Agentic Table Search for Kitana Queries

Mateo Juliani
msj2164@columbia.edu
Columbia University

Nicky Collins Khorasani
nck2130@columbia.edu
Columbia University

Kaushal Damania
kd2990@columbia.edu
Columbia University

ACM Reference Format:
Mateo Juliani, Nicky Collins Khorasani, and Kaushal Damania. 2025. Agentic Table Search for Kitana Queries. In *Proceedings of W6113.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 ABSTRACT / INTRODUCTION

Given a base table and a target column, Kitana - a data-centric AutoML system - searches through a database to find additional tables and columns to join onto the original table to improve the prediction accuracy of the target column. However, Kitana requires clean tables to join onto the original query table. Therefore given a larger corpus of uncleaned tables, one challenge is how to decide which tables to clean and pass to Kitana to increase the prediction accuracy of the underlying target column. A traditional approach might include using word embeddings to find similar tables to the target table / column however this method 1) might not be able to grasp the true semantic meaning of a table, 2) does not take advantage of the Kitana's query history, and 3) is not sophisticated enough to find important columns that are only accessible through multi-joins. In this paper, we propose and test different agentic system to address these limitations. First, we propose agents to find single tables that are most likely to improve the accuracy of the initial target column given Kitana's search history (denoted No-hop case). We find that our system is able to find single tables that increase the R^2 *of the original target column by 24% and achieve a 12.5% higher* R^2 *more than the naive embedding approach. Second, we propose an agent capable of identify a sequential list of join operations to join tables on the target query that would not have been accessible through a single join (multi-hop case). We compare various exploration strategies for finding these joins in a cost effective way.*

1.1 One Sentence Summary

We propose agentic systems to find tables and columns that improve the accuracy of machine learning (ML) prediction problems.

1.2 Use Case, Audience, and Needs

This paper is primarily for data scientists, ML engineers, and researchers who use ML models to predict target variables. Data is the main resource powering these models, and finding the right data is key to accurately explaining a target variable. However, with ever growing datalakes, it is becoming more and more difficult to find the right data to power these models. This paper details agentic systems to help ML users find data relevant to their queries. Without this ability, these individuals and their institutions would suffer from sub-optimal results and a loss of productivity.

W6113, May 2025, New York, New York, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELATED WORK

At its core, the problem we are trying to solve is a tabular search problem: finding the best table for a given query. Below, we provide an overview of how other papers have explored similar problems.

2.1 Finding Related Tables:

Earlier solutions have looked to use word embeddings ([1], [14], [12]) on a table's schema or other natural language present in the table to find similar tables. Certain Text2SQL papers ([3]) also use this method. However, methods that look to use word embedding methods struggle to provide the proper contextual information for numeric data. Further, these methods might ignores row information (particularly when they just embed the schema), which is essential to understanding the true context of a table. Consequently, other works have explored searching for specific columns or rows related to the underlying query ([2], [10]). These methods either result in encoding an entire row, which could become expensive depending on the table size, or limiting the search to a few cells in a table. The latter method is helping for answering a specific query, however for finding tables with entire columns to augment a query, this method might be too narrow. Furthermore, all of these methods look to find tables based on some form of similarity. While that might work for Text2SQL or question answering, a column that is similar to the target column or underlying query table does not guarantee that it will have increased predictive accuracy. While the paper pulls uses prior tabular search methods, the main differentiating factor is using the Kitana augmentation history to aid the search for which tables to add to the cleaned database.

2.2 Finding Join-able Tables:

Finding join-able tables in data lakes is another crucial aspect of tabular search that presents unique challenges. Zhu's work [15] on search and join algorithms for tables in data lakes highlights the complexity of identifying meaningful join candidates among vast collections of heterogeneous tables, particularly when join conditions are not explicitly defined. Traditional approaches for finding join-able tables often rely on syntactic matching of column values or metadata similarity, which can miss semantically valid join opportunities. DeepJoin [4] addresses these limitations by leveraging pre-trained language models to capture deeper semantic relationships between tables, enabling the discovery of non-trivial join candidates that conventional methods might overlook. These approaches demonstrate significant improvements over schema-matching techniques; however, they primarily focus on finding

tables that can be joined rather than tables that will improve predictive performance once joined. This distinction is crucial, as joinability does not necessarily correlate with usefulness for prediction tasks. Furthermore, most existing join discovery methods are optimized for single-hop scenarios and struggle with the complex path planning required for multi-hop join sequences, where intermediate tables must be identified and joined in a specific order to reach the most valuable augmentation tables.

3 PROBLEM, CAUSE, AND INSIGHT

3.1 Problem Definition

At a high level, the problem can be seen as a data table search problem, where a user wants to find a table that is most likely to help predict a target column in another table. More formally, given a query q_i, which is composed of a table and a target column, our goal is to find a subset of tables T' from a corpus of tables T to clean and include in the database that Kitana searches. For each q_i, we have a set of g_i of past tables that Kitana found to augment the accuracy of q_i, along with the corresponding increase (or decrease) of the R^2 value, denoted as a_i. We denote the budget to search for the table as b_i.

3.2 Cause

Ever-growing data lakes filled with messy data create the need to 1) search through a data lake to find tables related to a user's underlying query and 2) clean tables once found. However, both of these tasks are non-trivial. For searching through tables, a RAG approach could be used; however, this approach can suffer either from improper or missing contextualization of the table [6] or incur high costs if the entire table is embedded [2]. Further, while a baseline embedding may find contextually similar tables, they can fall short when identifying join-able tables. Similarly, an embedding method cannot identify multiple tables that must be joined together first, before joining onto the query table. We denote this phenomenon as the multi-hop case.

Even after the tables are located, further challenges remain in structuring them in the correct format to join onto the main query table. Several papers, including Cocoon [7] and Jellyfish [13], have been proposed to use LLM's to process and clean data. While likely superior to manually cleaning a dataset, these methods still incur high costs and have a limit on how many tables can be cleaned and processed at a time.

3.3 Insight

We believe that an agentic table search system can help overcome these problems. Our first insight is that LLMs excel at extracting and inferring nuanced semantic meaning from tables ([7] [5]). This nuanced semantic meaning can help differentiate which tables can provide value to a query from those that cannot. For example, suppose that two housing datasets can join onto a Kitana query for a table called *Sydney_housing.csv*. One is called *melb_housing.csv*, while the other is called *hou_housing.csv*. While a simple word embedding model might rank the two potential tables as equally likely to enhance the query, an LLM might discern that *melb_housing.csv* could include data from Melbourne, which might be more helpful than housing data from Houston. Further context from the table, such as addresses or longitude and latitude coordinates, could provide additional insights that an embedding model could not discern. Several papers ([7] [13]) have explored this topic.

Our second key insight is that while semantic similarity is a good starting point to find tables that could enhance a query, there is no guarantee that semantically similar tables could improve the accuracy of the underlying table. In fact, in some cases, more orthogonal tables or column names could enhance the query more. For example, if you are looking to predict flight delays, weather, although not the most semantically similar word to flight, might be a strong predictor. Instead, having a source of truth could enhance the query more. The Kitana query history can provide this source of truth and act as guidance for which tables/columns can help improve the query the most. Additionally, given budget constraints, this past query history can dictate where to look. For example, if we have previously tested augmenting a housing price dataset with local school quality, cleaning further datasets related to school quality might not be the best use of the budget.

Our final key insight is that these traditional approaches only consider one table at a time. Consequently, situations where linking multiple tables through intermediary join keys (e.g., table A is joined with table B, and table C is subsequently joined with table B via distinct attributes) would be missed or ignored even though they might unlock more potential in augmenting a query. We denote this case the multi-hop case. The primary difficulty of multi-hop lies in accurately tracking column lineage and managing the evolving schema across successive join operations to ensure the semantic integrity and utility of the resultant query. Conventional approaches relying solely on single-pass semantic similarity assessments are often inadequate for resolving ambiguities and identifying optimal join paths in these complex scenarios. Therefore, we believe that an iterative, multi-turn methodology is required to find these multi-hop scenarios and unlock columns that would not be accessible had we used a traditional approach of comparing similar tables one at a time.

This paper explores two settings. First, it explores the efficacy of three different architectures - naive embedding, LLM Enrich, and Agentic Selector - that look to use table semantics and prior Kitana query histories to search for individual tables that are most likely to increase the accuracy of a query. We denote this case as the No-hop case. Second, this paper introduces a novel method that leverages LLMs for proposing candidate join operations within such multi-hop sequences. To represent the potential joins and track data provenance, our approach employs a dynamic graph structure that records the applied operations and potential table integrations. Furthermore, we investigate and implement expansion strategies designed to navigate the solution space of join sequences efficiently, ensuring adherence to operational budget limitations while maximizing the quality of the integrated data.

4 SOLUTION

4.1 Budget Considerations

Large-scale data lakes can contain hundreds or thousands of candidate tables, making it expensive to evaluate each one using LLM calls. Therefore, to ensure scalability and lower costs, we introduce a query-specific budget constraint b_i that limits the total number

of LLM tokens our agent can use during table selection. The aim is to be more efficient than naively running Kitana across the entire data lake while maintaining high-quality selections.

4.1.1 Greedy Budgeting. In the simplest implementation, we use a greedy budgeting approach. The agent processes candidate tables and tracks the cumulative number of tokens used. Once the total exceeds the query budget b_i, the process halts and the currently selected tables are returned. This method is fast and easy to implement, but can be myopic: tables earlier in the list may be included even if they are marginally useful or token-expensive.

4.1.2 Value-per-Token Budgeting. To improve selection efficiency under budget constraints, we developed a more informed strategy based on the concept of *value per token*. For each candidate table T_j, we estimate its token cost c_j by constructing the actual LLM prompts that would be used for description or join-key analysis. We also assign a value score v_j based on similarity to the query or historical effectiveness.

Given these estimates, we sort tables by their value-to-cost ratio $\frac{v_j}{c_j}$ and greedily select the top entries until the total cost reaches b_i. This heuristic approximates the knapsack problem and prioritizes high-yield, low-cost tables — for example, shorter tables with relevant join keys and minimal metadata.

A key ingredient of the "Value-per-Token" budgeting is the "value" function. In its simplest form the value of a table can be the cosine similarity of the embedding. More complicated approaches could include "distance" from good or bad tables in terms of cosine similarity which we can access from the Kitana history. Due to time constraints, we could only evaluate performance using the greedy budgeting strategy.

4.2 No-hop: Embedding Approach

As a baseline, we developed a naive embedding approach. For any given query, we embed the query's table name and target column. Similarly, we embed the table name along with its column names for each table in the data lake. To decide which tables to clean and add to the Kitana database, we select the top X tables similar to the query table and column based on cosine similarity. Figure 1 details the table selection process.

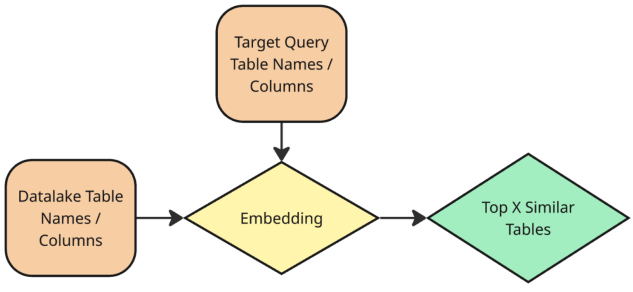

Figure 1: Baseline Embedding Architecture

4.3 No-hop: LLM Enrich

Our next approach is the "LLM Enrich method", which looks to address some of the baseline embedding's limitations. Our first observation is that the embedding approach is unable to find tables that are join-able with the target query. For example, the embedding approach would prioritize a table with column names that are more similar to the query column's even though there are no columns that could be joined onto the query table. Our second observation is that the column value's themselves provide important information that embeddings cannot interpret well. Some examples include data issues (such as missing values) and semantic meaning of the columns, in particular when the column values are numeric. The LLM Enrich approach attempts to overcome these limitations.

The baseline architecture is as follows, and can be seen in Figure 2. First, we use the embedding approach to find Y semantically similar tables. Next, we prompt an LLM to 1) identify the potential join keys in the table and 2) provide a brief description of what the column represent. Finally, we ask the LLM whether or not the table is join-able with the target table. We consequently filter out any table that the LLM determines is not join-able with the query table. For the remaining tables, we prompt the LLM to describe the table given a few samples of the table. We embed these descriptions, along with the description of the query table, and then select the tables with the highest similarity to the query table based on cosine similarity.

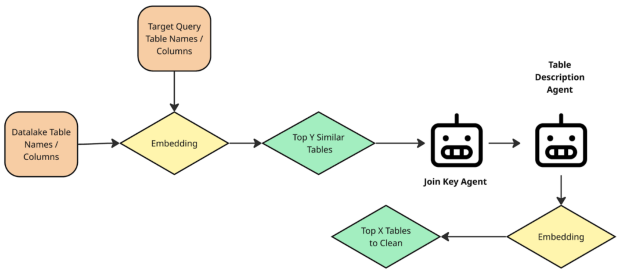

Figure 2: LLM Enrich Agent Framework

4.4 No-hop: Agentic Selector

While the LLM Enrich approach potentially excels in finding join-able, semantically similar tables, one down side is the number of LLM calls required. For one query, at most 4(Y + 1) (3 for the join agent, 1 for the table description for each table plus the query table) calls are required, which could become expensive. Further previous works have shown that LLMs can be effective selectors or rankers ([10],[11]). Therefore, an LLM agent might be able to directly select which tables are most likely to enhance the query. The LLM selector framework is as follows and is depicted in figure 3. Similar to the LLM enrich agent, we start off by selecting Y similar tables from our data lake to the query table. Given these Y tables, we split them up into K sets. For the first set, we prompt the LLM to select the top X tables to enhance the target query given 1) the query table name + columns, 2) the prior tables that helped increase the accuracy the

most and least from the Kitana history and 3) the table's names and columns in the set. We append the tables outputted from the prior set to the next set, and run the process again. At the end, the system outputs X final tables to clean. One immediate benefit of this framework versus the LLM Enrich method is the number of LLM calls. At most, the agent will perform $\frac{Y}{K}$ calls. We explore how the choices of Y and K impact accuracy in section 5.3.

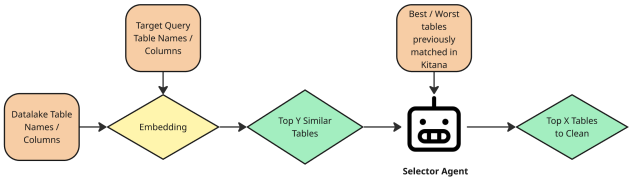

Figure 3: LLM Selector Agent Framework

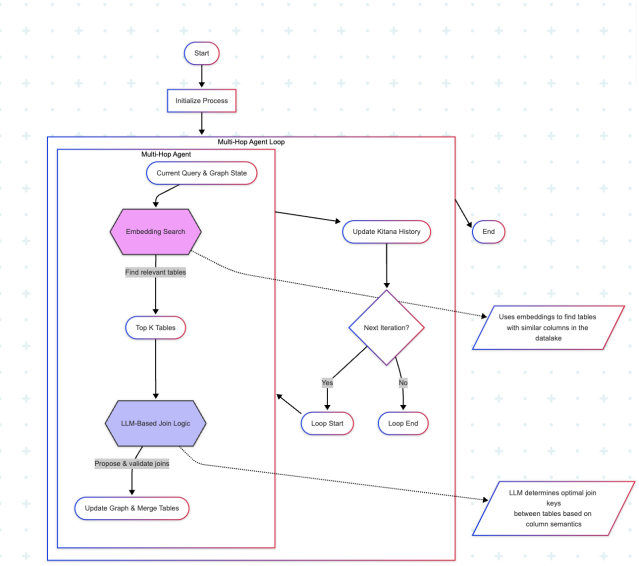

Figure 4: Multi-Hop Agent Framework

4.5 Multi-Hop

The selector agent previously described focused on finding the optimal tables for a single, direct query. However, its design does not extend to effectively managing multi-hop join scenarios, which involve sequential table linkages. Given that the Kitana system requires the specification of join keys in conjunction with table selection, a sophisticated framework is necessary to not only identify relevant tables but also to propose viable join keys for constructing these relational chains. The successful formation of such join chains is contingent upon feedback from Kitana regarding the efficacy of each join operation, thereby necessitating an iterative, multi-turn approach. This iterative methodology is visually represented in Figure 4.

The process commences with a search operation, informed by the initial query and the current state of the evolving graph that represents table integrations. This search can leverage any of the methodologies detailed in prior sections. Following the identification of candidate tables, an LLM is employed to propose potential join keys. The table resulting from a proposed join is then submitted to Kitana for validation of the merge's success.

A significant consideration in constructing join chains is the incorporation of metadata and data samples from multiple tables. This can lead to a substantial increase in computational resource consumption, potentially exhausting the allocated budget if metadata from all entities in extensive potential join paths are considered. To address this challenge, our strategy prioritizes the evaluation of tables based on their individual characteristics, rather than formulating subsequent search queries based on the full context of already integrated tables within a chain. This targeted approach, however, introduces the critical decision point of determining the optimal order for node expansion within the search space.

To this end, we explore fundamental graph traversal algorithms, such as Breadth-First Search (BFS) and Depth-First Search (DFS). To enable a more informed search process, we integrate feedback from Kitana, specifically leveraging the performance metrics of completed join operations to guide subsequent decisions on which nodes (tables or intermediate join results) to expand. This guided expansion is informed by the heuristic that join paths which have already contributed substantial information may exhibit diminishing returns for further augmentation, thereby optimizing the search within budgetary constraints.

5 EVALUATION

5.1 Evaluation Framework and Metrics

For our evaluation framework, we started by curating approximately 500 tables in our data lake. These tables represent uncleaned tables that Kitana cannot access when searching for tables to augment the initial query. These tables were mostly pulled from Kaggle and provide data largely on real estate, and country statistics (focused on economic, medical, environmental, political, and societal data points).

For the No-hop case, we then create 17 test queries with target columns including housing prices, cost of living indices, petroleum consumption, and pollution rates. For each test case, we create an initial starting database from which Kitana can pull from to augment the original query. Figure 5 provides an overview of each test case.

For each query, we will run Kitana on the initial database to get a baseline R^2 value. Then, we will run each of our methodologies on the data lake, and select 5 tables to add within the selected budget B. Afterwards, we will run Kitana again on the new dataset and measure the R^2 improvements. To find the upper bounded R^2 value, we will run Kitana on the full data lake. In some cases, certain queries have a corresponding table that achieves a high R^2 (≥ 0.98), which will be used as the upper bound.

For our Multi-Hop agent, the evaluation does not focus on the effectiveness of the table search component, as any of the aforementioned search techniques can be employed to identify join candidates. Furthermore, the use of LLMs to propose join operations has been previously established [9]. Consequently, our evaluation for multi-hop scenarios centers on assessing the effectiveness of

Test Case	Data File Name	Target Column	Group By Columns
1	master.csv	suicides_no	Country, year
2	raw_data.csv	human_development_index	Country
3	Cost_of_Living_Index_by_Country_2024.csv	Groceries Index	Country
4	housing_geo_data.csv	median_house_value	latitude, longitude
5	house_details.csv	Price	Address
6	crime-rate-by-country-2023-base.csv	crimeIndex	country
7	quality_of_life.csv	stability	country
8	quality_of_life.csv	rights	country
9	quality_of_life.csv	health	country
10	quality_of_life.csv	safety	country
11	quality_of_life.csv	climate	country
12	quality_of_life.csv	costs	country
13	quality_of_life.csv	popularity	country
14	corruption.csv	corruption_index	country
15	corruption.csv	annual_income	country
16	Petrol Dataset June 20 2022	Daily Oil Consumption	country
17	most-polluted-countries.csv	pollution_2023	country

Figure 5: List of Test Queries

various chain expansion strategies under budgetary constraints. To evaluate these techniques, we utilize synthetically generated join chains. These chains are constructed using LLM calls to suggest related items or entities. We then place correlated data at various locations within these fabricated chains. The use of synthetic data is justified in this context as it allows for controlled experimentation to specifically test the efficacy of different expansion strategies, particularly given the difficulty of finding real-world datasets with readily verifiable, complex multi-hop join structures and known optimal paths. Similar to the No-hop case, we evaluate the efficacy of the agent based on achieved R^2. For all test with either use gpt-4o-mini or Gemini 1.5 Flash.

5.2 Results

5.2.1 No-hop results. Figure 6 showcases the results for the 17 No-hop tests. The average baseline R^2 was 0.535 and the average maximum R^2 was 0.765. Overall, the base embedding approach achieved an average R^2 of 0.594, while the LLM Enrich and LLM Selector agents improved on that score by around 10%-12%.

Surprisingly, increasing the budget only marginally improved the R^2 for the LLM Enrich and Agentic Selector methods. In some cases, adding more budget decreased performance, suggesting that more tables and exploration might decrease performance (potentially because of the larger context / increase numbers of decisions the LLM needs to make). We further explore this concept in section 5.3.2.

Agent Type	R^2 by Token Budget (B)				
Budget	B=2000	B=5000	B=10,000	B=15,000	B=20,000
Baseline	0.535	0.535	0.535	0.535	0.535
Embedding	0.594	0.594	0.594	0.594	0.594
LLM Enrich	0.632	0.639	0.635	0.622	0.593
Agentic Selector	0.670	0.671	0.675	0.646	0.681
Upper Bound	0.765	0.765	0.765	0.765	0.765

Figure 6: Average R^2 from 17 No-hop test cases

5.2.2 Multi-hop results. Figures 7 and 8 illustrate the performance of different expansion strategies under varying budgetary constraints, defined as a percentage of the total budget required to complete all join chains.

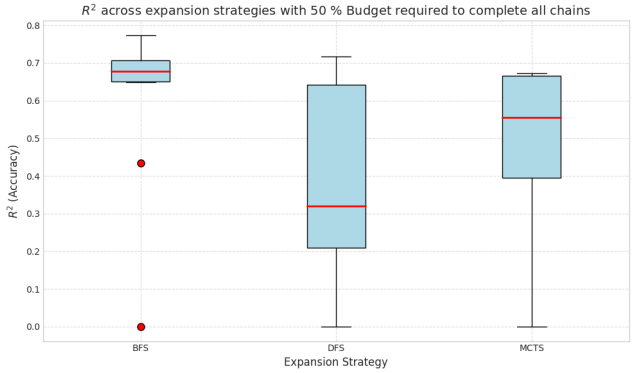

Figure 7: R^2 across expansion strategies with 50 % Budget required to complete all chains

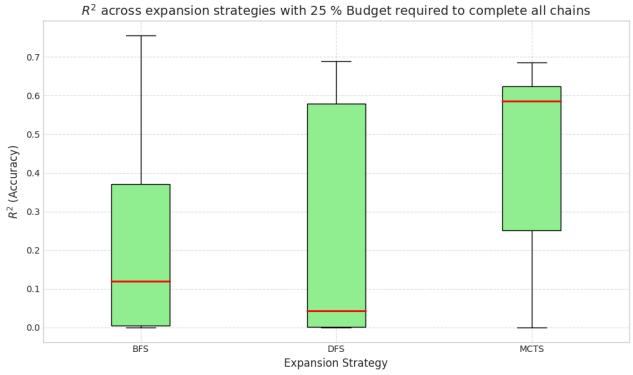

Figure 8: R^2 across expansion strategies with 25 % Budget required to complete all chains

The results indicate that the optimal expansion strategy is contingent upon the available budget. Specifically, as shown in Figure 7, when the budget is set to 50% of the total required, Breadth-First Search (BFS) exhibits superior performance, achieving a higher R^2 value compared to Monte Carlo Tree Search (MCTS) and Depth-First Search (DFS). This can be attributed to BFS's methodology of exploring tables in closer proximity to the initial query within the join chain, which allows it to identify key tables when sufficient resources are available for broader expansion.

Conversely, under a more constrained budget (25% of the total, as depicted in Figure 8), the MCTS method surpasses both BFS and DFS in terms of R^2 accuracy. In this scenario, BFS's effectiveness is diminished as the limited budget restricts its ability to expand adequately to locate critical tables. DFS, which prioritizes depth, demonstrates greater performance variability, likely due to the arbitrary placement of crucial information within any given chain; its success is highly dependent on whether the deep exploration path happens to uncover these valuable nodes early. MCTS, by offering a more balanced approach between exploration breadth and depth, proves to be the most robust strategy when operating under stringent budgetary limitations.

Test Case	Data File Name	Target Column	Group By Columns	Using Kitana History	ex History
1	master.csv	suicides_no	Country, year	0.52	0.52
2	raw_data.csv	human_development_index	Country	0.75	0.78
3	Cost_of_Living_Index_by_Country_2024.csv	Groceries Index	Country	0.95	0.95
4	housing_geo_data.csv	median_house_value	latitude, longitude	0.99	0.33
5	house_details.csv	Price	Address	0.99	0.99
6	crime-rate-by-country-2023-base.csv	crimeIndex	country	0.9	0.9
7	quality_of_life.csv	stability	country	0.73	0.74
8	quality_of_life.csv	rights	country	0.85	0.85
9	quality_of_life.csv	health	country	0.66	0.66
10	quality_of_life.csv	safety	country	0.498	0.49
11	quality_of_life.csv	climate	country	0.34	0.33
12	quality_of_life.csv	costs	country	0.32	0.32
13	quality_of_life.csv	popularity	country	0.15	0.23
14	corruption.csv	corruption_index	country	0.99	0.99
15	corruption.csv	annual_income	country	0.95	0.95
16	Petrol Dataset June 20 2022	Daily Oil Consumption	country	0.9	0.73
17	most-polluted-countries.csv	pollution_2023	country	0.99	0.99
Average				0.73	0.69

Figure 9: Change in R^2 from Removing Kitana History

5.3 Ablation Studies

5.3.1 Kitana Query History. A main component of our theory is that the Kitana search history provides further context for the LLM to ground itself through in-context learning. To test this theory, we run the Agentic Selector pipeline with and without the Kitana history. We run the agent without budget constraints and select 5 tables to add for each query. Figure 9 show cases the results.

Overall, the R^2 value of the target improves slightly (0.045) when including the Kitana history. Drilling down more deeply, most queries are not changed when removing the history, suggesting that they were not utilizing the history. Further, in cases where the R^2 value was low to begin with, removing the Kitana history slightly improved the R^2 value (one case for popularity of the quality_of_life.csv dataset where the R^2 improved from 0.15 to 0.23), suggesting that when the prior query history did not improve the target history, the Kitana query history may provide more noise than signal, although further tests are needed given the small sample.

However, in a few cases (test case 4 and 16), the R^2 value dropped significantly, highlighting the importance of the query history to potentially guide the LLM to select the best tables.

5.3.2 Set Size for Selector Agent. An important parameter for the selector agent is the number of tables it evaluates at a time. The more tables evaluated at the same time (and hence a smaller number of sets), the higher potential chance that the LLM is not able to parse out the correct table. On the other hand, too many sets will increase the number of LLM calls and increase the chance of the LLM being overwhelmed by the larger context, as shown in [8] (although [3] challenges this claim). For the following test, we select 30 tables from the embedding model for test case 15, and alter the window size from 30 (1 set) to 6 (5 sets). We ask the LLM to select 5 final tables and do not include budget constraints. Figure 10 showcases the results

Unsurprisingly, the token cost increases as more sets. However, the R^2 value reaches its upper bound with only 2 sets, suggesting more sets is helpful, but there are diminishing returns as the number of sets increases. We performed a similar test but with 100 selected tables from the embedding model, with the results summarized in figure 11. We see similar results to the 30 initial table test, where adding further sets experiences marginal gains (or in some cases marginal losses). More interestingly, the overall average R^2 is lower when the selector agent starts with 100 tables vs 30, highlighting the importance of filtering the initial set of tables given that it seems

Number of Sets	R^2	Token Costs
1	0.94	2500
2	0.96	3094
3	0.96	3794
5	0.96	5099

Figure 10: R^2 from Changing Set Size - 30 Initial Tables

that large contexts hurt the model's performance (although this is model dependent).

Number of Sets	R^2	Token Costs
1	0.89	8454
2	0.9	8849
3	0.91	9586
4	0.87	10686
5	0.85	10393
10	0.9	14888

Figure 11: R^2 from Changing Set Size - 100 Initial Tables

6 DISCUSSION AND CONCLUSION

In conclusion, we explored how agentic systems can help find tables to enhance Kitana queries. We explored the no-hop case of finding a single table at a time where we find that our agentic approach using Kitana query histories increase performance vs the baseline by 24% and improve on the baseline embedding approach by 13%. Additionally we explored the multi-hop case, and demonstrate how an agentic system can find a sequence of tables to join to enhance a query that would have not have been possible with single table joins in Kitana. We show that the agentic system can find tables that enhance the original query with only exploring half or a quarter of all the potential join paths.

Overall, we demonstrate the feasibility of using agentic systems to find tables to improve Kitana queries. However, there are still several directions this project could go as the selector agent was still 12% below the upper bounded accuracy. These directions could include 1) creating more sophisticated selector agents by training a classification model based on prior Kitana histories, 2) creating a more sophisticated agentic system that dynamically decides the workflow structure depending on query needs (vs our fixed pipeline structure) and 3) optimizing LLM calls further by calling smaller, fine tuned agents or by optimizing prompts by using packages such as DSPy.

REFERENCES

[1] R. Castro Fernandez, E. Mansour, A. A. Qahtan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 989–1000, 2018.

[2] S.-A. Chen, L. Miculicich, J. M. Eisenschlos, Z. Wang, Z. Wang, Y. Chen, Y. Fujii, H.-T. Lin, C.-Y. Lee, and T. Pfister. Tablerag: Million-token table understanding with language models, 2024.
[3] Y. Chung, G. T. Kakkar, Y. Gan, B. Milne, and F. Ozcan. Is long context all you need? leveraging llm's extended context for nl2sql, 2025.
[4] Y. Dong, C. Xiao, T. Nozawa, M. Enomoto, and M. Oyamada. Deepjoin: Joinable table discovery with pre-trained language models. *arXiv preprint arXiv:2212.07588*, 2022.
[5] X. Fang, W. Xu, F. A. Tan, J. Zhang, Z. Hu, Y. Qi, S. Nickleach, D. Socolinsky, S. Sengamedu, and C. Faloutsos. Large language models(llms) on tabular data: Prediction, generation, and understanding – a survey, 2024.
[6] Y. Gorishniy, I. Rubachev, and A. Babenko. On embeddings for numerical features in tabular deep learning, 2023.
[7] Z. Huang and E. Wu. Cocoon: Semantic table profiling using large language models, 2024.
[8] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts, 2023.
[9] L. Patel, S. Jha, C. Guestrin, and M. Zaharia. Lotus: Enabling semantic queries with llms over tables of unstructured and structured data. *arXiv preprint arXiv:2407.11418*, 2024.
[10] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G. T. Kakkar, Y. Gan, A. Saberi, F. Ozcan, and S. O. Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql, 2024.
[11] Z. Qin, R. Jagerman, K. Hui, H. Zhuang, J. Wu, L. Yan, J. Shen, T. Liu, J. Liu, D. Metzler, X. Wang, and M. Bendersky. Large language models are effective text rankers with pairwise ranking prompting, 2024.
[12] P. Yin, G. Neubig, W. tau Yih, and S. Riedel. Tabert: Pretraining for joint understanding of textual and tabular data, 2020.
[13] H. Zhang, Y. Dong, C. Xiao, and M. Oyamada. Jellyfish: A large language model for data preprocessing, 2024.
[14] Z. Zhao and R. Castro Fernandez. Leva: Boosting machine learning performance with relational embedding data augmentation. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1504–1517, New York, NY, USA, 2022. Association for Computing Machinery.
[15] E. Zhu. *Search and Join Algorithms for Tables in Data Lakes*. University of Toronto (Canada), 2019.

APPENDIX

A.1 Additional Figures and Results

Each figure in this appendix complements the findings discussed in the main text.

- **Figure 12** shows how the LLM Enrich Agent performs across the different test cases and budget constraints.
- **Figure 13** shows how the Selector Agent performs across the different test cases and budget constraints.

					R^2 by Token Budget (B)				
Test Case	**Data File Name**	**Target Column**	**Group By Columns**	Baseline R^2	**B=2000**	**B=5000**	**B=10,000**	**B=15,000**	**B=20,000**
1	master.csv	suicides_no	Country, year	0.5200	0.5208	0.5225	0.5207	0.5208	0.5213
2	raw_data.csv	human_development_index	Country	0.7500	0.7467	0.7288	0.7235	0.7186	0.7348
3	Cost_of_Living_Index_by_Country_2024.csv	Groceries Index	Country	0.9400	0.9574	0.9587	0.9484	0.9567	0.9591
4	housing_geo_data.csv	median_house_value	latitude, longitude	0.4500	0.2823	0.2822	0.3379	0.3379	0.2822
5	house_details.csv	Price	Address	0.3400	0.9936	0.9936	0.9936	0.3441	0.3449
6	crime-rate-by-country-2023-base.csv	crimeIndex	country	0.3800	1.0000	1.0000	1.0000	1.0000	1.0000
7	quality_of_life.csv	stability	country	0.7200	0.7373	0.7345	0.7425	0.7425	0.7425
8	quality_of_life.csv	rights	country	0.8000	0.8499	0.8499	0.8526	0.8497	0.8526
9	quality_of_life.csv	health	country	0.6300	0.6616	0.7511	0.7511	0.8262	0.6595
10	quality_of_life.csv	safety	country	0.4700	0.4773	0.4773	0.4736	0.4926	0.5140
11	quality_of_life.csv	climate	country	0.2200	0.3659	0.3548	0.3548	0.3548	0.3548
12	quality_of_life.csv	costs	country	0.3200	0.5470	0.4642	0.4614	0.4544	0.4614
13	quality_of_life.csv	popularity	country	0.0700	0.0648	0.0529	0.1194	0.1194	0.1468
14	corruption.csv	corruption_index	country	0.8300	0.8527	0.8527	0.8527	0.8527	0.8524
15	corruption.csv	annual_income	country	0.8300	0.8389	0.9301	0.8389	0.8368	0.8376
16	Petrol Dataset June 20 2022	Daily Oil Consumption	country	0.1600	0.1669	0.1669	0.1669	0.1669	0.1669
17	most-polluted-countries.csv	pollution_2023	country	0.6700	0.6811	0.7351	0.6649	0.9947	0.6509
			Average:	0.5353	0.6320	0.6385	0.6355	0.6217	0.5930

Figure 12: LLM Enrich Agent Full Results

					R^2 by Token Budget (B)				
Test Case	**Data File Name**	**Target Column**	**Group By Columns**	Baseline R^2	**B=2000**	**B=5000**	**B=10,000**	**B=15,000**	**B=20,000**
1	master.csv	suicides_no	Country, year	0.5200	0.5238	0.5208	0.5207	0.5207	0.5207
2	raw_data.csv	human_development_index	Country	0.7500	0.7543	0.7371	0.7474	0.7385	0.7492
3	Cost_of_Living_Index_by_Country_2024.csv	Groceries Index	Country	0.9400	0.9580	0.9557	0.9497	0.9588	0.9621
4	housing_geo_data.csv	median_house_value	latitude, longitude	0.4500	0.3379	0.3379	0.3379	0.3379	0.4351
5	house_details.csv	Price	Address	0.3400	0.3441	0.3441	0.3441	0.3441	0.3441
6	crime-rate-by-country-2023-base.csv	crimeIndex	country	0.3800	1.0000	1.0000	1.0000	1.0000	1.0000
7	quality_of_life.csv	stability	country	0.7200	0.7425	0.7425	0.7425	0.7425	0.7425
8	quality_of_life.csv	rights	country	0.8000	0.8305	0.8497	0.8497	0.8497	0.8497
9	quality_of_life.csv	health	country	0.6300	0.8187	0.8187	0.8187	0.8250	0.8187
10	quality_of_life.csv	safety	country	0.4700	0.4769	0.4773	0.4769	0.4769	0.4769
11	quality_of_life.csv	climate	country	0.2200	0.6207	0.6207	0.6207	0.6207	0.6207
12	quality_of_life.csv	costs	country	0.3200	0.5470	0.5313	0.5470	0.5470	0.5470
13	quality_of_life.csv	popularity	country	0.0700	0.0648	0.0648	0.1324	0.0648	0.1324
14	corruption.csv	corruption_index	country	0.8300	0.8527	0.8527	0.8527	0.8527	0.8496
15	corruption.csv	annual_income	country	0.8300	0.9301	0.9301	0.9301	0.9301	0.9301
16	Petrol Dataset June 20 2022	Daily Oil Consumption	country	0.1600	0.8819	0.9693	0.9242	0.9656	0.9411
17	most-polluted-countries.csv	pollution_2023	country	0.6700	0.7118	0.6501	0.6746	0.6325	0.6486
			Average:	0.5353	0.6703	0.6707	0.6747	0.6457	0.6805

Figure 13: Selector Agent Full Results