** dataset, respectively. Each cell reports: Recall, F1-score, and cell-wise correct rate of **directly-updated columns.**

Method \ Model	ChatGPT-4o	LLaMA-3.1-8B-Instruct	Qwen2.5-7B-Instruct
***Castle*(w/o data)**	**99.95 | 85.25 | 85.21**	**80.84 | 81.24 | 80.43**	**75.52 | 72.01 | 67.93**
Multi-SQL (w/ data)	52.21 | 68.58 | 52.16	50.31 | 68.97 | 52.17	52.39 | 68.73 | 55.39
Baseline (w/ data)	52.16 | 68.53 | 52.09	50.09 | 67.55 | 52.00	51.12 | 65.80 | 52.06
Method \ Model	**ChatGPT-4o**	**LLaMA-3.1-8B-Instruct**	**Qwen2.5-7B-Instruct**
***Castle*(w/o data)**	**93.03 | 90.02 | 85.93**	**89.32 | 81.36 | 83.31**	**87.98 | 83.21 | 84.02**
Multi-SQL (w/ data)	89.10 | 74.12 | 69.58	83.80 | 58.80 | 56.09	85.91 | 65.01 | 61.23
Baseline (w/ data)	88.45 | 76.16 | 70.15	82.42 | 55.14 | 53.76	85.04 | 66.10 | 70.49

Table 4: Evaluation of update performance across models with methods on **Soccer Transfer** and **Retail** dataset, respectively. Each cell reports: Recall, F1-score, and cell-wise correct rate of **causal cascade updated columns** (without derived values).

each column within each row) across the entire database after applying one natural language update instruction. The Cell-wise correctness (CC) is defined as follows:

$$\text{CC} = \frac{|\delta_{\text{Correct Cell Updated}}|}{|\Delta_{\text{Total Cell Requiring Updates}}|} \quad (2)$$

Last but not least, in order to further study how causal dependent columns are correctly updated aside from direct updates, we further break these metrics down into two complementary components: columns explicitly mentioned in the natural language instruction (C_{direct}) and cascading columns inferred from causal or structural dependencies (C_{cascade}).

3.3 Results

While ***Castle*** does not provide a `SELECT`-like query result as output, we evaluated our update results by querying them and comparing them with the corresponding ground truth, as soon as the update operation occurred in the database. In Table 3 and Table 4, we present the evaluation results of update performance for directly-updated columns (C_{direct}) and causal cascade-updated columns (C_{cascade}, but without derived columns for `TRIGGER`s to maintain) across three representative LLMs and three SQL generation methods: ***Castle*** (ours, schema-only), MultiSQL (content-augmented), and baseline method (content-augmented).

In addition to direct update commands, we also evaluated the ability of LLMs to generate correct SQL `TRIGGER` statements within our workflow that enforce ripple-effect data consistency with our causal cascade `UPDATE` queries. The generated trigger is considered correct if, once after deployment, it consistently maintains the correctness of the summary table right after transactional updates, as compared with ground truth using cell-wise correctness. The result is shown in Table 5 alongside

Method / Model	ChatGPT-4o	LLaMA-3.1-8B-Instruct	Qwen2.5-7B-Instruct
Castle (Trigger only)	83.31	79.37	77.29
***Castle* w/ trigger**	83.84	79.91	73.53
Method / Model	**ChatGPT-4o**	**LLaMA-3.1-8B-Instruct**	**Qwen2.5-7B-Instruct**
Castle (Trigger only)	87.64	77.90	81.01
***Castle* w/ trigger**	86.81	79.51	81.78

Table 5: Evaluation of LLM-generated `TRIGGER` via cell-wise correctness, bottom row represents ***Castle***'s average cell-wise correctness over all data columns and tables with integrated trigger generation mechanism.

Castle having trigger generated in the system, both of their cell-wise correctness are the average rate of experiments conducted 100 times. Our experiments demonstrate that data consistency can be maintained automatically and robustly, even across complex, multi-row updates.

3.4 Discussion

In our evaluation, we measured "where to update" via the recall metric of cascade reasoning, and also "what to update" with F1-score and cell-wise correctness metrics.

The recall metric in our experiment directly evaluates the LLM's ability to correctly identify where to update in the given table schema(s), i.e., which columns (both direct and causal/cascade) should be updated based on natural language instructions. Our schema-only approach, ***Castle***, consistently achieves the highest recall across all models, particularly with GPT-4o, outperforming content-augmented baseline methods on both directly and causally updated columns. This demonstrates ***Castle***'s ability to reason over schema semantics and relationships without access to table content.

F1-score and cell-wise correctness indicate the model's proficiency in determining what values to update and producing the correct SQL subqueries filling the outer skeleton. We discover ***Castle*** performs consistently better, especially in scenarios with larger or more complex schemas, or having column interdependencies (as in the Retail dataset), where achieving high cell-wise correctness becomes more challenging.

Table 5 presents the evaluation of LLM-generated triggers for maintaining aggregate or materialized columns in real-time. The results show that triggers generated by ***Castle*** (both standalone and integrated) are comparable to LLM-generated complex SQL queries. However, triggers are instantly activated and only require one-time effort that can provide long-term, automatic consistency without requiring repetitive code generation or human intervention. This finding shows the potential of integrating LLM-based trigger generation into modern DBMS.

4 Related Work

4.1 Text2SQL

In general, Text2SQL (or NL2SQL) takes a given natural language text query as a task, generates the task-specific SQL queries (Mitsopoulou and Koutrika, 2025; Liu et al., 2024; Ma et al., 2025), and compares the query result table with the groundtruth provided by baselines such as Spider (Lei et al., 2024), BIRD (Li et al., 2023), and CoSQL (Yu et al., 2019). However, until recently, Text2SQL datasets contained almost exclusively `SELECT` queries, and update operations have been little investigated in Text2SQL research. The recent MultiSQL approach (Li et al., 2024) supports generating simple direct update commands with table content provided to LLMs for SQL generation (Shen and Kejriwal, 2024; He et al., 2025). Moreover, the authors of MultiSQL provided a benchmark dataset that includes `UPDATE` commands. However, those LLM-generated virtual data lack quality and real-world verifiability for causal relationships. In our work, two real verifiable datasets from different domains with different structures are used to verify our proposed method in causal cascade update.

In Text2SQL tasks, another common shortcoming for accurate query generation is the need for table context to understand natural language intents better (Sun et al., 2018). If table content from the actual query table is provided, it could significantly increase the accuracy of the generated query (Mitsopoulou and Koutrika, 2025). However, such an approach undermines data confidentiality. By con-

trast, our approach achieves schema-only reasoning without having to send table content to LLMs.

Recent approaches, such as CHASE-SQL (Pourreza et al., 2025), focus on improving SQL query generation via divide-and-conquer strategies and chain-of-thought (CoT) (Wei et al., 2022). Our approach also applies this reasoning strategy to address the challenges mentioned in the paper's introduction section.

4.2 LLM-Assisted Data Wrangling

Recent research has expanded the role of LLMs from purely generating SQL statements to performing broader data wrangling tasks. For instance, CodexDB (Trummer, 2022) leverages Codex models to automate database interactions, demonstrating LLM capabilities for diverse database operations. TableLLM (Zhang et al., 2025) is a dedicated model for document-level (lightweighted) spreadsheet manipulations, including insert, update, and delete operations. However, these operations require the whole table to be fed to the context window of LLMs. In addition, each operation is generated in isolation, neglecting cascades or multi-record dependencies.

Unlike these methods, ***Castle*** combines LLM-generated SQL code with a schema-driven reasoning process, systematically managing those causal cascade updates in denormalized schemas.

4.3 Ripple Effects in Knowledge Editing

Knowledge editing (Mitchell et al., 2021; Meng et al., 2022a,b) in LLMs aims to update specific factual information within a model without necessitating retraining. However, such interventions often lead to "ripple effects" (Cohen et al., 2024), where modifications to one fact inadvertently influence related or unrelated knowledge within the model. (Cohen et al., 2024) introduced the RippleEdits benchmark to assess these effects, revealing that current editing methods frequently fail to ensure consistent knowledge updates, thereby compromising the model's reliability. Further analysis in GradSim (Qin et al., 2024) identified gradient similarity (GradSim) as a key indicator of ripple effects, demonstrating a strong positive correlation between GradSim and the successful propagation of edits. To address these challenges, (Zhao et al., 2024) proposed RippleCOT, an in-context learning approach that integrates chain-of-thought reasoning to enhance the accurate dissemination of edits across related facts. Collectively, these works underscore the complexities inherent in knowledge editing for LLMs and highlight the necessity for advanced methods to manage unintended ripple effects.

While prior research has primarily examined ripple effects within LLMs (Cohen et al., 2024; Qin et al., 2024), our work shifts focus to the ripple effects occurring in external databases that serve as knowledge bases for LLMs. In retrieval-augmented generation (RAG) pipelines, effectively managing ripple effects during data retrieval by the DBMS can significantly enhance the accuracy and reliability of downstream LLM outputs (Shi et al., 2024; Zhao et al., 2024). Our approach offers a complementary perspective to existing model-level interventions, emphasizing the importance of database-level strategies in mitigating unintended ripple effects.

5 Conclusion

Castle addresses the challenge in causal-driven cascade updates with respect to both "where to update" and "what to update". It also demonstrates that general pre-trained LLMs can reason over schema structures to perform cascade-consistent SQL updates without requiring access to table contents, thus providing a broader, trustworthy, structured LLM reasoning for general data systems and code generation.

Limitations

Like most practical analytical queries and business intelligence (BI) workloads, we also assume the scenarios where data and its derived values are stored in a single unified database for optimized query performance. We thus do not consider federated or multi-database environments. Additionally, we do not consider multi-hop post-cascade updates due to the absence of real-world, verifiable datasets that reliably capture such propagation chains. Lastly, our work considers natural language instructions for the database to be explicit, which are generated via a unified script as part of input to LLMs; our study does not consider instructions in different languages aside from English.

References

Andrey Balmin and Yannis Papakonstantinou. 2005. Storing and querying xml data using denormalized relational databases. *The VLDB Journal*, 14(1):30–49.

Daqing Chen. 2012. Online Retail II. UCI Machine Learning Repository.

Roi Cohen, Eden Biran, Ori Yoran, Amir Globerson, and Mor Geva. 2024. Evaluating the ripple effects of knowledge editing in language models. *Transactions of the Association for Computational Linguistics*, 12:283–298.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205*.

Mingqian He, Yongliang Shen, Wenqi Zhang, Qiuying Peng, Jun Wang, and Weiming Lu. 2025. Star-sql: Self-taught reasoner for text-to-sql. *arXiv preprint arXiv:2502.13550*.

Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. 2016. Vdda: automatic visualization-driven data aggregation in relational databases. *The VLDB Journal*, 25(1):53–77.

Ralph Kimball and Margy Ross. 2013. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

Chunhui Li, Yifan Wang, Zhen Wu, Zhen Yu, Fei Zhao, Shujian Huang, and Xinyu Dai. 2024. MultiSQL: A schema-integrated context-dependent Text2SQL dataset with diverse SQL operations. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13857–13867, Bangkok, Thailand. Association for Computational Linguistics.

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Preprint*, arXiv:2305.03111.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024. A survey of nl2sql with large language models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109*.

Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2025. Sql-r1: Training natural language to sql reasoning model by reinforcement learning. *arXiv preprint arXiv:2504.08600*.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022a. Locating and editing factual associations in gpt. *Advances in neural information processing systems*, 35:17359–17372.

Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. 2022b. Mass-editing memory in a transformer. *arXiv preprint arXiv:2210.07229*.

Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D Manning. 2021. Fast model editing at scale. *arXiv preprint arXiv:2110.11309*.

Anna Mitsopoulou and Georgia Koutrika. 2025. Analysis of text-to-sql benchmarks: Limitations, challenges and opportunities. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, pages 199–212. OpenProceedings.org.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2025. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *The Thirteenth International Conference on Learning Representations*.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems*, volume 36, pages 36339–36348. Curran Associates, Inc.

Jiaxin Qin, Zixuan Zhang, Chi Han, Manling Li, Pengfei Yu, and Heng Ji. 2024. Why does new knowledge create messy ripple effects in llms? *arXiv preprint arXiv:2407.12828*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.

Ke Shen and Mayank Kejriwal. 2024. Select-sql: Self-correcting ensemble chain-of-thought for text-to-sql. *arXiv preprint arXiv:2409.10007*.

Yucheng Shi, Qiaoyu Tan, Xuansheng Wu, Shaochen Zhong, Kaixiong Zhou, and Ninghao Liu. 2024. Retrieval-enhanced knowledge editing for multi-hop question answering in language models. *arXiv e-prints*, pages arXiv–2403.

Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic parsing with syntax- and table-aware SQL generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 361–372, Melbourne, Australia. Association for Computational Linguistics.

Immanuel Trummer. 2022. Codexdb: synthesizing code for query processing from natural language instructions using gpt-3 codex. *Proc. VLDB Endow.*, 15(11):2921–2928.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. 2025. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, and 5 others. 2019. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Xiaokang Zhang, Sijia Luo, Bohan Zhang, Zeyao Ma, Jing Zhang, Yang Li, Guanlin Li, Zijun Yao, Kangli Xu, Jinchang Zhou, Daniel Zhang-Li, Jifan Yu, Shu Zhao, Juanzi Li, and Jie Tang. 2025. Tablellm: Enabling tabular data manipulation by llms in real office usage scenarios. *Preprint*, arXiv:2403.19318.

Zihao Zhao, Yuchen Yang, Yijiang Li, and Yinzhi Cao. 2024. Ripplecot: Amplifying ripple effect of knowledge editing in language models via chain-of-thought in-context learning. *arXiv preprint arXiv:2410.03122*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

A Table Schema

Soccer Transfer Database Schema

```
CREATE TABLE IF NOT EXISTS {
    table_name} (
    player_id
        SERIAL PRIMARY KEY,
    player_code
        VARCHAR(100),
    first_name
        VARCHAR(100),
    last_name
        VARCHAR(100),
    full_name
        VARCHAR(255),
    date_of_birth
        VARCHAR(100),
    age
        NUMERIC,
    height
        DECIMAL(4,2),
    citizenship
        VARCHAR(100),
    position
        VARCHAR(100),
    foot
        VARCHAR(20),
    joined_date
        VARCHAR(100),
    contract_expires
        VARCHAR(100),
    social_media
        JSONB,
    birthplace_city
        VARCHAR(100),
    birthplace_country
        VARCHAR(100),
    club_code
        VARCHAR(100),
    club_name
        VARCHAR(255),
    squad_size
        NUMERIC,
    average_age
        DECIMAL(4,2),
    foreigners_number
        NUMERIC,
    foreigners_percentage
        DECIMAL(5,2),
    national_team_players
        NUMERIC,
    stadium_name
        VARCHAR(255),
    stadium_seats
        VARCHAR(50),
    net_transfer_record
        VARCHAR(50),
    coach_name
        VARCHAR(255),
    competition_code
        VARCHAR(50),
    competition_type
        VARCHAR(50),
    competition_country
        VARCHAR(100),
    competition_seasoned_href TEXT
    );
```

UCI Retail Database Schema

```
CREATE TABLE IF NOT EXISTS {
    table_name} (
        stockcode TEXT,
        description TEXT,
        quantity INTEGER,
        country TEXT,
        region TEXT,
        y2010q4_quantity INTEGER,
        y2011q1_quantity INTEGER,
        y2011q2_quantity INTEGER,
        y2011q3_quantity INTEGER,
        y2011q4_quantity INTEGER,
        PRIMARY KEY(stockcode,
            country)
    );
```

B Trigger

```
SELECT event_object_table AS table_name,
     trigger_name
FROM information_schema.triggers
GROUP BY table_name, trigger_name
ORDER BY table_name, trigger_name;
```

Code 2: An example query of listing Triggers names and corresponding tables in a PostgreSQL database.

```
SELECT tgname
FROM pg_trigger
WHERE tgrelid = 'player_record'::
    regclass;
```

Code 3: An example query of checking Trigger on table `player_record`

```
CREATE OR REPLACE FUNCTION
    update_squad_size_transfer()
RETURNS TRIGGER AS $$
BEGIN
    -- Decrement squad size from old
        club
    IF OLD.club_code IS NOT NULL THEN
        UPDATE player_record
        SET squad_size = squad_size - 1
        WHERE club_code = OLD.club_code;
    END IF;

    -- Increment squad size for new club
    IF NEW.club_code IS NOT NULL THEN
        UPDATE player_record
        SET squad_size = squad_size + 1
        WHERE club_code = NEW.club_code;
    END IF;

    RETURN NEW;
END;
```

Code 4: An example of trigger function on table `player_record`

C LLM Prompts Examples

Castle Prompt

```
Database Schema:
{schema}

Instruction:
{instruction}

Generate an UPDATE SQL statement
    to update the player's
    club_name and club_code columns
    , and do consider the ripple
    effects via this update, since
    this update may cause other
    columns update. Other columns,
    if required, should be handled
    via subqueries, you will never
    know the content of the data
    table except table schema.

Think about this step by step, and
     you need just one SQL UPDATE
    query (could be with subqueries
    ) as output.
1. First, what are the columns
    needed to be updated for this
    table schema? Come up with a
    UPDATE skeleton with columns
    need to update, no LIMIT is
    needed in outer skeleton.
2. Second, query each column data
    needed to be used for each
    columns update, remember to use
     the LIMIT clause in SUBQUERY
    since the subquery is used to
    fill the outer skeleton.
3. Combine the previous two
    queries.

Carefully follow these rules for
    SQL formatting:
- Use double quotes for all SQL
    identifiers (table names,
    column names).
- Use single quotes around all
    literal string values (such as
    player codes or club names).

Think step-by-step and return
    exactly one SQL UPDATE query as
     output. Please only return the
     SQL statement in a code block
    and do not generate anything
    else.
```

Soccer Transfer Instruction

```
Player '{first_name} {last_name}'
    (code: {player_code}) changed
    club from {from_club_code} to {
    dest_club_code}. Update his/her
     information.
```

Baseline Prompt

```
Database Schema:
{schema}

Instruction:
{instruction}

You need just one SQL UPDATE query
     as output. Please only return
    the SQL statement in a code
    block and do not generate
    anything else.
```

Multi-SQL Prompt

```
Database Schema:
{schema}

Instruction:
{instruction}

Sample table data:
{table_sample_content}


Generate an UPDATE SQL statement
    to update the player's
    club_name and club_code columns
    ,
# Chain of Thought:
1. Identify which table and
    columns to update.
2. Determine the WHERE clause to
    target the correct rows.
3. Compute any needed values via
    subqueries if necessary (e.g.,
    aggregations).
4. Assemble into one valid UPDATE
    statement.

Think step-by-step and return
    exactly one SQL UPDATE query as
     output. Please only return the
     SQL statement in a code block
    and do not generate anything
    else.
```

UCI Retail Instruction

```
Now a return / refund case
    happened, stockcode is '{
    StockCode}', quantity in this
    case is {Quantity}, and its
    invoicedate is at {InvoiceDate
    }, unitprice: {UnitPrice}, and
    it happend in {Country}. Update
     this transaction in the
    summary table.
```

D LLM Generated SQL Examples

ChatGPT-4o with Castle on UCI Retail

```
UPDATE
    online_retail_quarterly_summary
SET
    "quantity" = "quantity" + (
        SELECT SUM("quantity")
        FROM (
            SELECT -1 AS "quantity
                "
            WHERE "stockcode" = '
                84978' AND "country
                " = 'United Kingdom
                ' AND "invoicedate"
                 BETWEEN '
                2011-10-01' AND '
                2011-12-31'
            LIMIT 1
        ) AS subquery_quantity
    ),
    "y2011q4_quantity" = "
        y2011q4_quantity" + (
        SELECT SUM("quantity")
        FROM (
            SELECT -1 AS "quantity
                "
            WHERE "stockcode" = '
                84978' AND "country
                " = 'United Kingdom
                ' AND "invoicedate"
                 BETWEEN '
                2011-10-01' AND '
                2011-12-31'
            LIMIT 1
        ) AS subquery_y2011q4
    )
WHERE "stockcode" = '84978' AND "
    country" = 'United Kingdom';
```

ChatGPT-4o with Multi-SQL on UCI Retail

```
UPDATE
    online_retail_quarterly_summary
SET
    quantity = quantity - 1,
    y2011q4_quantity =
        y2011q4_quantity - 1
WHERE
    stockcode = '84978' AND
    country = 'United Kingdom';
```

ChatGPT-4o with Baseline on UCI Retail

```
UPDATE
    online_retail_quarterly_summary
SET y2011q4_quantity =
    y2011q4_quantity - 1
WHERE stockcode = '84978' AND
    country = 'United Kingdom';
```