Exploring Retrieval-Augmented Generation for Tabular Data

Group Members:
Michael Yoshida,
Nick Howard,
Thomas Llamzon,
Andres Pedreros,
Kalundi Serumaga,
Databases II (Advanced Databases)

Project Report

December 9, 2025

Contents

1 Abstract

Large language models (LLMs) commonly struggle with tabular reasoning due to context and token limits, motivating retrieval centered approaches such as TableRAG. In this project, we re-implemented the TableRAG pipeline from input, schema and cell indexing, to retrieval, prompt assembly and PyReAct-based reasoning. This was accomplished using only method descriptions from the original TableRAG paper. Three benchmarks were reconstructed (TabFact, BIRD and Arcade) into the TableRAG JSONL format and the pipeline was evaluated across small, medium, and million-cell large tables. Our reproduction of TabFact achieved 85.1% accuracy, closely matching published results, while tests on large tables showed that retrieval-based expansion improves performance when tables exceed the model's token budget. Using GPT-4o-mini, expansion increased accuracy by 6.9% on BIRD and 3.9% on Arcade, supporting the paper's claim that TableRAG's advantages are most prominent under severe truncation constraints. Across datasets, however, we also observed high sensitivity to preprocessing decisions, table coverage, and model token limits, all factors not emphasized in the original TableRAG article. Beyond replication, we extended TableRAG by conducting the first systematic analysis of the cell-encoding budget (B), a key parameter controlling how many distinct cell values are embedded during retrieval. Across ten budget conditions ranging from 0 to 100,000 cells, our results reveal a counter-intuitive U-shaped performance curve: extremely small budgets (5 cells) and very large budgets (10K-20K) outperform mid-range 10-500 cell values. We further identify a previously unreported principle that performance is not driven by the absolute budget, but by the ratio of encoded cells to table size, with accuracy peaking when 10-20% of a table's cells are indexed. This key finding refines the fixed-budget recommendation in the original TableRAG paper and provides a practical rule for adaptive budget selection. Finally, we show that budgets above 50,000 cells produce no measurable gains even for million-cell tables, indicating saturation and guiding cost-efficient deployment. Together, these findings clarify when TableRAG succeeds, expose its limitations, and contribute new insights into how retrieval granularity shapes performance in large-scale table reasoning.

2 Introduction

The rise of large language models (LLMs) has sparked interest in their potential applications in analyzing and processing large-scale data. On a small scale, these tasks are trivial for most commercial LLMs, but a fundamental limitation is their context window: the limit of tokenized information that can be input into an LLM.

For example, at the time of the original TableRAG publication, medium sized tables with 100 columns and 200 rows required approximately 40,000 tokens, which exceeded the upper limits of many commonly used LLMs. Previous efforts have looked at reducing the token load by summarizing tables into just their schema or indexing methods, but these still have their own information retention and performance concerns.

Si-An Chen et al.'s TableRAG framework addresses these concerns by implementing schema retrieval using column names combined with cell-value retrieval of distinct or frequent categorical values. This robust retrieval process is combined with query expansion and results in table representations that fit within LM token context limits [1].

In addition to the TableRAG and its retrieval-focused framework, the same authors developed a broader system-level approach called TAG (Table-Augmented Generation). TAG's three-part workflow includes query synthesis, query execution within the database and answer generation. Existing methods such as Text2SQL or table-based RAG achieve only parts of the larger workflow TAG was created to orchestrate. TAG was created with realistic queries in mind, merging database computation with LM and real world knowledge and reasoning[2].

Together, TableRAG supplies scalable access to large tables while TAG streamlines and unifies retrieval integration, database execution and generative reasoning. Both programs underscore the importance of hybrid systems that incorporate LMs with structured retrieval and database operations[1][2].

3 Background

3.1 Literature Review

This research endeavour is made in reference to two articles: TableRAG: Million-Token Table Understanding with Language Models, and Text2SQL is Not Enough: Unifying AI and Databases with TAG. It's worth addressing the important distinction between TableRAG and TAG. These are not the same thing. Where TableRAG focuses on minimizing and optimizing what is fed into the model, i.e. fitting the model's context window, TAG focuses on dividing labor between databases and language models across multiple stages, using the database to execute queries efficiently and the language model to add semantic understanding and generate natural answers. TableRAG focuses on smart retrieval for extremely large datasets, TAG aims to implement a more modular and efficient model architecture.

3.2 Benchmarking in Million-Token Table Understanding with Language Models [1]

This reference article highlights how prior "program-aided LMs" process entire tables, which often adds unnecessary overhead. Critical information to questions like "What is the average price of wallets?" lies in the column names (i.e. wallet price column), data types, and cell values that directly relate to the question. The article underscores how giving a language model entire tables that could consist of millions of tokens (smallest unit of text an LLM can process) hits context length limits, increasing latency and cost.

The authors' approach to TableRAG integrates schema retrieval and cell retrieval to extract essential information from tables, assisting an LM in solving queries on the provided information. The goal of their research is to design a scalable prompting / retrieval network for LMs that works on million token tables, keeps token usage small and independent of table size, and still gives enough information for correct reasoning. The token complexity of prior table-prompting paradigms can be benchmarked using three baselines:

- **Read Table**: LM sees the entire table – $O(NM)$; which is impossible for huge tables.
- **Read Schema**: LM only sees column names and types – $O(M)$; loses cell-level information.
- **Row-Column Retrieval**: Encode every row and column, retrieve top-K rows & columns, and form a $K \times K$ sub-table – $O(NM)$; semantics get fuzzy as rows/columns scale.

In large-scale table understanding, a table T is represented as

$$T = \big(v_{ij}\big), \quad 1 \le i \le N,\ 1 \le j \le M.$$

where N is the number of rows and M is the number of columns, and v_{ij} is the cell value at row i and column j. A table prompting function P yields an answer A given by an LM L to a natural language question Q. The table prompting function can be represented as

$$A = L\big(P(T)\big).$$

The authors' objective is to develop an efficient P that significantly reduces the size of the prompt, |P(T)|, compared to the original table |T|, so that an LM can effectively use large tables. At this point, it's worth elaborating on the stages of the TableRAG pipeline to provide necessary background on the approach for the later sections of this paper:

Table Ingestion

The system first ingests a large table that could have tens of millions of cells and converts it to two representations: one centered on schemas and one centered on cell values. Ingestion only happens *once* offline, allowing the system to scale beyond any model's context window.

Schema Database Construction

Each column of the table is converted into a structured schema record containing the **column name**, **data type**, and a small sample of **representative example values**. These metadata elements are embedded into vectors and stored in a **schema database**. This index supports fast similarity search so the model can retrieve only the small subset of columns relevant to a user query.

Cell Database Construction

The system extracts distinct (column, cell value) pairs and stores them in a **cell database**. Each distinct value is normalized, embedded, and indexed. Because large tables may contain millions of values, TableRAG enforces a **cell encoding budget (B)** (maximum number of cell-value entries that can be encoded), and **frequency-based truncation** (rare values may be dropped if a column contains too many distinct entries). This ensures retrieval remains computationally feasible even when tables approach the million-token scale.

Tabular Query Expansion

The system generates multiple rewritten versions of the query given a natural language question. These expansions capture synonyms, paraphrases, variations in wording, and rare but meaningful formulations. Each expanded query is then embedded separately. This step increases the chance of matching the correct schema elements and cell values in later retrieval steps by creating more opportunity for matching.

Schema Retrieval

Each expanded query is used to search the **schema index**. The system retrieves the **top–K most relevant columns**, based on semantic similarity. This step filters the table down to a small, targeted set of columns that are likely needed for reasoning, dramatically reducing prompt size.

Cell Retrieval

After schema retrieval narrows the scope, the system then searches the **cell-value index** to extract the most relevant cell entries from the retrieved columns. This yields a small list of concrete values (e.g., names, numbers, identifiers) that give the LM enough of a contextual basis to answer the question without ingesting the entire table.

Prompt Assembly

The LM prompt is constructed from the **original question**, the **retrieved schema elements**, the **retrieved cell values**, and a program-aided reasoning template. The prompt is kept extremely compact because it contains only the relevant slices of the table, fitting comfortably within the LM's context window.

Program-Aided Solving using PyReAct

TableRAG uses a **PyReAct agent**, an extension of ReAct (Reason + Act) that allows the LM to reason step-by-step, reference retrieved table elements, generate Python-like tool calls for filtering, aggregating, or comparing values, and iteratively refine intermediate results. This keeps logical operations structured and prevents hallucination in multi-step reasoning.

Code Execution

Whenever the LM agent makes a tool call (e.g., filtering rows, computing aggregates), the system executes that code in a controlled environment. This execution uses real schema information, real cell values, and any intermediate data computed earlier. TableRAG ensures deterministic and accurate table operations by delegating arithmetic, sorting, and filtering to code rather than the LM.

Answer Extraction

Finally, the agent uses the intermediate results to generate the final natural-language answer. The LM produces a concise response based on: the executed code outputs, the retrieved schema/cell information, and its reasoning chain. This hybrid process of *retrieval* + *structured reasoning* + *code execution* allows TableRAG to achieve high accuracy even when the underlying tables exceed the LM's context capacity by several orders of magnitude.

To assess the reasoning capabilities of TableRAG over extremely large tables, the authors developed two extensive QA (Question-Answer) Datasets, ArcadeQA and BirdQA, derived from the Arcade and Bird-SQL datasets (found in the reference papers [1, 2]). They also used a TabFact dataset, including synthetic tables running from 100 100 to 1,000 1,000 to examine the impact of different table sizes under the same question and key information. Their experiments use **GPT-3.5-Turbo, Gemini-1.0-Pro**, and **Mistral-Nemo-Instruct-2407** as LM solvers, and consider four baselines for table access strategies to compare to TableRAG:

1. **ReadTable**: Embed the entire table in the prompt, then ask the LM to solve the problem with the information.

2. **ReadSchema**: Assume table content is not directly accessible; incorporate data types and column names into the prompt.

3. **RandRowSampling**: Randomly select rows from the table with equal probabilities. Used to underscore the value of targeted retrieval methods.

4. **RowColRetrieval**: Encode rows and columns, then retrieve the top K rows and columns based on their similarity to the question's embedding to form a sub-table. Truncate tables to $\frac{B}{2M}$ rows, limiting the number of tokens encoded to B (cell encoding budget).

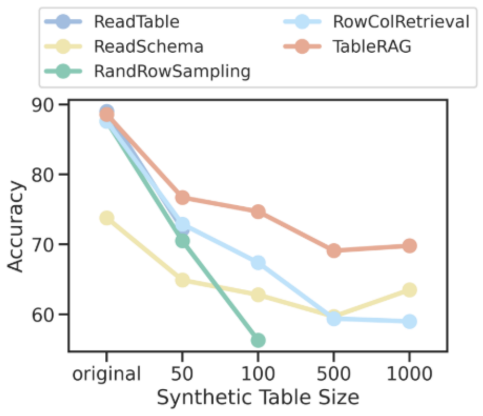

Figure 1: Performance evaluation of synthetic TabFact tables. TableRAG shows superior accuracy, and it decreases gracefully compared to the baselines.

On ArcadeQA and BirdQA, TableRAG outperforms all baselines for comparison across all three LMs used. On the synthetic TabFact tables, TableRAG's accuracy drops gracefully as table size increases, maintaining acceptable accuracy at 1 million cells, whereas ReadTable fails at this magnitude and only works on small tables. TableRAG also has the highest column and cell retrieval F1 score (precision & recall).

TableRAG Research Significance

Overall, the effectiveness and value of TableRAG comes from how it solves the retrieval problem to provide the most efficient context for a given question.

This paper informs our research by demonstrating that scalable table reasoning with LMs requires decoupling table size and prompt size. This is the fundamental distinction that TableRAG makes to improve large-scale table usability and understanding with schema-aware and value-aware retrieval instead of full table ingestion. We build on this research by re-implementing and running the TableRAG pipeline in code, testing it over several datasets and exploring when its retrieval approach actually helps. We use TableRAG as a starting point to understand how large tables can be compressed into useful context and where its methods work well.

3.3 *Text2SQL is Not Enough: Unifying AI and Databases with TAG*

This reference article researches TAG, Table-Augmented Generation. TableRAG focuses on gathering the most relevant and optimal context for a query to improve retrieval effectiveness over large datasets, whereas TAG focuses on improving the semantic depth that an LM can support in its answer to a natural language question, beyond the capabilities of pure Text2SQL. TableRAG optimizes context retrieval, TAG improves answer quality. Real queries over data often require more than relational algebra to serve natural language questions. They require sophisticated combinations of domain knowledge, world knowledge, exact computation, and semantic reasoning to serve better insights over data. The article also highlights how pure Text2SQL breaks down when a natural language query demands information not explicitly encoded in the tables. The goal of TAG is to unify database engines and language models (LMs) into a general system that can handle both structured queries and LM-level reasoning. TAG is broken down into three key steps performed iteratively:

(1) **Query Synthesis:** $\mathrm{syn}(R) \to Q$

The query synthesis step, syn, translates a user's natural language request R into an executable database query Q. Unlike pure Text2SQL, syn is not restricted to SQL grammar. Instead, it may produce a query that combines SQL operations with semantic predicates, LM-based filters, or user-defined functions that aim to capture nuances in text-heavy or knowledge-dependent fields. This step determines *what* data needs to be retrieved and *how* the system should process it. It interprets the user's request, chooses the appropriate relational and semantic operators, and formulates a query plan the database engine can use in *query execution*. This step bridges natural language requests/questions and database execution.

(2) **Query Execution:** $\mathrm{exec}(Q) \to T$

The query execution step exec executes Q on the database system to compute and gather the relevant data T. The system may use a traditional SQL engine or an AI-augmented engine supporting LM user-defined functions and semantic ranking operators, enabling the model to perform tasks that SQL cannot express (e.g. classification, similarity filtering, or contextual ranking). An important characteristic of this step is that execution performs exact computation and structured filtering on the database side, ensuring correctness and efficiency before the LM generates the final answer.

(3) **Answer Generation:** $\mathrm{gen}(R,T) \to A$

The answer generation step gen uses R and T to generate the final natural language answer A. This step may involve summarizing the table, comparing rows, performing semantic reasoning, or interpreting the results in light of external knowledge. TAG allows the LM to reason at a higher level than SQL, enabling more comprehensive interpretations while grounding the answer in the database output.

The TAG paper presents a broad design space that characterizes how LMs and databases can be combined to solve complex data queries. TAG supports diverse query types like point lookups, comparisons, classification tasks, ranking, and multi-step aggregations. This more accurately reflects the range of operations users naturally ask for. It also accommodates multiple structures of data like structured relational tables, semi-structured text fields, and vector representations. Execution engines may be classical (SQL) or AI-augmented, enabling LM-driven filtering or ranking directly inside the database workflow. Answer generation can also occur in a single pass or iteratively, allowing the LM to process large result sets in smaller chunks. Together, this design space shows that TAG is not just a method but a general framework for building hybrid data–AI systems.

To evaluate TAG, the authors construct a benchmark by modifying questions from the BIRD Text2SQL dataset. They rewrite queries to require semantic reasoning or external knowledge that SQL alone cannot encode. For example, it can categorize regions like "Silicon Valley", identifying "technical" versus "non-technical" posts, or ranking entries based on subjective attributes. Each question is paired with a manually annotated ground truth answer. The benchmark includes both match-based queries and more complex comparison, ranking, and aggregation tasks, each with variants that require either world knowledge or text reasoning.

This benchmark exposes the limitations of Text2SQL and motivates the need for TAG.

The paper evaluates several baselines to contextualize TAG's performance. These include classic Text2SQL, which attempts to generate SQL that directly yields the final answer; RAG over tables, which retrieves rows and asks an LM to answer from them; retrieval plus LM ranking pipelines; and Text2SQL combined with LM-based answer generation. The strongest baseline is a two-step Text2SQL+LM approach that uses SQL to collect candidate rows and an LM to interpret them. Finally, the authors manually craft TAG pipelines for each query category, serving as an upper bound on what TAG can achieve when syn–exec–gen is carefully coordinated.

The results show that traditional approaches struggle on the new benchmark. Text2SQL fails when questions require reasoning not represented in the schema, and RAG-based approaches perform extremely poorly (often near 0%) because they rely too heavily on retrieval and cannot execute structured computation. Even the strongest non-TAG baseline achieves low accuracy across categories. In contrast, hand-written TAG pipelines achieve roughly 55% accuracy. It improves performance by 20–65 percentage points over all baselines. TAG also reduces runtime by shifting computation to the database, demonstrating that hybrid execution yields both accuracy and efficiency benefits.

TAG Research Significance

The TAG paper provides an essential foundation for understanding how natural language interfaces can be extended beyond pure SQL queries. TAG shows how structured computation and LM reasoning can be combined into a unified pipeline, which helps us contextualize our own work with TableRAG and large-scale retrieval. For our research, TAG helps us understand the broader landscape of hybrid LM-database systems and highlights the kinds of reasoning and execution patterns needed to handle complex, real-world queries. It serves as the conceptual counterpart to TableRAG. It informs us of how retrieval, execution, and generation can be coordinated at system-level scale.

4 ARCHITECTURE ANALYSIS

4.1 System Architecture

4.1.1 Overview

The TableRAG system adopts a modular architecture designed to support scalable table oriented RAG systems, flexible LLM integration, and reproducible experimental pipelines. Each component is deliberately decoupled so that retrieval strategies, model providers, prompting styles, and agent behaviours can be swapped or extended without reconfiguring the broader system. At a high level, the architecture consists of four synergistic layers: agent, retrieval, model, and prompts. These layers are supported by a suite of utilities and scripts for database construction, test execution, and performance evaluation.

4.1.2 Agent Layer

The agent layer implements the core logic for interacting with tables and coordinating reasoning. It includes TableAgent, a class which manages prompt assembly, iterative reasoning, tool invocation, and integration with the retrieval and model layers. The agent orchestrates the entire workflow, from receiving a user query to producing a final answer. It uses a ReAct-style reasoning loop that alternates between natural-language thought steps and executable actions.

4.1.3 Retrieval Layer

The retrieval layer provides schema, cell, row, and column-level retrieval mechanisms. It supports multiple retrieval strategies, including BM25 for lexical matching, embedding-based dense retrieval using vector representations, and hybrid retrieval that combines both signals. This layer is designed so that the agent can flexibly select the most appropriate retrieval mode depending on the table size, heterogeneity, and the query at hand. The modular design ensures that relevant table slices can be isolated efficiently before reasoning.

4.1.4 Model Layer

The model layer abstracts interactions with LLM providers such as OpenAI, Google Vertex AI/Gemini, and others. It unifies prompt submission, response parsing, and token accounting behind a consistent interface, enabling experiments to swap models with minimal configuration changes. By decoupling model interaction from the reasoning code, the system can incorporate new model APIs or versions without modifying the retrieval or agent layers.

4.1.5 Prompt Layer

The prompt layer contains task-specific prompt templates used during reasoning. These templates include instructions for schema-aware reasoning, program-aided tool use, and task-specific RAG prompting. The separation of prompt logic from the agent ensures reproducibility and allows systematic variation of prompting strategies during evaluation.

4.1.6 Utility Layer

The utility layer implements helper functions for table loading, type inference, safe code execution, configuration parsing, and logging. These utilities support the core pipeline but remain independent from the reasoning and retrieval logic to enhance modularity.

4.2 Design Principles

4.2.1 Modularity

Each major function: retrieval, prompting, model interaction, and agent logic is encapsulated in its own module or class. Each module has a single responsibility, examples are: agents handle reasoning and execution, utilities handle data transformations, and prompts handle template management. This separation enables independent development, testing, and maintenance.

4.2.2 Abstraction

Interfaces for LLMs and retrievers abstract away provider-specific details like authentication, rate limiting, and response formatting, allowing different agent types to share the same execution pattern while varying their retrieval and reasoning strategies.

4.2.3 Extensibility

The system is designed for extensibility through several modular patterns and abstractions. Multiple agent types (e.g., PyReAct, RandSampling, TableSampling) implement distinct reasoning and table-processing algorithms, allowing easy substitution or expansion. Model and Retriever classes encapsulate LLM and retrieval logic, enabling new models or retrieval methods to be added without altering the core pipeline. Prompt templates are maintained independently from the agent logic, supporting alternative prompting strategies.

4.3 Design Patterns and Algorithms

4.3.1 ReAct Reasoning Loop

The agent uses a loop that alternates between LLM-generated "Thoughts" and "Actions", which are executed on the table data. Observations from code execution are fed back into the prompt for the next iteration. This enables complex, multi-step reasoning and dynamic table manipulation.

4.3.2 Prompt Construction

Prompts are dynamically built using markdown representations of tables and task-specific templates. The agent supports multiple prompt types and adapts the table context based on agent type (full table, sampled rows, schema-only, etc.).

4.3.3 Retrieval Strategies

The agent supports several retrieval algorithms including embedding based, BM25, and hybrid methods, and it can sample or filter table data before reasoning. The retriever is modular, allowing easy extension or replacement.

4.3.4 Token Management

The agent tracks input and output token counts for each query, enforcing context limits and logging usage for analysis.

4.3.5 Logging and Reproducibility

Results, including reasoning traces and answers, are saved to disk in both JSON and text formats. The agent can reload existing logs to avoid redundant computation.

4.4 Data Flow Mechanisms

4.4.1 Data Ingestion and Initialization

The pipeline initiates with the load_dataset module, which ingests task-specific datasets in JSON Lines format. To ensure consistency across the data pipeline, the system assigns unique identifiers to each entry and derives a standardized table_id from the associated table captions. Following data loading, the TableAgent class initializes the runtime environment by configuring essential parameters, including the retrieval strategy, maximum recursion depth, and the specific LLM backend.

4.4.2 Reasoning Agent Paths

The framework supports two distinct dataflow pathways depending on the size of the table and the specific agent configuration: the Direct Reasoning Agent (PyReAct) and the Retrieval-Augmented Agent (TableRAG).

4.4.2.1 Direct Reasoning Agent

For datasets that fit within the underlying model's context window, the system utilizes the PyReAct strategy to ingest the full table context directly. This process begins by converting the raw table text into a Pandas DataFrame. The system dynamically calculates the token count of the table. If this count exceeds the predefined limit, the agent triggers a row-based sampling mechanism to reduce input size, or raises an error if reduction is not feasible. Once processed, the table is serialized into a Markdown format and injected directly into the prompt template alongside the user's query to facilitate immediate reasoning.

4.4.2.2 Retrieval-Augmented Agent

The TableRAGAgent subclass addresses large-scale tables through a specialized "hallucination-and-retrieval" mechanism designed to filter data prior to reasoning. This process initiates with a schema retrieval phase, where the agent queries the LLM to generate potential column names relevant to the user's statement; these suggestions are subsequently used to query the retriever for the actual database schema. Concurrently, the system executes a cell retrieval phase by prompting the LLM to extract categorical keywords from the query, which serve as search terms to locate specific rows and filter out numerical noise. The final prompt is synthesized using only the retrieved schema and cell values thereby maintaining a manageable context window.

4.4.3 Iterative Solver Loop

The core of the framework is the solver_loop, which implements the ReAct paradigm. This loop allows the model to "think" and "act" cyclically until a termination condition is met. At each iteration, the model analyzes the current interaction history to generate a "Thought" regarding the next logical step. When an external calculation is deemed necessary, the model outputs an "Action" consisting of executable Python code. The system parses this code and runs it within a sandboxed environment, where the Pandas DataFrame is pre-loaded as the variable df to enable direct filtering and aggregation. The output of this execution is captured as an "Observation" and appended to the context, driving the loop repeatedly until the model produces a "Final Answer" or the maximum recursion depth is exceeded.

5 Replication & Approach

Our goal in this project was to independently reimplement the TableRAG pipeline and evaluate how well it reproduces the Million Token Table findings. The authors shared the code for the pipeline [3] but not the datasets they used, as such we attempted to recreate the dataset. Our data and results from our tests are available in the appendix.

The paper used a variety of datasets to benchmark TableRAG. These included BIRDQA (Question-Answer) [4], TabFact [5], and ArcadeQA [6]. Notably, BIRDQA and ArcadeQA are not official datasets and were derived by the authors from the publicly available BIRD and Arcade datasets. The authors processed those datasets to have a different format than originally intended. BIRD is a Text-to-SQL benchmark with answers in the format of a SQL query and Arcade is a natural-language-to-Python/pandas benchmark with answers in Python/Pandas. Neither original format was compatible with the TableRAG pipeline.

5.1 BIRDQA Construction

To create BIRDQA, the authors selected the questions that queried only one table, then the authors executed the SQL queries from the BIRD question set, extracted the result, and converted the format into a simple Question–Answer format. Below is an example of the two formats.

BIRD format:

```
{
  "db_id": "database_name",
  "question": "...",
  "SQL": "SELECT ...",
  "evidence": "...",
  "difficulty": "simple"
}
```

TableRAG format (JSONL):

```
{
  "table_text": [[col1, col2], [val1, val2]],
  "question": "...",
  "label": "answer",
  "table_caption": "...",
  "table_id": "...",
  "id": "..."
}
```

They also flattened the referenced database tables into a single 2D structure stored in `table_text`. The `label` field holds the executed query result. This format enables evaluation of both retrieval quality (did we find the right columns/cells?) and end-to-end accuracy (did we get the right answer?).

5.2 BIRD-Mini

BIRD-Mini is a subsection of the larger BIRD database. It is composed of "500 high-quality SELECT-only instances ... carefully selected from the original BIRD dataset." These questions concern data in 11 databases and 93 tables, which is a subsection of the larger BIRD dataset. In our replication we used this smaller subset to better understand how the factors of embedding size affect the accuracy of the test.

5.3 ArcadeQA

Arcade is a natural language to code dataset. Again the authors converted from a code-based answer to a value-based answer by selecting for the questions where the output was a single value. Then, they executed the answer code to extract that value and used it as the answer, creating ArcadeQA.

ArcadeQA and BIRDQA were not publicly available, nor was the code to produce them. After reaching out to the authors of *The Million Token Table* we received a response that the datasets and the code to produce them were being withheld while waiting for release approval from Google, the sponsor of the study.

5.4 TabFact

TabFact is a dataset derived from 16,573 Wikipedia tables. Each question is a statement that has been human annotated ENTAILED (True) or REFUTED (False). Differing from BIRDQA and ArcadeQA, which are question-answering datasets where the LLM is tasked with finding the values in a database, TabFact tests whether a statement is supported by table evidence. To test the limits of TableRAG, the authors generated synthetic data in the TabFact format to create 1000x1000 size tables.

6 Tests & Analysis

6.1 General Experimental Setup

To ensure consistency across tests we standardized the test configuration, unless otherwise noted, all tests used the following parameters.

- **Mode:** Embedding-based retrieval
- **Embedding model:** text-embedding-3-large (OpenAI)
- **Retriever type:** TableRAG (schema + cell retrieval)
- **Configuration:**
  - top-k = 5
  - max_encode_cell_budget = 10,000

A table with full data of the results can be found in the appendix.

6.2 Reproduction Attempt: Small Benchmark (TabFact)

To validate the correctness of our implementation of the TableRAG pipeline and ensure that our processing of the data was correct, we first attempted to reproduce the results of the TabFact benchmark on a small subsection of the data (350 individual tests).

Experimental Setup:

- Model: GPT-3.5-turbo-0125
- Dataset: TabFact test split, converted to TableRAG JSONL using custom conversion
- Questions: 350 samples from different sections of the question set

During preprocessing, each table was embedded into a FAISS vector index using the OpenAI API. These vectors were then used during the Tabular Query Expansion of the TableRAG pipeline to give the LLM access to table context while generating Pandas commands to retrieve specific cells.

Performance Metrics

- Accuracy: 85.14%
- Average tokens per test: 2516 (including initial prompt: 815 tokens)

Analysis

Our test run on a subsection of the TabFact dataset slightly exceeded that paper's accuracy of 83.1 demonstrating a successful implementation of the TableRAG approach. The difference (2.1%) is likely due to test set variance (350 questions vs full set) or possibly small variability in the GPT-3.5-Turbo API output as LLMs are non-deterministic.

One important thing of note is the table characteristics for the original TabFact data. The table size of our selection is generally small, with an average size of 13 by 7 (rows, columns). The implication is that there was little to no truncation of the data during embedding.

Strategic truncation of data is the paper's most important innovation, but this effect is only displayed when the tables are larger than the embedding size. When we tested the TabFact dataset *without* expansion there was little effect on the accuracy. In fact, it improved by 1%. Though this may seem to contradict the paper findings, we believe it does not. Tabular expansion's purpose is to condense table data that does not fit into the context window of the LLM; for the sequential subset of questions that we initially tested, all of the table data was able to be fit into the context window of the LLM. Consequently, the model performed slightly better with the raw table data, likely because the retrieval mechanism—designed to compress massive tables—introduced unnecessary processing steps for these small inputs. This initial test did not reflect the intended use for TableRAG due to the smaller table size.

6.3 Reproduction Attempt: BIRD-QA (Reconstructed)

After validating that the pipeline worked, we attempted to run a larger and more difficult dataset. Compared to TabFact, BIRD is a much more challenging benchmark both in size and difficulty. The largest number of rows in a BIRD table is 1,056,320 compared to the 48 in the subset of TabFact tests we ran. Furthermore, the TabFact answers were restricted to True or False while the BIRD questions asked for specific values making it much more challenging.

Experimental Setup:

- Model: gpt-3.5-turbo-0125
- Dataset: BIRD-QA (Reconstructed)
- Questions: 393 samples from different sections of the question set

Like before, during preprocessing, selected cells from each table were embedded into a FAISS vector index using the OpenAI API. Unlike before, there was much less coverage of the tables in the embedding; most of the tables were not fully embedded and some of the largest had less than 1% embedded. What follows is a breakdown of table embedding coverage.

Table Encoding Coverage for BIRD-QA (Reconstructed)

- Max encode cell limit: 10,000 cells
- Tables fully encoded: 125 (31.8%)
- Tables truncated: 268 (68.2%)
- Average coverage: 41.10%
- Median coverage: 12.92%
- Min coverage: 0.09%
- Max coverage: 100%

Coverage Distribution

Coverage Range	# Tables
100% (fully encoded)	125
50–99%	24
10–50%	55
1–10%	82
<1%	107

Test Results

- Accuracy: 29.52%
- Average tokens per test: 4,276.2

Analysis

The more challenging tests show a sharp dropoff in accuracy at 29% compared with the previous tests at 85%. This is consistent with the study which had 45% accuracy, 40 percentage points lower than the TabFact accuracy. We believe the large difference in the accuracy of our replication is due to an error in processing the BIRD dataset to be compatible with the TableRAG format. The paper filtered out only the questions that used a single table from the BIRD dataset resulting in 308 questions. They did not specify which questions in the paper, and when we attempted the same process we found 393 single-table questions.

In the test we had both low accuracy and low retrieval, indicating the model often lacked sufficient context for writing Pandas queries and reasoning. In some cases the model would refuse to answer, commenting on the lack of information. One example is test 38 where the answer given was "Unfortunately, the most common bond type cannot be determined based on the given information."

To evaluate the paper's claims that their tabular expansion improved the accuracy we ran the BIRD-QA (Reconstructed) tests both with and without tabular expansion, this time using a newer model GPT-4o-mini for both tests. With these tests we saw a 6.92% increase in accuracy with expansion over without expansion. The 4o-mini with expansion had an accuracy of 35.3% while without expansion had an accuracy of 33%. This is slightly lower than the paper's increase of 9% (50.7% vs 46.5%). We believe that the differences largely come from differences in data processing or models used. Interestingly, the newer model outperformed the GPT-3.5 tests without expansion, indicating that token limit (16,285 for 3.5 Turbo vs 128,000 for 4o) and baseline model performance may be a stronger factor in accuracy than preprocessing.

6.4 Reproduction Attempt: Arcade-QA (Reconstructed)

We also attempted to replicate the finding with the Arcade dataset. In those tests, we found that expansion increased the accuracy of the test, though the baseline performance of the tests were much lower than the study's. This difference is due to difficulty in replicating the ArcadeQA dataset which was not provided by the study.

Again, the processed data was not provided nor the method for producing it. In the study they mentioned

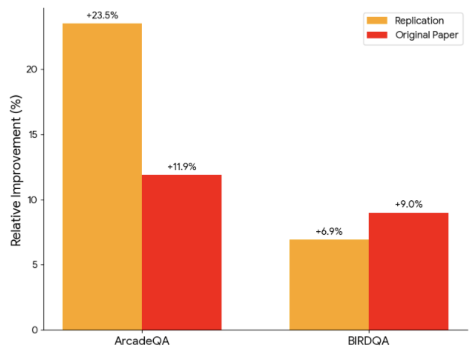

Figure 2: Relative Accuracy Improvement from Tabular Expansion

that they selected tests that were questions rather than statements, such as “Generate code that...”. To replicate their data we then needed to search the Arcade data for “question words” (how, what, etc.); it is likely that our set of question words differed from the authors as our processing resulted in fewer tests than theirs. It is likely that this difficulty in processing the data was the cause of the difference in accuracy.

In our tests, we found that expansion yielded a relative accuracy improvement of 23.53%, up from 16.67% without expansion to 20.59% with expansion. The percentage increase is greater than that found in the study, 11.91%, but the overall accuracy is much lower, with the study’s accuracy with expansion being 57.3%.

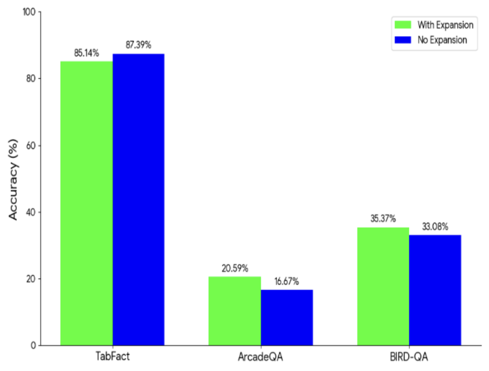

Figure 3: Impact of Tabular Expansion Across Benchmarks During Replication

7 Extension & Novelty

7.1 Cell Encoding Budget Analysis

The original TableRAG paper proposes a cell encoding budget (B) to cap token usage during the cell retrieval phase. The authors justify a default limit of 10,000 cells in Figure 6 of their paper, showing accuracy plateaus beyond this threshold. To independently validate this claim and explore the relationship between encoding budget and performance, we conducted a systematic series of experiments on the BIRD-Mini dataset.

BIRD-Mini consists of 92 high-quality single-table questions drawn from 11 databases. The tables in this dataset vary significantly in size, ranging from 14 cells to over 7.7 million cells, providing a representative cross-section for analyzing how embedding size affects retrieval quality across different scales. While the replication in Section 6 used our broad extraction of 393 questions, our detailed parameter analysis in this section focuses on a high-quality subset (BIRD-Mini) to reduce noise allowing for a more detailed analysis of the effect of embedding budgets.

7.2 Experimental Setup

We tested ten distinct cell encoding budgets using two models. For GPT-3.5-turbo-0125, we evaluated budgets of 5, 10, 25, 50, 100, 500, 10,000, and 20,000 cells. By necessity, we used GPT-4o-mini to test higher budgets of 50,000 and 100,000 cells to explore behavior at extreme scales. All experiments used the text-embedding-3-large embedding model and maintained consistent retrieval parameters (top-k = 5 for schema and cell retrieval).

7.3 Key Findings

7.3.1 Finding 1: Non-Monotonic Relationship

Contrary to initial expectations, accuracy does not simply increase with larger encoding budgets. The results reveal a U-shaped curve where both very small budgets (5 cells) and large budgets (10,000–20,000 cells) outperform mid-range values (10–500 cells). At budget = 5, accuracy reached 33.70%, matching the performance of budget = 10,000, while mid-range budgets like 50 and 100 achieved only 23.91–25.00%.

7.3.2 Finding 2: Optimal Ratio Discovery

We discovered that the most critical factor is not the absolute encoding budget, but rather the ratio between the budget and the total table size. Our analysis shows that when the encoding budget represents 10–20% of total table cells, accuracy peaks at 56.5%. This insight provides a practical heuristic for adaptive budget selection based on table characteristics.

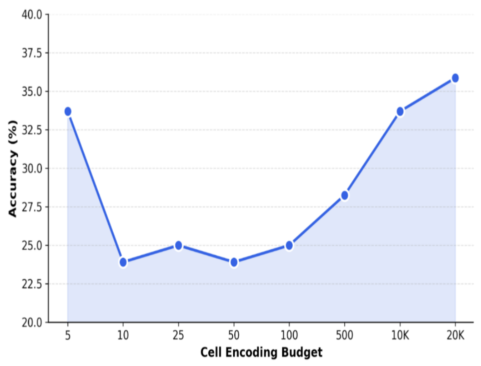

Figure 4: Accuracy vs Cell Encoding Budget

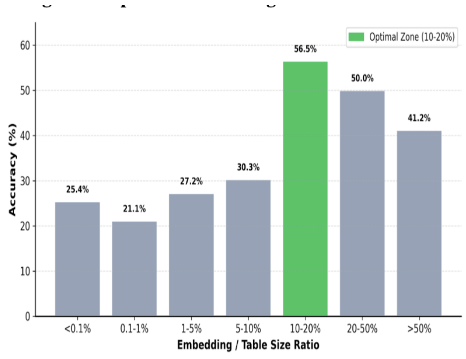

Figure 5: Optimal Embedding-to-Table Size Ratio

7.3.3 Finding 3: Minimal Encoding Effectiveness

Perhaps the most surprising result is the effectiveness of extremely small encoding budgets. With only 5 cells encoded, accuracy reached 33.70%—more than double the 14.13% achieved with no cell encoding at all. This suggests that schema retrieval is doing most of the heavy lifting for query understanding, with cell retrieval serving primarily as a disambiguation or grounding signal rather than the primary source of context. In other words, knowing which columns matter appears more critical than knowing which specific values they contain.

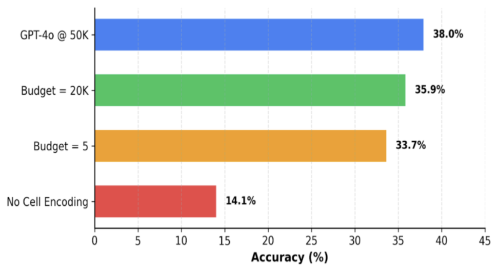

Figure 6: Impact of Cell Retrieval on Accuracy

Table 1: **Model Comparison at High Embedding Budgets**

Configuration	**Accuracy**	**Init Tokens**	**Total Tokens**
GPT-3.5 @ 20K	35.87%	1064.8	4227.2
GPT-4o-mini @ 50K	38.04%	1191.6	5978.9
GPT-4o-mini @ 100K	38.04%	1160.2	6352.4

7.3.4 Finding 4: Saturation at High Budgets

Testing GPT-4o-mini at budgets of 50,000 and 100,000 cells revealed a saturation effect. Both configurations achieved identical accuracy of 38.04%, indicating that increasing the budget beyond 50,000 provides no additional benefit. Encoding 100,000 cells instead of 50,000 doubles the embedding computation and storage costs while yielding zero accuracy improvement; a clear case of diminishing returns that practitioners should avoid.

7.4 Performance by Table Size

Our analysis revealed that optimal embedding configuration depends significantly on table scale. For tiny tables (less than 1,000 cells), a minimal budget of 5 cells achieves 68.8% accuracy with GPT-3.5—matching the best GPT-4o-mini configuration at 100,000 cells. For medium tables (10,000–100,000 cells), a budget of 20,000 cells with GPT-3.5 achieves 80% accuracy, outperforming GPT-4o-mini at higher budgets by over 30 percentage points. For very large tables (over 1 million cells), budgets of 10,000–50,000 cells achieve similar accuracy around 36%. At this scale, the bottleneck shifts from cell coverage to retrieval precision—*with millions of cells, even 50,000 embeddings represent less than 5% coverage*, meaning the system's ability to retrieve the right cells matters more than how many cells are indexed.

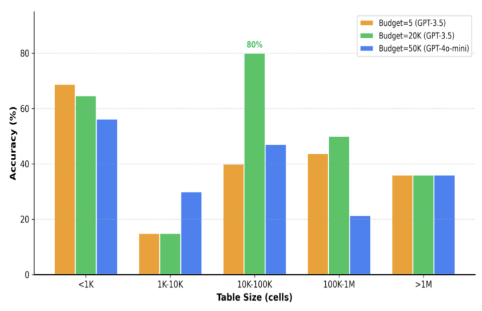

Figure 7: Accuracy by Table Size Category

7.5 Implications & Conclusion

These findings refine the original TableRAG paper's Figure 6 recommendations in several important ways. Rather than using a fixed budget of 10,000 cells, systems should adaptively select budgets based on table size, with a target ratio of 10–20% of table cells providing optimal retrieval quality. Our discovery of the U-shaped accuracy curve suggests that for small tables, minimal encoding may actually outperform larger budgets by avoiding the introduction of noise, while medium-sized tables benefit from higher budgets around 20,000 cells.

For budget-constrained deployments, GPT-3.5 with budget = 20,000 achieves 35.9% accuracy; for highest accuracy requirements, GPT-4o-mini with budget = 50,000 achieves 38.0% but at approximately 40% higher token cost. However, increasing beyond 50,000 cells provides no measurable benefit. The success of minimal cell encoding (5 cells achieving 33.7%) highlights that schema retrieval provides the primary signal for many queries, with cell retrieval serving a supplementary grounding role. Notably, 46.7% of questions fail regardless of configuration, pointing to opportunities for future work in query classification and hybrid reasoning strategies.

Overall, this extension provides empirical evidence that cell encoding budget is not a one-size-fits-all parameter and requires calibration based on table characteristics and deployment constraints.

8 Appendix

Full processed question sets as well as per test output can be found here:
https://drive.google.com/drive/folders/1Q0_2vuGblUyrTTPQmWHb6Tykv58Hy4XM

8.1 Section 6. Tests and Analysis Data

Dataset	Model	Retrieval Strategy	Accuracy	Avg Iterations	Avg Total Tokens
TabFact	GPT-3.5-Turbo	No Expansion	87.39%	2.32	1,663
TabFact	GPT-3.5-Turbo	TableRAG (Expansion)	85.14%	2.36	2,516
ArcadeQA	GPT-3.5-Turbo	No Expansion	16.67%	3.16	3,188
ArcadeQA	GPT-3.5-Turbo	TableRAG (Expansion)	20.59%	3.47	5,841
BIRD-QA	GPT-3.5-Turbo	TableRAG (Expansion)	29.52%	3.18	4,276
BIRD-QA	GPT-4o-mini	No Expansion	33.08%	3.04	4,113
BIRD-QA	GPT-4o-mini	TableRAG (Expansion)	35.37%	3.25	6,588

8.2 Section 7. Extension and Novelty Data (BIRD-Mini)

Model	Cell Encoding Budget	Accuracy	Avg Iterations	Avg Total Tokens
GPT-3.5-Turbo	0 (No Embedding)	14.13%	4.52	2,403
GPT-3.5-Turbo	5 Cells	33.70%	3.42	5,741
GPT-3.5-Turbo	10 Cells	22.83%	3.57	4,143
GPT-3.5-Turbo	25 Cells	25.00%	3.45	4,131
GPT-3.5-Turbo	50 Cells	22.83%	3.49	4,099
GPT-3.5-Turbo	100 Cells	23.91%	3.60	4,083
GPT-3.5-Turbo	500 Cells	28.26%	3.51	4,164
GPT-3.5-Turbo	1,000 Cells	21.74%	3.47	4,457
GPT-3.5-Turbo	10,000 Cells	34.78%	3.36	4,191
GPT-3.5-Turbo	20,000 Cells	34.78%	3.28	4,227
GPT-4o-mini	50,000 Cells	36.96%	3.15	5,979
GPT-4o-mini	100,000 Cells	35.87%	3.40	6,352

9 References

[1] S. Chen et al., "TableRAG: Million-Token Table Understanding with Language Models," Advances in Neural Information Processing Systems (NeurIPS), 2024. arXiv:2410.04739.

[2] A. Biswal et al., "Text2SQL is Not Enough: Unifying AI and Databases with TAG," Conference on Innovative Data Systems Research (CIDR), 2025. arXiv:2408.14717.

[3] Research Codebase, "Exploring Table-Augmented Generation for Tabular Data," GitHub, 2025. [Online]. Available: https://github.com/readygroup1/TableRAG

[4] J. Li et al., "Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database-Grounded Text-to-SQLs," Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2305.03111.

[5] P. Yin et al., "Natural Language to Code Generation in Interactive Data Science Notebooks," Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL), 2023. arXiv:2212.09248.

[6] W. Chen et al., "TabFact: A Large-scale Dataset for Table-based Fact Verification," International Conference on Learning Representations (ICLR), 2020. arXiv:1909.02164.

[7] Google Research, "TableRAG Implementation," GitHub, 2024. [Online]. Available: https://github.com/google-research/google-research/tree/master/table_rag

10 Reproducibility

Due to the large size of our working repository (71 GB with datasets and pre-built databases), we could not provide it as a cloneable GitHub repository. Alternatively, we have included links to our converted question sets and results, the original TableRAG code from the Google Research repository, and links to all the original datasets which are needed to build databases to run tests against. Complete step-by-step instructions are provided in the `ReproductionGuide.md` included with our data at https://drive.google.com/drive/folders/1Q0_2vuGblUyrTTPQmWHb6Tykv58Hy4XM?usp=sharing.

Our submission includes:

- **Experiment Results (67 MB)**: Pre-computed outputs from 20 experimental configurations across BIRD, TabFact, and ARCADE datasets
- **Processed Question Answer Datasets (2.7 GB)**: JSONL files used in our experiments
- **ReproductionGuide.md** which includes the specific commands to reevaluate our results and rerun the tests.

To reproduce or verify our results:

1. **Re-evaluate our results:** Clone the Google Research TableRAG repository from https://github.com/google-research/google-research/tree/master/table_rag, download our Results folder, and run the evaluation script (further instructions in Google Drive). No API key or database downloads required.

2. **Re-run experiments from scratch:** Requires an OpenAI API key and will incur API costs. Databases which our question sets run against can be obtained from official sources:

- **BIRD**: https://bird-bench.github.io/
- **TabFact**: https://github.com/wenhuchen/Table-Fact-Checking
- **ARCADE**: https://www.kaggle.com/datasets/nirmalgaud/arcade-dataset