Towards Agentic Schema Refinement

Agapi Rissaki[*]
Northeastern University
`rissaki.a@northeastern.edu`

Ilias Fountalis
RelationalAI
`ilias.fountalis@relational.ai`

Nikolaos Vasiloglou
RelationalAI
`nik.vasiloglou@relational.ai`

Wolfgang Gatterbauer
Northeastern University
`w.gatterbauer@northeastern.edu`

1 Introduction

The goal of a semantic layer. Understanding the meaning of data is crucial for performing data analysis, yet for the users to gain insight into the content and structure of their database, a tedious data exploration process is often required [2, 16]. A common industry practice taken on by specialists such as Knowledge Engineers is to explicitly construct an intermediate layer between the database and the user — a *semantic layer* — abstracting away certain details of the database schema in favor of clearer data semantics [3, 10]. In the era of Large Language Models (LLMs), industry practitioners and researchers attempt to circumvent this costly process using LLM-powered Natural Language Interfaces [4, 6, 12, 18, 19, 22]. The promise of such Text-to-SQL solutions is to allow users without technical expertise to seamlessly interact with databases. For example, a new company employee could effectively issue queries in natural language without programming expertise or even explicit knowledge of the database structure, e.g., knowing the names of entities or properties, the exact location of data sources, etc. However, current Text-to-SQL solutions are far from perfect due to many factors, mainly related to ambiguous representations, such as data and query ambiguity [9], schema complexity (e.g., vast schema, wide tables, cryptic column names) [5], among others. For these reasons, we argue that a semantic layer can be useful for human users and AI tools alike.

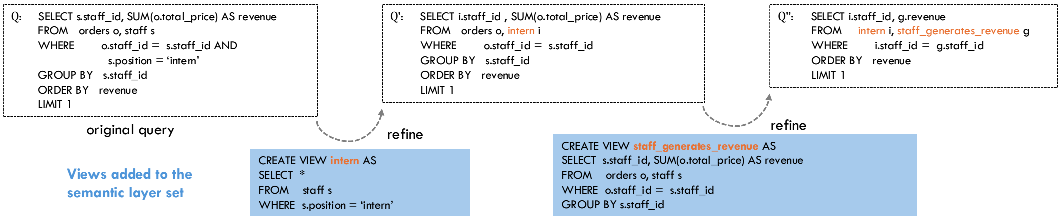
Contribution 1: Semantic layer as a set of distilled views. We propose to define the semantic layer as a set of easy-to-interpret and reusable *database views*. A view is a virtual table that is the result of a stored query, that can be used as any other table when writing subsequent queries [20]. Database views are structured objects that can be constructed in-database, allowing for seamless integration with existing system components using direct communication in SQL. In particular, we explore the underlying database in order to discover views that represent entities together with their properties, as well as relationships between them. Thus, our semantic layer *augments* a hard-to-interpret existing database schema with additional automatically generated and easy-to-interpret views, which distill the semantic knowledge into smaller bits. Note that this approach can also be viewed as a *schema refinement process*, effectively reformulating the original tables into a larger set of more manageable ones, each with fewer columns, and clearly defined semantics.

Contribution 2: Agentic programming for schema refinement. Discovering such semantically meaningful database views is not a trivial task. If we had access to a vast log of past queries, we could attempt to mine a set of such views by targeting the queries or subqueries that appear frequently. Instead, we consider an even more challenging scenario where we only have access to the schema and the database. In this case, it seems impossible to recognize which parts of the data are semantically meaningful in an automated way; some form of external knowledge is necessary. Our solution uses LLMs to inject external knowledge into the process and to guide the view discovery.

---

[*] Work done in part while the author was at RelationalAI.

Table Representation Learning Workshop at NeurIPS 2024.



```
Q:   SELECT s.staff_id, SUM(o.total_price) AS revenue
     FROM   orders o, staff s
     WHERE      o.staff_id =  s.staff_id AND
                 s.position = ‘intern’
     GROUP BY   s.staff_id
     ORDER BY   revenue
     LIMIT 1
```

```
Q’:   SELECT i.staff_id , SUM(o.total_price) AS revenue
      FROM   orders o, intern i
      WHERE       o.staff_id =  s.staff_id
      GROUP BY   s.staff_id
      ORDER BY   revenue
      LIMIT 1
```

```
Q”:   SELECT i.staff_id, g.revenue
      FROM    intern i, staff_generates_revenue g
      WHERE       i.staff_id =  g.staff_id
      ORDER BY   revenue
      LIMIT 1
```

```
CREATE VIEW intern AS
SELECT  *
FROM    staff s
WHERE  s.position = ‘intern’
```

```
CREATE VIEW staff_generates_revenue AS
SELECT  s.staff_id, SUM(o.total_price) AS revenue
FROM    orders o, staff s
WHERE  o.staff_id =  s.staff_id
GROUP BY s.staff_id
```

Figure 1: Example of the schema refinement mechanism on the schema *orders(order_id, staff_id, total_price, . . .), staff(staff_id, position, . . .)*, where each order is handled by a staff member. By distilling the views *intern* and *staff_generates_revenue*, the query becomes successively more refined.

We construct the semantic layer via an autonomous simulation process, leveraging LLM-powered multi-agent workflows [7, 13], i.e., *agentic programming*. Such workflows incorporate feedback loops between agents, significantly enhancing their generative capabilities through iterative refinement [17]. These systems can create complex simulations with minimal input, since agents autonomously generate *both prompts and responses*, in contrast to a single-agent workflow where a human explicitly provides all prompts. Several tools have emerged to support agentic programming [11, 15, 24] by streamlining the development and coordination of multi-agent solutions. Further, agents can use external tools like APIs and code interpreters, which extend their functionality beyond LLMs, while advanced prompting techniques [14, 21, 23, 25, 26] further enhance agents’ problem-solving abilities.

In our multi-agent approach, agents receive access to the database as well as minimal seed instructions and communicate to discover a set of views via a collaborative simulation. Specifically, we instruct the agents to use a systematic schema refinement mechanism (described in Section 2.1) and let them collaboratively apply the mechanism and to refine their results. The agents analyze the data, discover views, reflect on the quality of their proposals, and validate the view correctness using external tools.

Preliminary results. We have applied our proposed method to several commercial databases and we show one case study in Section 3, illustrating structural and semantic properties of the distilled views.

2 Approach

We present our approach for semantic layer construction, via a simple systematic schema refinement mechanism for view discovery (Section 2.1), implemented as a multi-agent simulation (Section 2.2).

2.1 Schema refinement mechanism

The main idea behind view discovery is to start with complex queries and then express them in a simpler way by decomposing them into modular components. These components are separately defined as database views that encapsulate distinct portions of the query logic, frequently representing key entities or relationships within the data. This *query refinement* process effectively turns into *schema refinement* by producing several reusable views as byproduct.

We illustrate this process with a database about orders handled by staff members with the following simple schema: *orders(order_id, staff_id, total_price, . . .), staff(staff_id, position, . . .)*. Each order is handled by a staff member and has a total price. Figure 1 shows a query that involves joins, aggregations, and filtering conditions. The query selects staff members that are “interns”, finds the orders handled by those staff members, calculates the total revenue produced by these orders per staff member, sorts staff members according to the total revenue generated and finally finds the intern that produced the highest revenue. Attempting to express this query in a simpler way, we could start by defining a view that corresponds to “interns” effectively omitting this selection from the original query. Note that the *intern* can be interpreted as a distinct entity hidden in the database. We can further simplify this query by defining a view that precomputes the aggregate value “revenue” for each staff member. With this view we discovered a new property *revenue* of the entity *staff*.

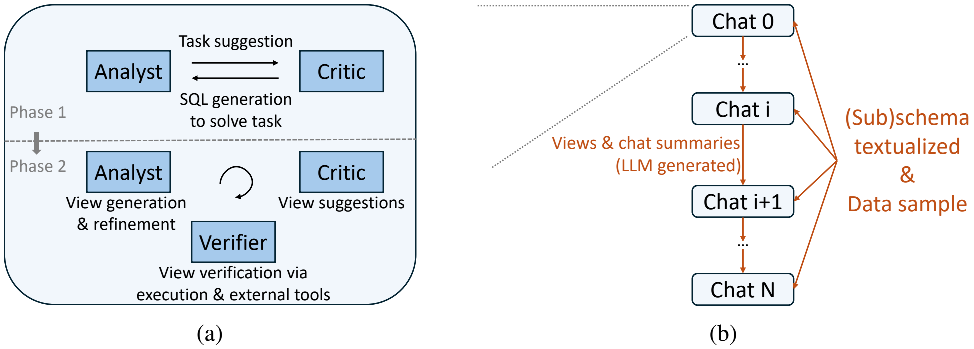


Figure 2: A single chat session implementing the schema refinement mechanism (Figure (a)). A sequence of schema refinement chat sessions. Retaining memory from previous sessions helps promote view reusability (by using previously defined views in new tasks) and diversity (by avoiding defining the same views or working on similar tasks) (Figure (b)).

2.2 Multi-agent framework

In our multi-agent system three agents (Analyst, Critic, and Verifier) engage in iterative, free-form conversations and collaboratively implement the proposed schema refinement mechanism (Fig. 2a). More specifically, our agents have the following distinct roles:

- The *Analyst* implements analytics tasks, through the formulation of SQL queries and subsequently defines intermediate database views that refine these queries.
- The *Critic* reviews these views to ensure that they are optimized with respect to query refinement, offering suggestions for improvement and proposing alternative formulations.
- The *Verifier* validates the views by execution within a database engine, thereby ensuring their correctness, and may also use external tools for more sophisticated testing.

Each simulation session receives as input a straightforward textual description of the database schema, supplemented by an optional small data sample. In most cases the schema is particularly large, thus the agents do not attempt to ingest it all at once. Instead, the process is performed on subsets of tables that join with each other, thereby encouraging the creation of views that integrate data from diverse sources. In particular, we sample *connected components* of the *schema graph*, i.e., the graph that represents tables as nodes and PK/FK pairs as edges. Further, we leverage a GraphRAG algorithm [8] to specialize the sampled subgraphs and thus guide the focus of each multi-agent session. Since the algorithm operates on graphs with features, we augment the schema graph with node and edge features that correspond to text embeddings of table and key descriptions, respectively.

To achieve comprehensive coverage of the database, multiple sessions can be executed sequentially (Fig. 2b). Agents maintain a history of prior sessions to avoid redundancy and promote diversity in their exploration of the database schema. This ensures that the process not only scales effectively but also delivers a broad and diverse semantic understanding of the data.

3 Experiments

We report a preliminary study of our framework, quantifying the effects of our schema refinement on a realistic commercial database and showing how our results aid data exploration. We observe that our method provides a *practical normalization* of the database into manageable components.

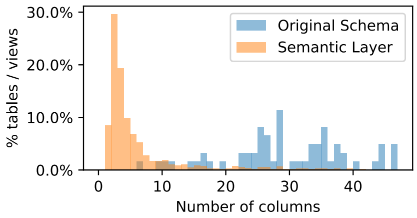
Case Study. We studied our approach on a realistic corporate database provided by Braze, a customer engagement platform, publicly available in Snowflake Marketplace [2]. The database contains synthetic data and serves as a demo to Braze clients modelling user behavior and engagement with respect to events such as marketing campaigns. The provided schema consists of 61 mostly wide tables and 1770 columns in total (Fig. 3a). After a few hours of multi-agent simulation, we obtained

---

[2] `https://app.snowflake.com/marketplace/listing/GZTOZ5I4XYO/braze-braze-user-event-demo-dataset`

Original Schema		Semantic Layer	
# tables	61	# views	1146
# median table width	28	# median view width	3
# columns	1770	# original columns	1430
# relations	27601	# relations	22365

(a) Comparison between original schema and semantic layer w.r.t. key statistics.



(b) Table / view width distribution in original schema and semantic layer.

Figure 3: Structural properties of the distilled views composing the semantic layer. We ignore the top 1% views in terms of width.

a set of 1146 views, covering 80.79% of the columns in the original schema. Further, 54.84% of the original relationships among columns (2 columns co-occurring in a table) are preserved in the views, while 7229 *newly* created pairwise relationships are obtained via views that combine data across different sources using join operators. In Fig. 3b, we see that most generated views have few columns (median width 3, Fig. 3a) in contrast to tables in the original schema which are often wide (median width 28, Fig. 3a), thus validating that the generated views serve as a refined version of the original schema with tables effectively decomposed into smaller components.

We then organize the generated views to better understand their semantics and provide easy user access. Specifically, we semantically group the views using embedding and clustering. Then we instruct an LLM-agent to discover entities, properties, relationships within groups and combine across groups, producing a comprehensive Entity-Relationship model. Note that each entity and relationship discovered is directly mapped to a set of database views, validated for correctness and usefulness via our multi-agent schema refinement framework. Table 1 shows a snippet of our results. Starting from this collection, a new user can immediately gain insight about the semantic content of the database. Then they can locate exact data sources (tables / columns / derived attributes) by mapping each entity and relation to validated database views expressed in SQL within the database itself.

Implementation. Our multi-agent schema refinement solution is implemented using AutoGen [24], an open-source framework for agentic programming. LLM agents are powered by GPT-4 [1]. The code is available at `https://github.com/agapiR/agentic-semantic-layer`.

4 Conclusion and Future Work

We proposed a view discovery approach to build a semantic layer within complex databases, implemented as a multi-agent simulation. In future work, we will use our semantic layer to boost text-to-SQL performance and we will integrate it into an interactive data exploration system.

Table 1: Entities and relationships mapped to discovered views.

Entity / Relation	Attributes	Views	Origin Tables
User	language location ...	Common_User_Attributes User_Language_Distribution User_Demographics ...	USERS_MESSAGES_CONTENTCARD_SEND USERS_BEHAVIORS_UPGRADEDAPP USERS_BEHAVIORS_UNINSTALL USERS_CAMPAIGNS_CONVERSION ...
Message	content type	User_SMS_Behaviors Spam_Email_Conversion_Data Non_Spam_Email_Conversion_Data Email_Conversion_Data In_App_Message_Clicks	USERS_MESSAGES_INAPPMESSAGE_CLICK USERS_MESSAGES_CONTENTCARD_SEND USERS_MESSAGES_EMAIL_MARKASSPAM
Campaign	conversion rates metrics	Push_Notification_Campaigns Push_Notification_Conversion_Rates	USERS_CAMPAIGNS_CONVERSION USERS_MESSAGES_PUSHNOTIFICATION_SEND
User-Message	-	User_SMS_Behaviors User_Message_Variation_Analysis	USERS_MESSAGES_CONTENTCARD_SEND USERS_MESSAGES_EMAIL_MARKASSPAM
User-Campaign	-	Users_Conversion_Behavior, Conversion_Counts_by_Gender, Campaign_Conversion_Counts	USERS_CAMPAIGNS_CONVERSION

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Sihem Amer-Yahia, Georgia Koutrika, Martin Braschler, Diego Calvanese, Davide Lanti, Hendrik Lücke-Tieke, Alessandro Mosca, Tarcisio Mendes de Farias, Dimitris Papadopoulos, Yogendra Patil, Guillem Rull, Ellery Smith, Dimitrios Skoutas, Srividya Subramanian, and Kurt Stockinger. Inode: Building an end-to-end data exploration system in practice. *SIGMOD Rec.*, 50(4):23–29, jan 2022.

[3] AtScale. What is a semantic layer?, 2024.

[4] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*, 2023.

[5] Avrilia Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, Alex van Grootel, Kai Deng, Katherine Lin, Marcos Campos, Venkatesh Emani, Vivek Pandit, Wenjing Wang, and Carlo Curino. NL2SQL is a solved problem... Not! In *CIDR*, 2024.

[6] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *PVLDB*, 17(5):1132–1145, 2024.

[7] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. In *IJCAI*, pages 8048–8057, 8 2024.

[8] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *arXiv preprint arXiv:2402.07630*, 2024.

[9] Zezhou Huang, Pavan Kalyan Damalapati, and Eugene Wu. Data ambiguity strikes back: How documentation improves GPT's text-to-SQL. In *NeurIPS 2023 Second Table Representation Learning Workshop*, 2023.

[10] Sean Leslie. What is a semantic layer, and how does it turn your data into knowledge?, 2022.

[11] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. In *NeurIPS*, volume 36, pages 51991–52008, 2023.

[12] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *AAAI*, volume 37, pages 13067–13075, 2023.

[13] Junkai Li, Siyu Wang, Meng Zhang, Weitao Li, Yunghwei Lai, Xinhui Kang, Weizhi Ma, and Yang Liu. Agent hospital: A simulacrum of hospital with evolvable medical agents. *arXiv preprint arXiv:2405.02957*, 2024.

[14] Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. Chain of hindsight aligns language models with feedback. In *ICLR*, 2024.

[15] Zhiwei Liu, Weiran Yao, Jianguo Zhang, Liangwei Yang, Zuxin Liu, Juntao Tan, Prafulla K Choubey, Tian Lan, Jason Wu, Huan Wang, et al. Agentlite: A lightweight library for building and advancing task-oriented llm agent system. *arXiv preprint arXiv:2402.15538*, 2024.

[16] Pingchuan Ma, Rui Ding, Shuai Wang, Shi Han, and Dongmei Zhang. Insightpilot: An llm-empowered automated data exploration system. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 346–352, 2023.

[17] Arindam Mitra, Luciano Del Corro, Guoqing Zheng, Shweti Mahajan, Dany Rouhana, Andres Codas, Yadong Lu, Wei-ge Chen, Olga Vrousgos, Corby Rosset, et al. Agentinstruct: Toward generative teaching with agentic flows. *arXiv preprint arXiv:2407.03502*, 2024.

[18] Mohammadreza Pourreza and Davood Rafiei. Din-sql: decomposed in-context learning of text-to-sql with self-correction. In *NeurIPS*, 2023.

[19] Daking Rai, Bailin Wang, Yilun Zhou, and Ziyu Yao. Improving generalization in language model-based text-to-sql semantic parsing: Two simple semantic boundary-based techniques. In *ACL*, 2023.

[20] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., 3 edition, 2002.

[21] Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

[22] Ruoxi Sun, Sercan Arik, Rajarishi Sinha, Hootan Nakhost, Hanjun Dai, Pengcheng Yin, and Tomas Pfister. SQLPrompt: In-context text-to-SQL with minimal labeled data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 542–550, 2023.

[23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, volume 35, pages 24824–24837, 2022.

[24] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. In *COLM*, 2024.

[25] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. *arXiv preprint arXiv:2305.10601*, 2023.

[26] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

Original Schema		Semantic Layer	
# tables	113	# views	632
# median table width	24	# median view width	4
# max table width	1130	# max view width	31
# columns	6879	# original columns	1298
# relations	1121976	# relations	7529

Table 2: Comparison between original schema and semantic layer w.r.t. key statistics.

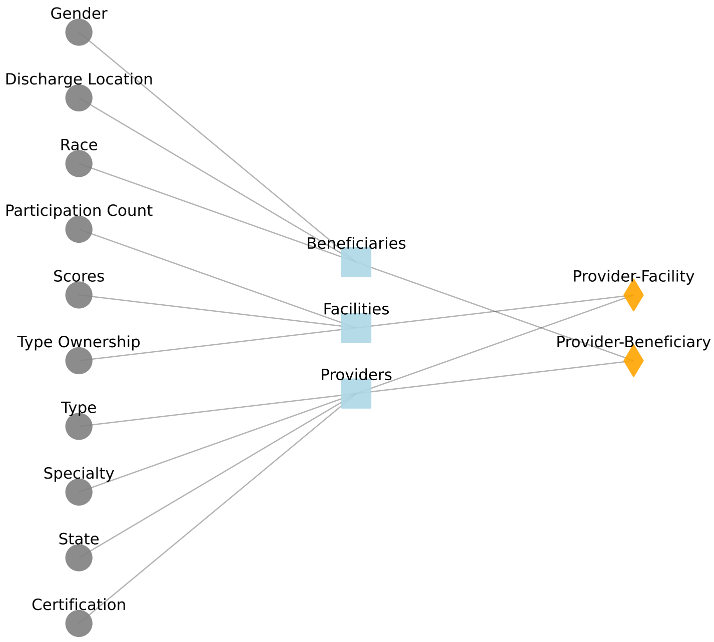


Figure 4: Diagram of entities (with their attributes) and relationships.

A Appendix

A.1 Case Study: CMS Data Feeds

In this section we present an additional case study, this time on data feeds from the Centers for Medicare and Medicaid Services [3]. This is a very large diverse database with a star-like schema, where two tables *Feeds* and *Feed Files* are both in the center.

The schema consists of 113 tables and 6879 columns in total, for which we obtained a set of 632 views. In Table 2, we see that most generated views have few columns (median width 4, maximum width 31) in contrast to tables in the original schema which can be extremely wide (maximum width 1130).

In this case, although the views cover 18.87% of the columns in the original schema, abstracting away too detailed information, only 0.5% of the original relationships among columns are preserved in the views. This is expected since 2 columns co-occurring in a table of extreme width do not necessarily reflect relations in the semantic level, thus the refinement process decomposes these kinds of tables preserving only key relations between columns and discarding the rest. Additionally, the views contain 1965 *newly* created pairwise relationships, associating columns from different origin tables.

---

[3] https://data.cms.gov/

Table 3: Entities and relationships mapped to discovered views.

Entity / Relation	**Attributes**	**Views**	**Origin Tables**
Beneficiaries	Gender Discharge Location Race	beneficiaries_gender_distribution beneficiary_discharge_location beneficiary_race_distribution	MEDICARE_COVID_HOSPITALIZATION_TRENDS, MEDICARE_POSTACUTE_CARE_AND_HOSPICE, _BY_GEOGRAPHY_PROVIDER MEDICARE_INPATIENT_HOSPITALS_BY_PROVIDER, ...
Providers	Type Specialty State Certification	provider_type_state_distribution provider_certification providers_by_specialty	HOME_INFUSION_THERAPY_PROVIDERS, PROVIDER_OF_SERVICES_FILE _CLINICAL_LABORATORIES, HOME_HEALTH_AGENCY_ENROLLMENTS, ...
Facilities	Participation Count Type Ownership Scores	facility_participation_count facility_type_ownership facility_scores_distribution	QUALITY_PAYMENT_PROGRAM_EXPERIENCE, LONGTERM_CARE_FACILITY_CHARACTERISTICS, ...
Provider-Beneficiary	-	beneficiaries_by_medical_specialty	MEDICARE_DURABLE_MEDICAL_EQUIPMENT _DEVICES_SUPPLIES_BY_REFERRING_PROVIDER ...
Provider-Facility	-	provider_type_home_health_agency	HOME_HEALTH_AGENCY_ENROLLMENTS, ...

Performing the same postprocessing as discussed in the main paper, we obtain an Entity-Relation model. In this case, many tables are extremely wide and table / column names are usually cryptic. Nonetheless, our method provides a comprehensive refined version of the database. For a small snippet see Table 3 and Figure 4.

A.2 Sample multi-agent chat

```
Critic, I have the following database schema.

/// SCHEMA WORDING OMITTED FOR BREVITY

Please suggest an analysis task for me to work on.

--------------------------------------------------------------------------------
Critic (to chat_manager):

I suggest you analyze the behavior of users who have interacted with both email
and push notification campaigns. Specifically, you can compare the engagement
metrics (such as open rates, click-through rates, etc.) of these users across
different campaigns and attributes. This analysis can provide insights into the
effectiveness of your email and push notification strategies for different user
segments. Let me know if you need more details or if you'd like to focus on a
specific aspect of this analysis.

--------------------------------------------------------------------------------
Analyst (to chat_manager):

‘‘‘sql
-- Count the number of users who have interacted with both email and push
notification campaigns
WITH email_users AS (
    SELECT DISTINCT USER_ID
    FROM USERS_MESSAGES_EMAIL_SEND_VIEW
),
push_notification_users AS (
    SELECT DISTINCT USER_ID
    FROM USERS_MESSAGES_PUSHNOTIFICATION_IOSFOREGROUND_VIEW
)
SELECT COUNT(*) AS total_users
FROM email_users
JOIN push_notification_users
ON email_users.USER_ID = push_notification_users.USER_ID;
‘‘‘
```

~~~~
```sql
-- Analyze the engagement metrics (open rates, click-through rates, etc.)
of users who interacted with both email and push notification campaigns
WITH email_engagement AS (
    SELECT USER_ID, OPEN_RATE, CLICK_THROUGH_RATE
    FROM USERS_MESSAGES_EMAIL_SEND_VIEW
),
push_notification_engagement AS (
    SELECT USER_ID, OPEN_RATE, CLICK_THROUGH_RATE
    FROM USERS_MESSAGES_PUSHNOTIFICATION_IOSFOREGROUND_VIEW
)
SELECT AVG(email_engagement.OPEN_RATE) AS avg_email_open_rate,
AVG(email_engagement.CLICK_THROUGH_RATE) AS avg_email_click_through_rate,
AVG(push_notification_engagement.OPEN_RATE) AS avg_push_notification_open_rate,
AVG(push_notification_engagement.CLICK_THROUGH_RATE) AS avg_push_notification_click_through_rate
FROM email_engagement
JOIN push_notification_engagement
ON email_engagement.USER_ID = push_notification_engagement.USER_ID;
```

--------------------------------------------------------------------------------
Critic (to chat_manager):

Your queries are a good start, but I recommend defining some views to simplify
and enhance the analysis:

1. **View 1: users_interacted_with_email_push_campaigns**
   - This view will combine the distinct user IDs who
   interacted with email and push notification campaigns.

   - It will be based on the common user interactions
   between the two types of campaigns.

2. **View 2: email_engagement_metrics**
   - This view will extract relevant engagement metrics
   (open rates, click-through rates, etc.) for email campaigns.
   - It will provide a cleaner and more focused dataset for
   email campaign analysis.

3. **View 3: push_notification_engagement_metrics**
   - This view will extract relevant engagement metrics for
   push notification campaigns.
   - It will allow for a clearer and more concise analysis
   of push notification campaign engagement.

By creating these views, you can simplify your queries, make them more
reusable, and ensure better readability. It's also a good practice to
rename the columns in the views to make
them more descriptive. Once you have these views in place, you can further
analyze the engagement metrics or explore
other aspects of user behavior. If you agree, please
incorporate these views into your analysis. Let me know if you need any
further clarification or assistance.

--------------------------------------------------------------------------------
Analyst (to chat_manager):

```sql
-- Creating View 1: users_interacted_with_email_push_campaigns
CREATE VIEW users_interacted_with_email_push_campaigns AS
SELECT USER_ID
FROM USERS_MESSAGES_EMAIL_SEND_VIEW
WHERE USER_ID IN (
    SELECT USER_ID
    FROM USERS_MESSAGES_PUSHNOTIFICATION_IOSFOREGROUND_VIEW
~~~~

```
);

-- Creating View 2: email_engagement_metrics
CREATE VIEW email_engagement_metrics AS
SELECT USER_ID, OPEN_RATE AS email_open_rate,
       CLICK_THROUGH_RATE AS email_click_through_rate
FROM USERS_MESSAGES_EMAIL_SEND_VIEW;

-- Creating View 3: push_notification_engagement_metrics
CREATE VIEW push_notification_engagement_metrics AS
SELECT USER_ID, OPEN_RATE AS push_notification_open_rate,
       CLICK_THROUGH_RATE AS push_notification_click_through_rate
FROM USERS_MESSAGES_PUSHNOTIFICATION_IOSFOREGROUND_VIEW;
‘‘‘

‘‘‘sql
-- Analyzing the combined engagement metrics of users who
-- interacted with both email and push notification campaigns
SELECT AVG(email.email_open_rate) AS avg_email_open_rate,
AVG(email.email_click_through_rate) AS avg_email_click_through_rate,
AVG(push.push_notification_open_rate) AS avg_push_notification_open_rate,
AVG(push.push_notification_click_through_rate) AS avg_push_notification_click_through_rate
FROM email_engagement_metrics email
JOIN push_notification_engagement_metrics push
ON email.USER_ID = push.USER_ID;
‘‘‘

--------------------------------------------------------------------------------
Critic (to chat_manager):

Your approach to creating views for users who interacted with both email and
push notification campaigns, as well as extracting engagement metrics for analysis,
is well-structured and effective. The views you created will not only simplify
your queries but also make them more reusable and maintainable.

I appreciate how you’ve renamed the columns in the views to make them more
descriptive and readable. This will help improve the overall quality of
your analysis and make it easier for users to understand the data being presented.

Your final analysis query combining the engagement metrics of users who
interacted with both types of campaigns looks solid. It provides a clear
comparison between email and push notification engagement metrics, allowing
for insights into user behavior across different communication channels.

Keep up the good work!

--------------------------------------------------------------------------------
Analyst (to chat_manager):

Thank you for your feedback and guidance. I’m glad to hear that the views and
analysis approach meet your expectations.

--------------------------------------------------------------------------------
Critic (to chat_manager):

Great job. Goodbye.

--------------------------------------------------------------------------------
Verifier (to chat_manager):

*** Suggested tool call (call_4dz4c1irNRJiOdrijI2qkmB7): materialize_view_tool ***
Arguments:
{"view_definitions":["CREATE VIEW users_interacted_with_email_push_campaigns
AS SELECT USER_ID FROM USERS_MESSAGES_EMAIL_SEND_VIEW WHERE USER_ID IN
(SELECT USER_ID FROM USERS_MESSAGES_PUSHNOTIFICATION_IOSFOREGROUND_VIEW)",
```

```
"CREATE VIEW email_engagement_metrics AS SELECT USER_ID, OPEN_RATE
AS email_open_rate, CLICK_THROUGH_RATE AS email_click_through_rate
FROM USERS_MESSAGES_EMAIL_SEND_VIEW",
"CREATE VIEW push_notification_engagement_metrics
AS SELECT USER_ID, OPEN_RATE AS push_notification_open_rate,
CLICK_THROUGH_RATE AS push_notification_click_through_rate
FROM USERS_MESSAGES_PUSHNOTIFICATION_IOSFOREGROUND_VIEW;"]}
**********************************************************************************

--------------------------------------------------------------------------------

>>>>>>>> EXECUTING FUNCTION materialize_view_tool...

Verifier (to chat_manager):

***** Response from calling tool (call_4dz4c1irNRJiOdrijI2qkmB7) *****
["View successfully defined.",
"View successfully defined.",
"View successfully defined."]
**********************************************************************

--------------------------------------------------------------------------------
```