Information Systems 142 (2026) 102787

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Integrating AI and the multidimensional model: The Cube+AI framework

Houssam Bazza [a], Sandro Bimonte [b], Stefano Rizzi [c,*], Sana Sellami [d], Hassan Badir [e]

[a] LSIA, Moroccan School of Engineering Sciences, Tangier, Morocco
[b] TSCF, INRAE - University of Clermont Auvergne, Aubière, France
[c] DISI, University of Bologna, Bologna, Italy
[d] LIS, Aix Marseille University, Marseille, France
[e] IDS, Abdelmalek Essaadi University, Tangier, Morocco

ARTICLE INFO

Recommended by Dennis Shasha

Dataset link: Query workload and prompts (Original data)

Keywords:
LLM
Text-to-SQL
Multimodal databases
Multidimensional model
AI-integrated DBMSs

ABSTRACT

The integration of AI models into DBMSs has recently been proposed and implemented in tools such as PostgresAI. However, these solutions do not take full advantage of the multidimensional vision of information typically adopted in complex analytical tasks. At the same time, AI offers a chance to reconsider and improve analytical tasks based on the multidimensional model. In this work, we propose *Cube+AI*, a fresh vision of the multidimensional model aimed at extending the expressive power of cube queries to include semantically rich and multimodal operations on free text and images in addition to the analysis of categorical and numerical data. Cube+AI supports crossed semantic searches between text and images for filtering and grouping; moreover, it significantly extends aggregation by operating on text and images rather than on numeric data only. Thus, while the multidimensional model provides the structured context necessary to enhance AI execution, AI models expand the multidimensional paradigm by enabling the analysis and aggregation of unstructured data. The Cube+AI framework includes (i) a formal extension of the multidimensional model, which supports the dynamical definition of virtual categorical levels derived at query time from other levels via AI, as well as (ii) a text-to-SQL method where an LLM is leveraged to translate natural language queries into SQL. Specifically, we propose an implementation that relies on PostgresAI as a DBMS and on the Gemini LLM as a natural language querying interface. The paper is completed by the discussion of a set of experiments made with a sample workload of queries and by some robustness tests.

1. Introduction

Artificial Intelligence (AI) has made significant strides in recent years, generating and comprehending natural language text and visuals with previously unheard-of capabilities. Thanks to these developments, decision-makers can now tackle complex tasks requiring human intuition and observation, going beyond traditional analysis [1]. Following this trend, several academic and industrial works have recently proposed the integration of AI models into Database Management Systems (DBMSs) leading, for instance, to relational DBMSs that natively support AI methods (e.g., PostgresAI, https://postgres.ai/) and to vector databases [2]. These AI-integrated DBMSs (AI-DBMSs) optimize the deployment of AI models by facilitating in-place execution, thereby eliminating the latency and overhead of data movement to external cloud environments while preserving established security and access control policies. Furthermore, the relational paradigm ensures that AI models operate on consistent, high-integrity data. This approach bypasses the need for traditional data-flattening processes, often achieved through complex integration tools, which are tedious, time-consuming, and prone to error. In a relational AI-DBMS, static flattened tables are replaced by dynamic SQL queries that join, filter, and feed structured data directly into AI models, ensuring both logical consistency and operational efficiency.

Despite these advancements, contemporary AI models integrated within DBMSs do not take full advantage of the multidimensional vision of information typically adopted in complex analytical tasks. Historically, multidimensional modeling has remained a cornerstone of Business Intelligence, enabling the use of statistical indicators to analyze huge quantities of numerical data stored in multidimensional cubes via the On-Line Analytical Processing (OLAP) paradigm [3]. The multidimensional model facilitates the analysis of numerical measures across diverse dimensions (analysis axes) and varying levels of granularity (hierarchies). While traditionally implemented within OLAP systems, cubes have also been successfully adapted for real-time streaming environments – such as Stream Data Warehouses – and conventional

* Correspondence to: DISI, University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy.
E-mail addresses: houssam.bazza@etu.uae.ac.ma (H. Bazza), sandro.bimonte@inrae.fr (S. Bimonte), stefano.rizzi@unibo.it (S. Rizzi), sana.sellami@lis-lab.fr (S. Sellami), badir.hassan@uae.ac.ma (H. Badir).

https://doi.org/10.1016/j.is.2026.102787
Received 25 June 2026; Received in revised form 23 July 2026; Accepted 23 July 2026
Available online 29 July 2026

static reporting [4]. Bridging the gap between AI execution and this structured multidimensional view presents a significant opportunity to enhance the depth and context of AI.

We believe AI offers a significant chance to reconsider and improve analytical tasks based on the multidimensional model. Indeed, while there has been some research on combining textual and picture data (e.g., [5,6]), multidimensional cubes are normally used to store and query structured data, mainly categorical attributes and numerical measures. We asserted in [7] that the emergence of AI models creates a new type of multidimensional cubes that incorporates unstructured data – like text and images – into the analytical processes. Based on that vision, in this work we propose *Cube+AI*, a fresh vision of multidimensional cubes that attempts to extend the expressive power of cube queries to include semantically rich operations on free text (e.g., "*Classify restaurant textual reviews into 'good' and 'bad' and group revenues based on this classification*") and images (e.g., "*Classify pictures of restaurants and count the orders made in restaurants with a terrace*") in addition to the analysis of categorical and numerical data. Because it allows for cross-semantic searches between text and images for filtering and grouping, our method can be categorized as *multimodal* [8]. Interestingly, it also expands *aggregation* by working with text (e.g., "*Create a summary of a set of reviews*") and images (e.g., "*Create a representative image of a set of dish pictures*") instead of just numeric data, with the help of generative models. Finally, the integration of AI also enables additional knowledge to be incorporated in analyses (e.g., "*Given a restaurant dimension with a level* country*, group orders by restaurant continent, with the continent of each country determined by AI*"). Therefore, in Cube+AI, *while multidimensional cubes provide the structured depth and context necessary to enhance AI execution, AI models simultaneously expand the multidimensional paradigm by enabling the analysis and aggregation of unstructured data*.

The broad querying expressiveness implied by Cube+AI calls for an equally effective querying interface. In this regard, recent advancements in data management have witnessed a significant surge in the integration of Large Language Models (LLMs) to facilitate natural language interfaces to databases, primarily through *text-to-SQL* translation [9]. This is particularly beneficial to AI-DBMSs, where a critical challenge arises from the semantic gap between an end-user's high-level intent and the physical data representation. Indeed, end-user queries often involve subjective or latent concepts – such as 'good' or 'bad', 'cheap' or 'expensive' – that are not explicitly defined within the database schema. Furthermore, SQL queries in AI-DBMS environments have increased in complexity, frequently incorporating vector embeddings and LLM calls directly within the query logic. Because there is often no direct, isomorphic mapping between an end-user's abstract request and the underlying relational structures, the use of LLMs to interpret intent and generate these sophisticated queries has become essential. Consequently, we posit that leveraging text-to-SQL methods for querying multidimensional cubes in Cube+AI is mandatory for enabling effective, AI-augmented data analysis.

To investigate the feasibility of Cube+AI, in [7] we have provided a proof-of-concept by relying on the open-source vector DBMS [2] Weaviate. However, that preliminary work does not provide a general and formal framework of AI and cube integration, nor does it discuss how queries can be formulated by end-users. Therefore, in this paper we significantly extend [7] by proposing a complete Cube+AI framework that includes (i) a formal model to represent multidimensional (structured and unstructured) data and semantically rich queries over them, as well as (ii) a text-to-SQL method where an LLM is leveraged to translate natural language queries into SQL queries to be executed on an AI-DBMS. Specifically, we experiment with the use of PostgresAI and of the Gemini LLM (https://gemini.google.com/). A functional overview of querying in Cube+AI is given in Fig. 1. The end-user formulates a query (possibly involving text and images) in natural language via Gemini, which translates it into SQL by relying on a library of AI-based functions and on the multidimensional (relational) schema of data. The query is then executed on PostgresAI, and the results are eventually returned to the user.

The novel research contributions of this paper are listed below in more detail:

1. A formal extension of the multidimensional model is proposed to support, besides categorical and numerical data, also text and images, and to enable the dynamical definition of virtual categorical levels, not defined at design-time but derived at query time from other levels, e.g., via AI-based discretization.
2. We introduce a core library of AI-based SQL functions for computing virtual levels at query time and for aggregating non-numerical data.
3. We give a definition of multidimensional query using both physical and virtual levels.
4. We show that Gemini can be trained, via prompt engineering, to translate natural language queries expressed on a multidimensional cube into SQL queries to be executed on PostgresAI.

We emphasize that the focus of this paper is on showing that the enhanced querying capabilities entailed by Cube+AI can indeed be supported by an AI-DBMS and that an LLM can effectively be used to achieve text-to-SQL; the issues related to query performance optimization are out of scope.

The remainder of the paper is structured as follows. After discussing the related literature in Section 2, in Section 3 we introduce the Cube+AI framework. The two components of the framework, namely, the formal model it relies on and LLM-based text-to-SQL translation, are detailed in Sections 4 and 5 respectively. In Section 6 we describe the implementation we made on PostgresAI. Section 7 discusses the experiments we made to test Cube+AI in terms of performances and robustness, and Section 8 closes the paper by discussing the main open challenges for research.

2. Related work

Common systems for multidimensional data analysis focus on structured data with numeric measures and categorical attributes. However, in past years, some works investigated the usage of textual data as measures, as discussed in [10]. For instance, DocCube is a framework for the multidimensional exploration of large document sets through classification [11]. Lin [12] introduced the TextCube, a multi-dimensional data cube structured with a term hierarchy specifically designed for text data analysis. By incorporating two key measures-term frequency vectors and inverted indexes, a TextCube seamlessly integrates information retrieval techniques with traditional OLAP operations. Conversely, TextRank [13] serves as an unsupervised, graph-based ranking framework for natural language processing. This model evaluates the global importance of individual words or sentences, making it highly effective for both keyword extraction in text classification and sentence extraction for automatic summarization. In [14], an architecture is proposed that integrates a corporate warehouse containing structured data with a warehouse of text-rich XML documents. This integration creates a contextualized warehouse, where facts from the structured warehouse are linked to their relevant contextual documents using a fact extractor module. Users define an analysis context by providing a sequence of keywords; the system retrieves relevant documents from the document warehouse and associates them with facts from the corporate warehouse. The integrated data is then materialized as a type of OLAP cube called the R-cube, which contains all the facts, their related contextual documents, and a numeric relevance score that indicates the importance of each fact-document pair with respect to the context. While the integration of textual data in OLAP systems has been recognized as effective and useful, the existing works are not AI-based; thus, they lack the generative analysis capabilities offered by LLMs.

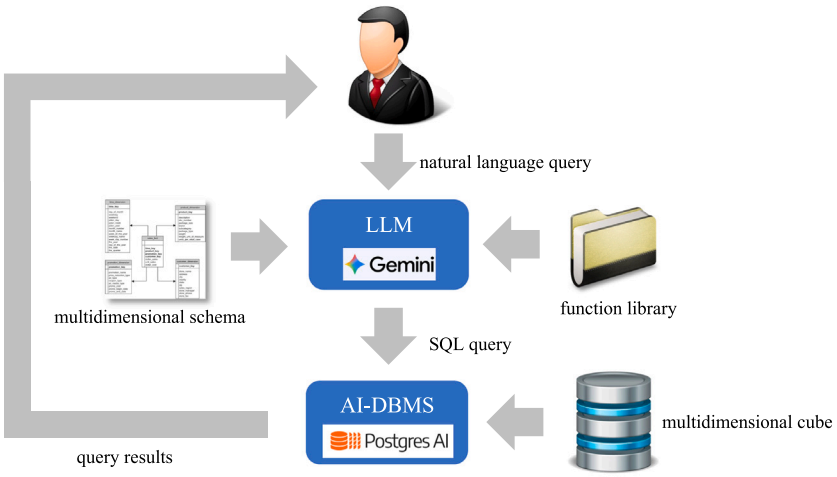


Fig. 1. Functional overview of the Cube+AI approach to querying.

Taking complex data into account in multidimensional models has also been investigated. For example, several works investigate storing and querying spatio-temporal multidimensional data [15]. A few works have studied the use of images in multidimensional analyses. For instance, iCube [6] is a similarity-based data cube designed to support OLAP similarity queries over medical images. iCube extends a traditional data warehouse by including a dimension table that stores intrinsic image features as vectors, along with distances to specific reference points (called representatives) within a metric space. A user query specifies an image and a range radius to return the images whose distance from the given one is below the radius. Multimedia data are analyzed in [16] by integrating them as measures and aggregating them to extract numerical summarized indicators. Photos are used in [17] as dimension members; by means of tags associated to them, this work allows a multidimensional exploration of huge volumes of images. In the same line, in [18] video data are introduced, allowing end-users to browse them in an online and multidimensional way. Like for textual data, these works do not rely on advanced AI methods; thus, they do not support multimodal, generative, and complex aggregation and selection methods.

A natural language interaction with OLAP systems has been investigated in a few works. COOL is a framework that supports a text-to-SQL approach tailored for an OLAP context [19]. Designed for conversational OLAP, COOL translates natural language questions into SQL queries that are executed against multidimensional data organized in a star schema. This framework achieves translation by parsing the natural language input through a structured grammar capable of recognizing both standard query keywords (such as "select" and "group by") and domain-specific vocabulary (e.g., "store" and "product"). To enable non-technical users to perform multi-dimensional data analysis, in [20] a syntax-driven parser is implemented that extracts database objects and analytical predicates from a natural language specification, sequentially compiling them into equivalent SQL queries designed for OLAP hypercubes.

Some more recent works improve over the previous ones by introducing LLMs as interfaces for OLAP systems. For example, LLMs are used to interact with MDX-based OLAP servers [21] and to generate SQL queries [22]. LLMs have also recently been used to explore multidimensional data by recommending queries to end-users [23]. On the one hand, these recent works emphasize the effectiveness of advanced AI models, such as LLMs, to interact with multidimensional datasets. On the other hand, they operate on numerical and categorical multidimensional datasets without complex data and advanced AI-based aggregation and filtering methods.

Finally, the analysis of data sources external to the organization (specifically, linked open data from the web) has been investigated in several works (see [24] for a survey). Some of these works propose to collect linked open data, transform them, and integrate them into relational data sources, while others propose to store multidimensional data as linked open data. These works show the benefit of using external data; however, they require this external data either to be structured in an ad hoc way or to be collected, cleaned, and integrated with internal multidimensional datasets by means of complex methodologies. Therefore, this integration step requires an intervention of ICT people and cannot be transparently achieved by end-users.

3. Cube+AI overview

In this section, we illustrate the main features of the Cube+AI model and introduce the approach we will adopt to translate end-user queries expressed in natural language into SQL for execution. As already stated, Cube+AI relies on an extension of the notion of *cube*. A cube is a multidimensional depiction of a business phenomenon relevant to decision-making. In the classical multidimensional model, a cube features a set of numerical *measures* and a set of categorical *dimensions*. Each dimension is the root of a hierarchy of categorical *levels* that can be used to select portions of the cube data and to group them by applying aggregation operators to measures.

To describe our proposal, we will rely on a retail case study where decision-makers want to analyze the orders made in a restaurant chain. Each order line is related to a restaurant and a dish; it is characterized by classical numerical measures, such as quantity and unit price, but also by a textual review and a picture of the dish, both provided by the customer. Besides, all restaurants and dishes have their official pictures. A possible end-user query on the order lines cube would be: "*For each year and each restaurant with a terrace, give me a short text that summarizes customer reviews*". This query asks for applying two AI tasks (picture classification and text summarization) based on the multidimensional model (grouping by restaurant and year). Therefore, it is an example of a Cube+AI query. On the one hand, it cannot be executed in a traditional OLAP setting since (i) the latter would not support generative text aggregation and (ii) there is no terraceYN Boolean level in the cube, but only a picture of each restaurant; on the other, directly applying a text summarization model to order lines would require a pre-processing step to create flattened tables for each year and restaurant.

From the structural point of view, Cube+AI extends the multidimensional model in two ways:

1. The distinction between dimensions and measures is overcome, and only two types of levels are distinguished:

- A *categorical* level is one with a finite domain of discrete values (e.g., level nation has values 'Italy', 'France', etc.).
- A *non-categorical* level potentially has an infinite domain of values; this group includes numerical levels (e.g., unitPrice and quantity), textual levels (e.g., customerReview), and image levels (e.g., restaurantPhoto).

2. *Virtual levels* are not defined at design-time; they can be dynamically defined as categorical levels to be derived at query time from other levels via some AI method, e.g., by discretization, classification, etc., and possibly by resorting to additional (i.e., external to the cube) knowledge.

As to querying expressiveness, we recall that, in classical multidimensional queries, selection and grouping are normally performed on categorical levels, while aggregation is done on measures (which in Cube+AI correspond to numerical levels) via classical aggregation operators such as SUM and AVG. Conversely, in Cube+AI, selection, grouping, and aggregation can be performed both on categorical and non-categorical levels thanks to the integrated use of AI. The additional operations enabled, besides those provided in classical multidimensional queries, are listed below:

1. **Selection and grouping on non-categorical levels**. This is made possible by adopting AI methods like classification to dynamically derive virtual categorical levels. This can be done with numerical levels (e.g., segment dishes into cheap, fair, and expensive based on their unit price), text levels (e.g., segment restaurants into cheap and expensive based on customer reviews), and image levels (e.g., segment restaurants into indoor and outdoor based on their pictures).
2. **Selection and grouping on unavailable categorical levels**. This is done by deriving virtual levels from other categorical levels via AI methods (e.g., group restaurants by their continents, the latter being derived from the restaurant countries).
3. **Aggregation on non-numerical levels**. This is made possible by applying AI methods to text levels (e.g., generate a summary of the reviews made for each restaurant) and image levels (e.g., generate for each dish a single representative picture with the common features shown in most customer pictures).

In the sample query described at the beginning of this section, grouping is done on physical levels year and restaurant, while selection is operated on virtual level terraceYN derived from restaurant pictures via classification (by computing their multimodal semantic distance with term 'terrace'). Finally, aggregation is done on text level review via summarization.

The second component of the Cube+AI framework, alongside the model, is text-to-SQL translation. As described above, the formulation of Cube+AI queries is challenging because these can involve (i) virtual cube levels, which are not physically part of the cube schema, and (ii) any methods (either AI or mathematical) required to compute these virtual levels and/or to aggregate levels. To address these challenges, the Cube+AI framework lets end-users express queries in natural language, which is then mapped onto selections, groupings, and aggregations according to the Cube+AI model by means of an LLM. This LLM is also responsible for selecting, from a core set of predefined methods we provide within a library, the methods required to achieve the tasks (e.g., classification) entailed by the query. When the query needs external knowledge (i.e., knowledge not included in the cube), the LLM calls a generative-AI function provided by the AI-DMBS. Finally, the LLM generates the SQL query for the AI-DBMS. Automation is crucial here, as manually writing SQL code for AI-DBMSs is tedious and error-prone. Overall, text-to-SQL translation in Cube+AI overcomes the barriers imposed by rigid, predefined data schemata and lets end-users seamlessly embed the similarity methods implemented in AI-DBMSs, enhancing the decision-making process with sophisticated analytical capabilities.

4. The Cube+AI model

In this section we introduce the formal model of Cube+AI. As already stated, the possibility of performing selection, grouping, and aggregation both on categorical and non-categorical levels allows the distinction between dimensions and measures to be overcome in Cube+AI. Thus, we give the following definition of cube schema.

Definition 1 (*Hierarchy and Cube Schema*). A *hierarchy* is a triple $h = (L, \succeq, \geq)$ where:

1. L is a set of *levels*, each level l being coupled with a *domain*, $Dom(l)$;
2. $\succeq$ is a *roll-up* lattice[1] of L; and
3. $\geq$ is a *part-of* partial order of $\bigcup_{l \in L} Dom(l)$.

The bottom level of $\succeq$, denoted ALL_h, has a single member ALL_h. The part-of partial order is such that, for each couple of levels l and l' such that $l \succeq l'$, for each value $u \in Dom(l)$ there is exactly one value $u' \in Dom(l')$ such that $u \geq u'$. A *cube schema* is a set of hierarchies, $C = \{h_i, i = 1, \ldots, n\}$. The top level of each hierarchy is called a *dimension* of C.

For the sake of simplicity, in all the following examples we will omit the ALL bottom levels of hierarchies and their ALL members.

Example 1. As a working example we use cube schema Order Line, which describes the orders placed at the restaurants of an international chain and features 9 dimensions:

- orderNumber, which identifies each single order.
- day, i.e., the order date with levels day ≥ month ≥ year. All these levels are categorical.
- dish, whose values correspond to all the dishes offered by the restaurant chain, with levels dish ≥ standardDishPicture. While dish is categorical, standardDishPicture is an image (Fig. 2.a).
- restaurant, whose values are the chain's various restaurants with their geographical locations and descriptions: restaurant ≥ city ≥ country; restaurant ≥ restaurantDescription; restaurant ≥ restaurantPicture. All these levels are categorical except restaurantDescription and restaurantPicture which are textual and image levels, respectively. The latter shows whether the restaurant is outdoor or indoor (Fig. 2.b).
- quantity, unit price, and amount (computed as quantity × unit price) are numerical levels that quantitatively describe each order line.
- review is a textual level containing the reviews given by customers (Fig. 2.c).
- dishPicture is an image level containing the pictures of the dishes actually served to customers (Fig. 2.d).

As to the part-of partial order we have, for instance, Le Marais ≥ Paris ≥ France and 2025-04-15 ≥ Apr-2025 ≥ 2025.

Formally speaking, the AI-enabled forms of selection and aggregation discussed above correspond to the possibility of dynamically defining categorical levels that are derived at query time from other levels, e.g., via *discretization, classification, summarization, and enrichment*, and possibly by resorting to additional (i.e., external to the cube) knowledge. These levels are called *virtual* to distinguish them from those present in the cube schema, which are called *physical*. Virtual levels extend the roll-up lattices of hierarchies as immediate successors of the physical levels they are derived from. To populate virtual levels and extend the part-of partial orders accordingly, a library of AI-based

[1] A *lattice* is a partially ordered set in which every two elements have a unique least upper bound and a unique greatest lower bound.

Fig. 2. Examples of standardDishPicture (a), restaurant picture (b), review (c), and dishPicture.

SQL functions is available to be applied to physical levels within queries for selection and grouping. If a query demands knowledge not included in the cube, we resort to the generative AI methods integrated in the AI-DBMS.

Example 2. With reference to our working example, an example of virtual level is cheapOrExpensive, the immediate successor of review. This level is assigned a value through a library function, by computing the (uni-modal) semantic distance between the review made for each order line and the terms 'cheap' and 'expensive'. An example of a corresponding part-of relationship is "Sweet, but the price …" ≥ expensive. Another example of virtual level is terraceYN, the immediate successor of restaurantPicture; its value is obtained again through a library function, by computing the (multimodal) semantic distance between the restaurant images and the term 'terrace'. As an example of a virtual level derived from a physical categorical level, we consider continent, derived via enrichment from country based on knowledge external to the cube resorting to a generative AI method integrated in the AI-DBMS. Finally, virtual level priceRange can be derived from unitPrice by comparing the price of each order line with the average price of that dish in that country as determined by classification (again, based on knowledge external to the cube). The extended lattices are depicted in Fig. 3.

Before defining cubes and queries over them, we need to introduce group-by's to rule how aggregations can be performed.

Definition 2 (*Group-by*)**.** Given cube schema C, a *group-by* of C is a tuple G of levels, at least one from each hierarchy of C. A *coordinate* of G is a tuple of values, one for each level of G. We denote with $\succeq_C$ the lattice of all the possible group-by's of C, induced by the roll-up lattices of the hierarchies in C. Given coordinate γ of group-by G and another group-by G' such that $G \succeq_C G'$, we will say that γ *rolls-up* to γ' if γ' is the coordinate of G' whose components are related to the corresponding components of γ in the part-of orders.

Note that, differently from the case of the classical multidimensional model, a group-by can include virtual and non-categorical levels as well. However, since virtual levels are always derived from other levels, the top group-by $G^\top$ in the $\succeq_C$ lattice (i.e., the finest one) includes only physical levels.

Example 3. Three group-by's of Order Line are $G^\top$ = ⟨orderNumber, day, restaurant, dish, quantity, unitPrice, amount, review, dishPicture⟩, G_1 = ⟨month, city, cheapOrExpensive⟩, and G_2 = ⟨year⟩, where $G^\top \succeq_H G_1 \succeq_H G_2$. Examples of coordinates of the three group-by's are, respectively, $\gamma^\top$ = ⟨♯0011, 2025-04-15, Le Marais, Bigburger, 20, 10.00, 200.00, "Sweet, but the price …", *picture*⟩, γ_1 = ⟨Apr-2025, Paris, expensive⟩, and γ_2 = ⟨2025⟩, where γ_0 rolls-up to γ_1 and γ_1 rolls-up to γ_2.

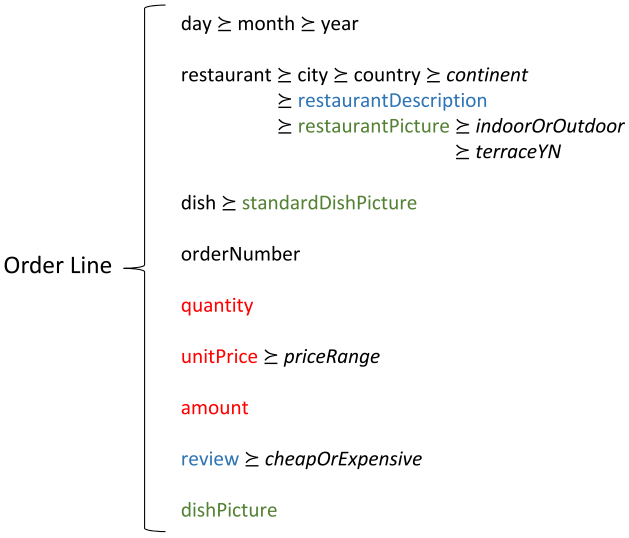


Fig. 3. Hierarchies extended with virtual levels (in italics) for the ORDER LINE cube; categorical levels in black, numerical levels in red, textual levels in blue, image levels in green. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Definition 3 (*Fact and Base Cube*)**.** A *fact* of C is a tuple including a component for each level of $G^\top$, each component storing a value of the corresponding level. A *base cube* over C is a set of facts of C.

In order to store base cubes on a (relational) AI-DBMS, we use an adaptation of the star schema. The *star schema* is widely recognized as the standard for relational implementations of the multidimensional model, as its denormalized structure allows efficient navigation, filtering, and aggregation of massive volumes of data through simple, high-performance joins. In its classical form, a star schema includes a fact table that contains measures, connected via foreign keys to multiple dimension tables that store levels. In Cube+AI, we introduce complex types to support non-categorical levels in fact and dimension tables; specifically, we use *text* and *bytea* types for textual and image levels, respectively. Furthermore, since there is no formal distinction between dimensions and measures in Cube+AI, we include in the fact table all degenerate dimensions as well (i.e., all dimensions that have no children in the roll-up lattice). Finally, we associate each text and image level to a vector column to be used for similarity queries.

Example 4. The schema used to implement the Order Line cube is shown in Fig. 4; ORDER_LINE is the fact table, while the others are dimension tables. Levels orderNumber, amount, quantity,

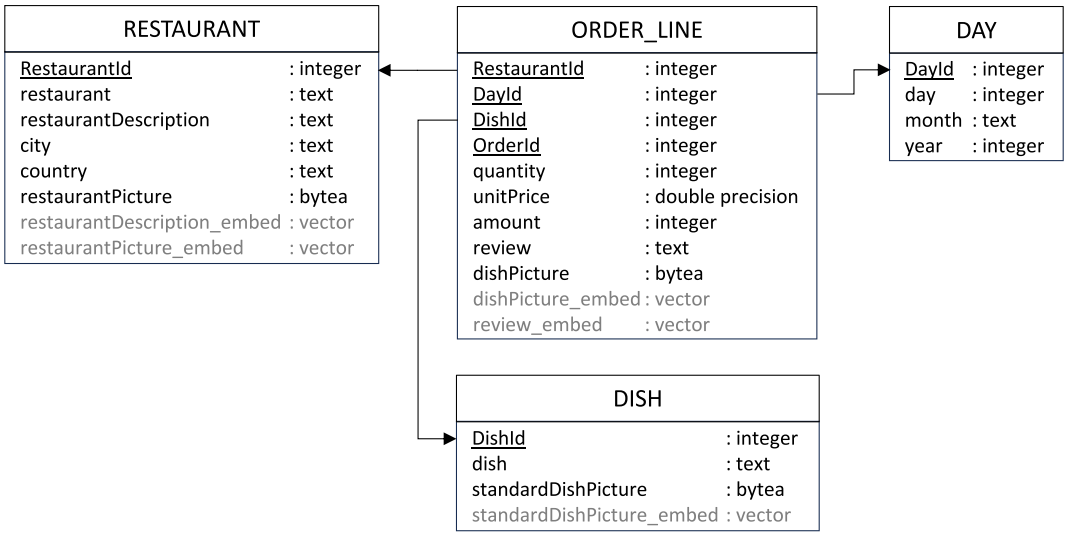


Fig. 4. Relational schema for the Order Line example (primary keys are underlined, vector columns are in gray).

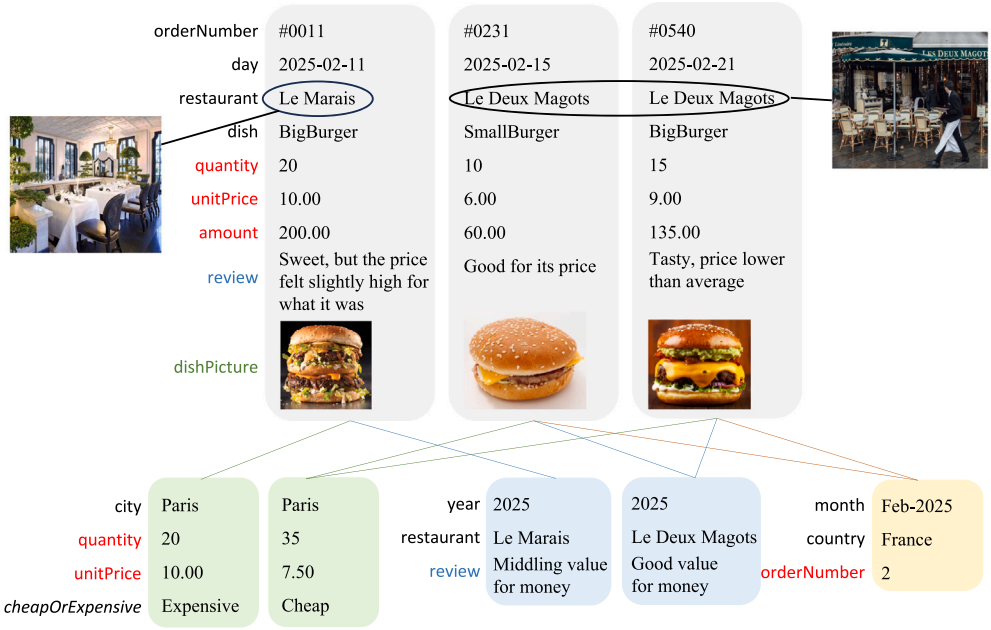


Fig. 5. Sample facts (gray) and their grouping for queries q_1 (green), q_2 (blue), and q_3 (orange) in Example 5. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

unitPrice, review, and dishPicture are degenerate dimensions; thus, they are stored within the fact table. The vector column associated to standardDishPicture is standardDishPicture_embed.

The definition we give of query in Cube+AI must consider the possibility that virtual levels are used together with physical ones for grouping and selection, and that categorical, textual, and image levels are used for aggregation together with numerical ones. In each query we have four components: G, which includes the categorical levels used for grouping; P, the set of selection predicates applied to levels of any type; M, which includes the levels of any type to be aggregated; and O, which explains how the levels in M will be aggregated (e.g., by using a simple sum or by generating a summary of texts).

Definition 4 (*Query and (Derived) Cube*)**.** Given a base cube $C^{\top}$ over schema $\mathcal{C}$, a *query* over $\mathcal{C}$ is a quadruple $q_C = (G, P, M, O)$ where:

1. G is a group-by of $\mathcal{C}$ including only categorical levels;
2. P is a (possibly empty) set of selection predicates, each involving one level of H;
3. M is a tuple of levels (M disjoint from G);

4. O is a tuple of aggregation operators such that there is one operator $\omega_i \in O$ for each level $l_i \in M$.

Let $G^+ = G \cup M$. The result of applying q_C to base cube $C^\top$, denoted $q_C(C^\top)$, is called a *(derived) cube* and is a partial function that assigns, to each coordinate γ of G satisfying the conjunction of the predicates in P, a value for each $l_i \in M$ computed by applying ω_i to the values of l_i for all the coordinates of G^+ that roll-up to γ.

As concerns the O component of q, we assume that a set of aggregation operators are available including, besides trivial ones such as SUM and AVG, AI-based ones such as TEXTSUMMARY; the methods for executing these latter operators are defined in our function library.

Example 5. As a first example, we consider query $q_1 = (\langle \text{city}, \text{cheapOrExpensive} \rangle, \{\text{month} = \text{Feb-2025}\}, \langle \text{quantity}, \text{unitPrice} \rangle, \langle \text{SUM}, \text{AVG} \rangle)$. This query returns, for each city, the total quantities ordered and the average unit prices of cheap and expensive dishes, respectively, during February 2025. In this case, quantity and unitPrice are used as measures in classical OLAP. As a second example, we consider a query that returns a summary of the reviews made in each restaurant during 2025: $q_2 = (\langle \text{restaurant} \rangle, \{\text{year} = 2025\}, \langle \text{review} \rangle, \langle \text{TEXTSUMMARY} \rangle)$. Clearly, the TEXTSUMMARY aggregation operator requires an AI-based function defined in the library. Finally, query $q_3 = (\langle \text{month} \rangle, \{\text{terraceYN} = \text{Y}\}, \langle \text{orderNumber} \rangle, \langle \text{COUNT} \rangle)$ returns the monthly number of orders made in each restaurant with a terrace. A few sample facts and their grouping for these queries are sketched in Fig. 5.

5. Text-to-SQL in Cube+AI

As mentioned in the introduction, the end-user initially submits a query in natural language, NLQ, to the chatbot of an LLM. NLQ can entail selection and grouping on any physical level of the cube schema. Importantly, selection and grouping may also involve virtual levels, as long as an explanation of how to derive them is given in NLQ. As an example to present our approach, in the remainder of the paper we will use a complex query that entails the most significant features of our approach: "*For each restaurant and price category (where the price category is determined by review similarity as 'cheap', 'fair', or 'expensive') provide a summary of customer reviews made in February 2025 for the orders that include a picture of a pepperoni pizza*" (this query corresponds to q_{10} in Table 1). Here, grouping is made on levels restaurant and priceCategory; however, while the former is physical, the latter is virtual and must be computed at run time by classifying review as suggested by the text. Thus, in this case, a virtual level is involved in grouping. Besides, a selection is required on virtual level pepperoniYN, to be derived at run time from dishPicture. Finally, aggregation relies on generative AI to create review summaries for each group. In all three cases, library functions are used.

The text-to-SQL task consists of translating NLQ into an SQL query, Q, to be formulated on a cube via an AI-DBMS (PostgresAI in our implementation). As already mentioned, to achieve this task, we resort to Gemini 3.5 Flash. To support Gemini in the translation, we created a library of AI-based SQL functions and engineered a prompt to explain when and how they can be used. At the core of these functions is the *vector* data type, a specialized array structure designed to store dense numerical representations of data, known as *embeddings*. Unlike standard arrays, the vector type is mathematically optimized for high-dimensional space, where each dimension captures a specific semantic feature of the underlying data (such as a word, image, or audio clip). It works by mapping these features into a coordinate system, allowing the AI-DBMS to perform complex geometric calculations. Instead of searching for exact keyword matches, AI-DBMSs use specialized similarity functions (such as Cosine Distance, Euclidean Distance, and Inner Product) to calculate the distance between vectors, thus enabling efficient semantic search and Retrieval-Augmented Generation (RAG). Moreover, AI-DBMSs integrate LLM reasoning directly within SQL queries by letting LLM functions be called as part of a SELECT statement to perform real-time data summarization, classification, or translation.

The complete prompt we propose for text-to-SQL is shown at our GitHub repository; in the following we summarize its main components, simplifying the procedure:

Prompt:

Role: You are an SQL programmer. I am the end-user. *Input*:

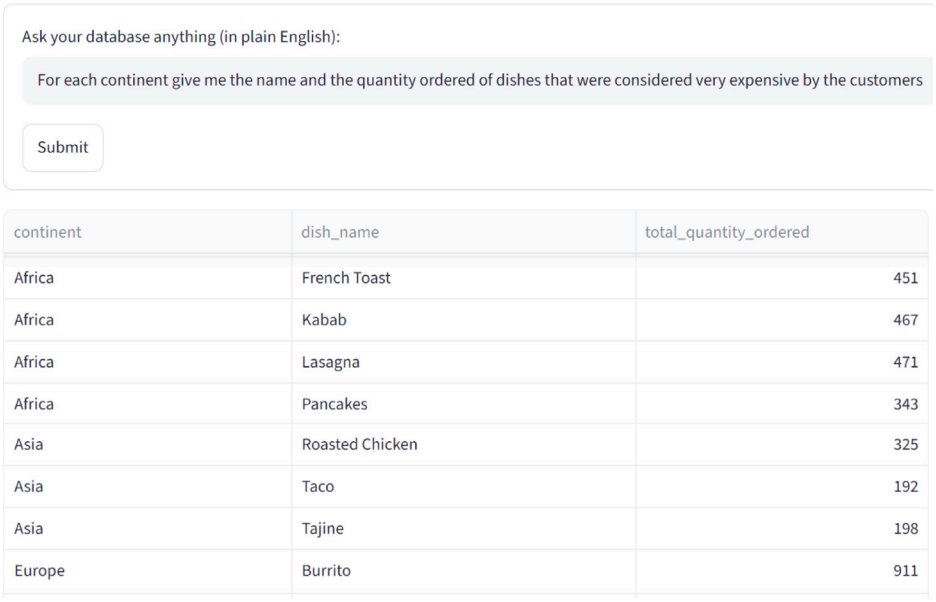
1. A natural language query, NLQ.
2. A relational schema R including one fact table and a set of dimension tables. Some columns of R have a corresponding embedding column named ''column_name''_embed. Levels required by NLQ and not present as columns of R are called virtual.
3. A file named library that contains a set of SQL functions. Each SQL function has:
   - Name
   - Description
   - A parameter named input_R_column, which corresponds to a column of a table of R
   - An optional parameter named input_function_prompt
   - Some SQL code that can be used as it is.

Task: Write the SQL formulation of NLQ, called Q.

Procedure:

1. Parse NLQ
2. Identify dimensions and measures, and map them to the tables of R
3. For each virtual level l required by NLQ that can be computed using ONLY columns in R, and for each AI-based aggregation operator ω, if any:
   (a) find the function F required by l or ω in library;
   (b) ask the user confirmation for F, providing its description; if the user does not confirm F, generate another one;
   (c) identify in NLQ the parameter input_R_column;
   (d) identify in NLQ the parameter input_function_prompt when the default value is not defined in F;
   (e) ask the user confirmation for input_function_prompt, else generate another one;
   (f) use ONLY the columns of R, and use the associated embedding columns whenever possible.
4. For each remaining virtual level l required by NLQ not computed at step 3, if any:
   - whenever possible, use a Common Table Expression (CTE) to establish a lookup table for l, and join it to the fact table;
   - identify in NLQ the text P describing l, to be used for prompting the LLM;
   - ask confirmation to the user for P; if the user does not confirm P, generate another one;
   - integrate column names from R into P if required;
   - include function ai.openai_chat_complete(P) in Q;
   - if ai.openai_chat_complete(P) is used for mathematical computation, selection, or grouping, wrap its result in a sanitization block.
5. Finalize Q with all remaining physical levels and aggregation operators

In the first part of the prompt, we assign the role of an SQL programmer to the LLM to ensure its reasoning is strictly oriented towards creating SQL queries. Besides the natural language query to be translated, NLQ, some additional inputs are provided to support the translation task:



continent	dish_name	total_quantity_ordered
Africa	French Toast	451
Africa	Kabab	467
Africa	Lasagna	471
Africa	Pancakes	343
Asia	Roasted Chicken	325
Asia	Taco	192
Asia	Tajine	198
Europe	Burrito	911

Fig. 6. A simple user interface for Cube+AI.

Table 1
Experimental workload W: natural language formulation.

Query	Natural language formulation
q_1	For each city and price category (where the price category is determined by review similarity as 'cheap', 'fair', or 'expensive'), find the average unit price and the total quantity ordered during February 2025
q_2	For each year and for each restaurant, give me a short text that summarizes customer reviews for February 2025
q_3	Give me the number of monthly orders made in restaurants with a terrace
q_4	For each restaurant continent and each month, give me the average quantity ordered (you can find the continent of each restaurant looking at the countries; for instance, if a restaurant is in France the continent is Europe)
q_5	For each restaurant, give me the total amount spent on orders that were considered expensive by the clients
q_6	For 2025, give me the total quantity of the dish named 'Burger' ordered in romantic restaurants
q_7	For each restaurant, and for orders where the picture taken by the client is similar to the official dish picture, give me the average price for 'Pizza' orders
q_8	Give me a one-line summary description for romantic restaurants
q_9	For each continent give me the name and the quantity ordered of dishes that were considered very expensive by the customers
q_{10}	For each restaurant and price category (where the price category is determined by review similarity as 'cheap', 'fair', or 'expensive') provide a summary of customer reviews made in February 2025 for the orders that include a picture of a pepperoni pizza
q_{11}	For each city and price range (either 'cheap', if the price is significantly lower than the average price of each dish in the same country based on your knowledge; 'expensive', if the price is significantly higher than the average price of each dish in the same country based on your knowledge; or 'fair' otherwise), find the average unit price and the total quantity ordered during February 2025
q_{12}	Find the average unit price of order lines by restaurant type (either indoor or outdoor)

- A relational schema, R, with its table names, columns, data types, and foreign keys. We also identify specific embedding columns to guide semantic operations.
- To prevent the LLM from generating SQL code with unsupported functions, we provide a structured library file, L. This file is critical because it forces the LLM to rely only on the functions defined within our approach, ensuring that any generated query is fully functional and compatible with PostgresAI. Each library function is defined with a name, a description, and an SQL prototype that the LLM must apply to the target query context. Each prototype requires one mandatory parameter (the database column to be processed, `input_R_column`) and an optional parameter `input_function_prompt` used to inject natural language context from the user's query directly into the function.

The second part of the prompt precisely describes the procedure the LLM must follow to transform NLQ into the final SQL query and includes the following steps:

1. **Query parsing**: The LLM parses NLQ to capture its intent and structural requirements.
2. **Schema mapping**: The LLM identifies the dimensions and measures involved in NLQ and maps them directly, when possible, to the physical tables and columns within the relational schema.
3. **Handling virtual levels and AI-based aggregation operators**: This step is critical as it explains to the LLM how to deal with cases where the user's query cannot be executed by only relying on data that are physically stored in the database and on standard aggregation operators. In such cases, the LLM must execute a specific procedure and explicitly ask the user for confirmation of any proposed functions, prompt texts, and parameters before proceeding:
    - If the required data can be derived or computed using the data stored in the schema (this is the case, for instance, of deriving priceCategory from review), the LLM must use these columns to derive this data either by combining them or by invoking functions from the library. The LLM is asked to prioritize the use of embedding columns whenever possible to optimize query execution performance.
    - If the required data cannot be derived from the database (for instance, continent), the LLM must invoke an AI-DBMS function (i.e., `ai.openai_chat_complete`), which generates a response to a prompt or conversation. After that, the LLM must use a specific function of the library dedicated to external lookup tables. To ensure that the data obtained can be safely and immediately integrated into Q, a set of rules is provided to sanitize the LLM's output.
    - If an AI-based aggregation operator is required by NLQ (for instance, TEXTSUMMARY), the LLM must look for its implementation in the library.
4. **Query Finalization**: After securing all user's confirmations, the LLM integrates the remaining physical levels and aggregation operators to generate in output the final, executable SQL query Q.

Example 6. Consider again query q_{10}. When Gemini is prompted with the corresponding natural language formulation given at the beginning of this section, it answers as follows:

Answer (excerpt):

- Steps 1 & 2: Parse NLQ and Map to R
    - Dimensions & Filtering Levels Identified: restaurant (maps to physical column RESTAURANT.RestaurantId), Month/Year (maps to physical columns DAY.month and DAY.year)
    - Filter Criteria: Month is 'February', Year is 2025 (DAY.month = 'February' and DAY.year = 2025)
    - Measures & Aggregation Operators Identified: Summary of customer reviews (requires an AI-based text aggregation operator to combine and summarize multiple text strings across the group)
    - Virtual Levels / Image Filters Identified: priceCategory (determined dynamically based on the text column ORDER_LINE.review mapped to three potential structural classifications: 'cheap', 'fair', and 'expensive'); dishPicture filter (filtering for pictures that include a 'pepperoni pizza' using the image column ORDER_LINE.dishPicture)
- Step 3: Process AI Library Functions
    1. Price Category Evaluation
        - Function found in Library: `Text Embedding Similarity`.
        - Parameter Identification: input_R_column → ORDER_LINE.review_embed; input_function_prompt → Dynamic strings depending on the bucket: 'cheap', 'fair', or 'expensive'.
    2. Dish Picture Filter
        - Function found in Library: `Image Embedding Similarity`.
        - Parameter Identification: input_R_column → ORDER_LINE.dishPicture_embedded; input_function_prompt → 'pepperoni pizza'.
    3. AI-based Aggregation (Review Summarization)
        - Function Found in Library: `Summarization-text`.
        - Parameter Identification: input_R_column → ORDER_LINE.review; input_function_prompt → Uses the default value 'give me a one-line summary.'
- Step 4: User Confirmation Requests
    1. Do you confirm the use of the library function `Text Embedding Similarity` to calculate the distance flags for 'cheap', 'fair', and 'expensive'?
    2. Do you confirm the explicit prompt parameters 'cheap', 'fair', and 'expensive' to extract and isolate these semantic profiles from the review embedding vector?
    3. Do you confirm the use of the library function `Image Embedding Similarity` to isolate dishes containing a 'pepperoni pizza'?
    4. Do you confirm the prompt parameter 'pepperoni pizza' to screen the order line images?
    5. Do you confirm the use of the library function `Summarization-text` as an aggregation operator to synthesize the customer reviews for each group?
- Step 5: Process Remaining Virtual Levels via LLM
    - All semantic requirements, filters, and aggregations for NLQ are explicitly satisfied by functions already present in your AI library file. Therefore, no standalone external LLM structures or sanitization blocks are needed.
- Step 6: Final SQL Formulation
    - Once you confirm the parameter definitions above, I will generate the final SQL query Q.

6. Implementation

In this section, we present our implementation of Cube+AI on PostgresAI. We first introduce the function library that guides the

generation of SQL queries (Section 6.1), then we discuss query implementation (Section 6.2). To streamline access by end-users and hide the technical complexity of setting up PostgresAI and running the scripts, we developed a simple web interface where end-users can interactively enter their queries in natural language and check the results (a sample screenshot is shown in Fig. 6). This interface is built with Python and uses Streamlit (https://streamlit.io/) to handle inputs and data rendering. It leverages the psycopg2 library (https://pypi.org/project/psycopg2/) to manage connections with PostgresAI and integrates the Gemini API (gemini-3.5-flash) to translate natural language queries into executable SQL queries.

6.1. Function library

The library defines the set of functions supported and guides Gemini in generating SQL queries to be executed by PostgresAI, enabling the advanced embedding-based similarity and aggregation operations required by Cube+AI over relational tables. It further specifies the parameters that must be specified during query generation.

Each function is defined by a name, a description, a set of input parameters identified from NLQ, and the associated SQL code. Depending on the function, the parameters may include an `input_function_prompt` parameter (i.e., a textual prompt with an optional default value), as well as one or more `input_R_column` parameters corresponding to relational attributes processed by the function. These specifications must be strictly followed during query generation to ensure compatibility between Gemini-generated SQL queries and the functions supported by PostgresAI. For example, when a column required by NLQ is not present in the database, Gemini must rely on the predefined function `ai.openai_chat_complete()` with the supported LLM, Llama 3.1 8B Instant, to retrieve or generate the necessary data. Without this library, Gemini could generate SQL queries that deviate from these specifications, for instance, by using unsupported models, undefined functions, or inconsistent similarity thresholds, leading to incompatibilities with PostgresAI.

Below we illustrate some functions defined in the library, together with their associated AI models and SQL implementations. We start with the function that generates a summary of a textual level (corresponding to the `TEXTSUMMARY` operator used in Example 5), defined as follows:

Name: *Summarization-text*
Description: This function summarizes text.
Parameters: `input_R_column` and `input_function_prompt` are parameters identified in NLQ. The default value of `input_function_prompt` is: "Give me a one-line summary".

SQL Code:

```
(
 ai.openai_chat_complete(
   'llama-3.1-8b-instant',
   jsonb_build_array(
     jsonb_build_object(
       'role', 'user',
       'content',
       input_function_prompt ||
       STRING_AGG(input_R_column)
     )
   )
 )->'choices'->0->'message'->>'content'
)::text
```

Similarly, the library provides embedding-based similarity functions for textual and image data. An example of a text embedding similarity function is given below:

Name: *Text Embedding Similarity*
Description: This function embeds a textual query and computes the similarity between the generated embedding vector and an existing text embedding column.
Parameters: `input_R_column` and `input_function_prompt` are parameters identified in NLQ.

SQL Code:

```
ai.ollama_embed( // ollama is a function of the Llama LLM
                 // that generates the vector
                 // representation of the input data
   'mxbai-embed-large',
   input_function_prompt,
   host => 'http://pgai-ollama:11434'
)::vector <=> input_R_column < 0.4
```

The library also supports embedding-based similarity computations for image data. To achieve this, images are first converted into textual descriptions using the Small Vision Language Model `moondream`, and these descriptions are then embedded using the text embedding model `nomic-embed-text` of Llama. The resulting embeddings are stored in the database and compared against query embeddings using the same vector similarity mechanism as for textual data.

Finally, the library supports direct similarity computations between two existing embedding vectors stored in the database, as illustrated below:

Name: *Embedding Similarity*
Description: This function computes the semantic similarity between two existing embedding vectors.
Parameters: `input_R_column1` and `input_R_column2` are parameters identified in NLQ.

SQL Code:

```
input_R_column1 <=> input_R_column2 < 0.4
```

6.2. Queries

The functions described above constitute the building blocks used by Gemini to generate executable SQL queries implementing Cube+AI operators. We illustrate their usage through query q_{10}, whose SQL translation as created by Gemini is shown in Fig. 7.

Query q_{10} combines standard multidimensional operations with AI-enabled functions from the library. The temporal selection is translated into the predicate `t.month = 'February' AND t.year = 2025` (where `t` is an alias for the `DAY` dimension table), while restaurants are identified and grouped through `ol.RestaurantId` (where `ol` is an alias for the `ORDER_LINE` fact table).

The remaining elements of the query correspond to information that is not explicitly represented by physical attributes in the relational schema. Their implementation, therefore, relies on AI-enabled functions from the library, namely:

- **Text Embedding Similarity**: The concept of priceCategory (either 'cheap', 'fair', or 'expensive') is not stored as a database attribute. The query dynamically derives this virtual level by measuring the Cosine Distance (`<=>`) between the text embeddings of the target words and the existing physical embedded review column (`ol.review_embed`) using the *Text Embedding Similarity* function.
- **Image Embedding Similarity**: The user's semantic request for pepperoni pizza is implemented using the *Image Embedding Similarity* function. The request is mapped onto the embedding column (`ol.dishPicture_embed`) using distance scoring. This illustrates that the LLM correctly prioritizes embedding columns whenever possible, as specified in the prompt, to optimize query execution.

```
SELECT ol.RestaurantId,
    -- Determining price category via the Text Embedding Similarity function
        CASE    WHEN (ai.ollama_embed('mxbai-embed-large', 'cheap',
                       host => 'http://pgai-ollama:11434')::vector <=> ol.review_embed) <
                      (ai.ollama_embed('mxbai-embed-large', 'fair',
                       host => 'http://pgai-ollama:11434')::vector <=> ol.review_embed)
                      AND
                      (ai.ollama_embed('mxbai-embed-large', 'cheap',
                       host => 'http://pgai-ollama:11434')::vector <=> ol.review_embed) <
                      (ai.ollama_embed('mxbai-embed-large', 'expensive',
                       host => 'http://pgai-ollama:11434')::vector <=> ol.review_embed)
                THEN 'cheap'
                WHEN (ai.ollama_embed('mxbai-embed-large', 'fair',
                       host => 'http://pgai-ollama:11434')::vector <=> ol.review_embed) <
                      (ai.ollama_embed('mxbai-embed-large', 'expensive',
                       host => 'http://pgai-ollama:11434')::vector <=> ol.review_embed)
                THEN 'fair'
                ELSE 'expensive'
        END AS priceCategory,
    -- AI-based Text Aggregation Operator (Summarization-text code injected)
    (ai.openai_chat_complete(
        'llama-3.1-8b-instant',
        jsonb_build_array(
            jsonb_build_object(
                'role', 'user',
                'content', 'give me one-line summary ' || STRING_AGG(ol.review, ', ')
            )
        )
    )->'choices'->0->'message'->>'content' )::text AS customer_reviews_summary
FROM ORDER_LINE ol
JOIN DAY t ON ol.DayId = t.DayId
WHERE t.month = 'February' AND t.year = 2025 -- Filtering physical constraints
    -- Image filtering using the Image Embedding Similarity function
    AND ai.ollama_embed('nomic-embed-text', 'pepperoni pizza',
                        host => 'http://pgai-ollama:11434')::vector
                        <=> ol.dishPicture_embed < 0.4
GROUP BY ol.RestaurantId, priceCategory;
```

Fig. 7. The SQL code created by Gemini for q_{10}.

- **Summarization-text**: In order to summarize the textual review column, the LLM uses the specific function defined within the library (`ai.openai_chat_complete`). It correctly applies the usage prototype by combining the required instruction parameter ("Give me a one-line summary") with the aggregated physical column data (`STRING_AGG(ol.review, ', ')`).

The execution of the generated SQL query on PostgresAI produces the result shown in Fig. 8.[2]

7. Experiments

In this section, we present an experimental evaluation of the proposed implementation. To this end, we designed a workload of queries covering the main analytical capabilities of Cube+AI and executed them on our PostgresAI-based prototype (see Section 7.1). We recall from the Introduction that query optimization is out of scope for this paper; nevertheless, in Section 7.2 we discuss the performances of the query workload with reference to an experimental setting in which PostgresAI runs on a laptop equipped with a 13th Gen Intel Core i7-13700H (14 Cores/20 Threads, up to 5.00 GHz), 32 GB LPDDR5 RAM, and a high-speed 1 TB PCIe Gen 4 NVMe SSD running Windows 11. Finally, we evaluate, in Section 7.3, the robustness of the framework by considering different natural language query formulations.

[2] The SQL code and outputs for the remaining queries, along with the database schema, can be found at our GitHub repository.

7.1. Workload

The workload W we have used to test Cube+AI includes 12 queries, designed to cover the most significant combinations of classical and advanced selection, grouping, and aggregation. The natural language formulations of these queries are shown in Table 1. Besides, Table 2 shows, for each query in W, the type(s) of level involved in grouping, selection, and aggregation. For each virtual level (in italics), we also show the type of the physical level from which it is derived. For instance, in q_3, "image → *cat*" means that the categorical (Boolean) virtual level terraceYN is derived from the image level restaurantPicture; in q_4, "cat → *cat*" means that the categorical level continent is derived from the categorical level country.

The *Method(s)* column shows the particular AI method(s) used. For example, in q_4, *Generative* refers to the library method `Summarization-text`, which generates a short summary of texts. The column *Ext.* refers to the usage or not of external cube data, while the last column, *Task(s)*, shows the main task(s) of the AI methods used by the query. For instance, *Ext.* is Yes for q_4, since the latter uses data external to the cube to create the continent virtual level from country in order to achieve an *Enrichment* AI task.

For grouping, selection, and aggregation, we have used all possible data types; for example, as to grouping, q_1 operates text, q_4 on a categorical value, while q_{12} involves images. Selection is organized in the same way: q_3 operates on images, q_5 on text, and q_2 on categorical values. Besides classical aggregation functions, denoted as *num* → *num*, we introduce AI-based aggregation functions that are applied to text and categorical values (e.g., q_2 and q_3). Note that aggregation of images

	restaurant_id integer	pricecategory text	customer_reviews_summary text
1	1	expensive	Overall reviews are overwhelmingly positive, praising the excellent "very great experience," "good vibes," and "delicious meals," but some reviewers note that the experience is "expensive" and the ...
2	1	fair	Customers reported a highly positive experience with the service or product, praising its quality, ease of use, and efficient support.
3	2	cheap	"I was impressed by the overall atmosphere and service, but the quality of the food didn't quite match the premium price I paid for it."
4	2	expensive	Mixed reviews for the meal, ranging from extremely satisfied to overpriced with some average taste, with overall recommendation still leaning towards positive.
5	3	cheap	Reviews that "highly recommend" are overwhelmingly positive evaluations that praise a product, service, or experience, and often mention specific benefits or features that make it stand out, fre...
6	3	expensive	Many reviewers praised the restaurant for its delicious meals and good vibes, but were divided on the value for the high price.
7	3	fair	Reviewers were extremely satisfied with their experience, highlighting exceptional service, quality, and overall satisfaction.
8	4	cheap	I'm happy to help, but I need more information. Which reviews or products would you like me to give a one-line summary of?
9	4	expensive	The reviews describe an "amazing" but "expensive" meal with varying opinions on the value for money, consistently expressing positive experiences.
10	4	fair	Reviewers generally expressed high satisfaction and delight with a pleasant or exceptional experience at a specific location or with a service.
11	5	cheap	"Highly Recommend" is a Netflix reality series that invites experts to rate various products and services in a straightforward, unapologetic manner, providing viewers with their honest, unbiased ...
12	5	expensive	The majority of reviews describe a very positive experience, with frequent mentions of a "very great experience", "amazing meal", "very good vibes", and "highly recommend", indicating a generall...
13	5	fair	Customers have generally expressed satisfaction with the service or product, describing it as excellent, great, or very good, with some mentioning specific positive experiences.
14	6	cheap	The reviewer had an overwhelmingly positive experience and highly recommends whatever it was they reviewed.
15	6	expensive	Overwhelmingly positive reviews with some minor criticism about high prices, consistently praising the quality of the meals and overall dining experience.
16	7	cheap	"Highly Recommend" and "Very Good Vibes" are usually seen as positive phrases typically found in customer reviews indicating exceptional service, quality, or experience.
17	7	expensive	The overwhelming consensus is that the establishment offers a wonderful dining experience with highly praised food and service, but with some reviewers criticizing the high price.
18	8	cheap	The reviewer was extremely satisfied but found the taste unremarkable compared to the price.

Fig. 8. The result of q_{10}.

Table 2
Experimental workload W: classification (U and M stand for uni-modal and multimodal, respectively).

Query	Grouping	Selection	Aggregation	Method(s)	Ext.	Task(s)
q_1	cat, text → *cat*	cat	num → num	(U) Embedding similarity	N	Classification
q_2	cat	cat	text → text	(U) Generative	N	Summarization
q_3	cat	image → *cat*	cat → num	(M) Embedding similarity	N	Classification
q_4	cat → *cat*, cat	–	num → num	(U) Generative	Y	Enrichment
q_5	cat	text → *cat*	num → num	(U) Embedding similarity	N	Classification
q_6	–	cat, text → *cat*	num → num	(U) Embedding similarity	N	Classification
q_7	cat	cat, image → *cat*	num → num	(M) Embedding similarity	N	Classification
q_8	–	text → *cat*	text → text	(U) Embedding similarity, Generative	N	Classification, Summarization
q_9	cat → *cat*	text → *cat*	num → num	(U) Embedding similarity, Generative	Y	Classification, Enrichment
q_{10}	cat, text → *cat*	image → *cat*	text → text	(M) Embedding similarity, Generative	N	Summarization, Classification
q_{11}	cat, num → *cat*	cat	num → num	(U) Generative	Y	Discretization
q_{12}	image → *cat*	–	num → num	(M) Embedding similarity	N	Classification

Table 3
Database sizes in Gigabytes.

Fact table rows	DB size
18 000	7.4 GB
180 000	79.5 GB
700 000	220.2 GB

is not included in W because, although it is formally supported by Cube+AI, it is not currently possible in Llama. From query q_1 to q_7, one AI method is used. Queries from q_8 to q_{10} are more complex since they use multiple AI methods on different query components, as previously seen for q_{10}.

7.2. Performance

To evaluate the proposed implementation in terms of storage requirements and query execution performance, we have defined three versions of the Order Line cube with increasing sizes of the fact table, ranging from 18 000 to 700 000 rows. Table 3 shows that the storage space is not excessive even for large fact tables; this is especially significant considering that each row of the fact table includes an image whose size ranges between 145 kB and 650 kB.

Fig. 9 shows the execution time of each query in W for the different fact table sizes. Most queries execute in a few seconds even on the largest fact table, with the exception of q_{11}. For q_{11}, the SQL formulation generated by Gemini performed AI-based classification row by row, forcing PostgresAI to execute an AI call for every order line item, even when multiple orders shared the same features (e.g., dish name, country, and unit price). The execution time for this formulation was approximately 4 h. To avoid these redundant calls, we manually optimized the query by first computing the classification only for the distinct combinations of these features and then reusing the results. This optimization substantially reduced the number of AI requests and improved execution time to 14 min for the largest fact table, as shown in Fig. 9.

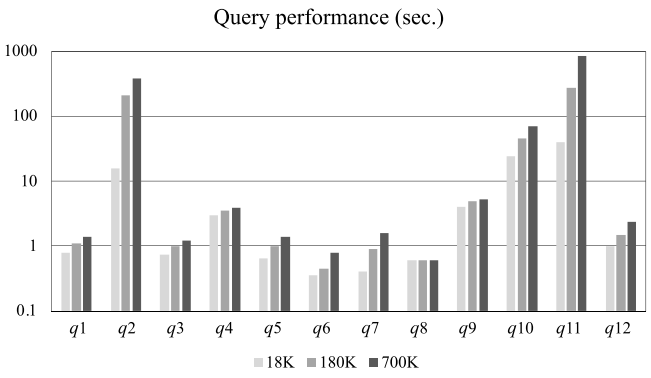


Fig. 9. Query performances (in seconds, algorithmic scale) for workload W with three fact table sizes.

A percentage breakdown of the execution times is shown in Fig. 10 with reference to the largest fact table (those for the other fact tables are obviously quite similar). In the chart, ET stands for the time taken by the embedding function to embed the text required for the query; ST for the time taken to perform the similarity calculation between the query vectors; LLMT for the time taken to request external information from the LLM (either for summarization or external knowledge); and SQLT for the time taken to perform the remaining operations (WHERE,

Table 4
Breakdown of execution times for W with the 700K fact table.

Query	ET	ST	LLMT	SQLT	Total
q_1	0.40	0.01	–	0.99	1.40
q_2	–	–	377	4.00	381.00
q_3	0.20	0.07	–	0.93	1.20
q_4	–	–	3.10	0.80	3.90
q_5	0.27	0.20	–	0.93	1.40
q_6	0.26	0.08	–	0.46	0.80
q_7	–	1.40	–	0.20	1.6
q_8	0.15	0.10	0.3	0.05	0.60
q_9	0.20	0.30	4.3	0.50	5.30
q_{10}	1.00	9.00	58.30	1.70	70.00
q_{11}	–	–	847.00	3.00	850.00
q_{12}	0.30	0.07	–	2.00	2.40

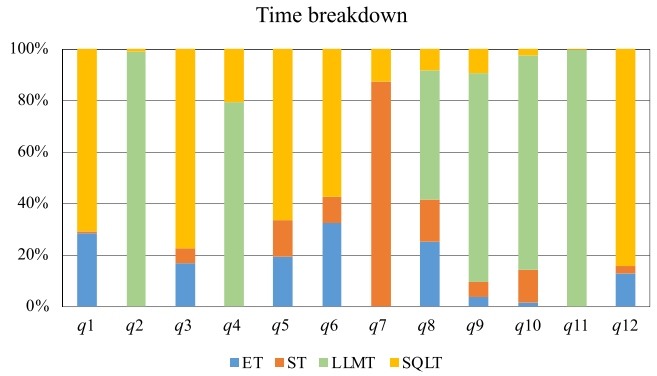


Fig. 10. Breakdown of execution times for W with the 700K fact table (ET: time to embed the text required; ST: time to compute similarity; LLMT: time to request external information; SQLT: time for the remaining operations).

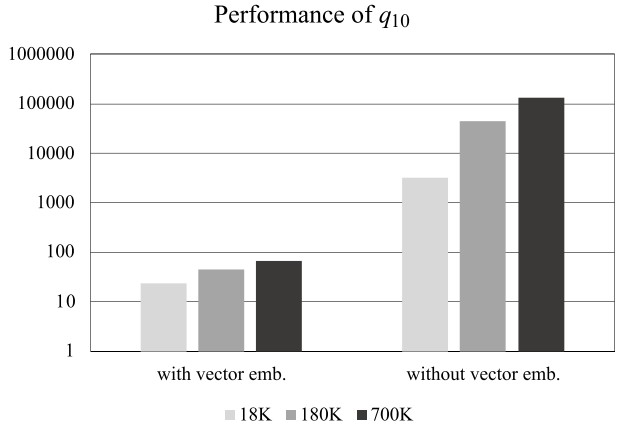


Fig. 11. Performances of q_{10} (in seconds, algorithmic scale) with and without vector embeddings.

JOIN, GROUP BY). Clearly, the time distribution changes significantly with the query depending on the operations involved (for instance, q_2 requires no embedding, so both ET and ST are null, while q_5 does not use the LLM, so LLMT is null). However it is important to note that, in absolute terms, the only relevant time is LLMT, which takes 6 min for q_2 and 14 min for q_{11}, since all other components are largely below 10 s for each query (see Table 4).

Finally, Fig. 11 compares the performance of query q_{10} with and without the use of embeddings. The results show that using embeddings natively stored within the AI-DBMS improves performance by $2 \div 3$ orders of magnitude. Although generating these embeddings is very time-consuming (for instance, generating image and text embeddings from scratch for the 700K database takes about 1500 and 30 h, respectively), we recall that it should be done incrementally at ETL (Extract-Transform-Load) time, i.e., when the cube is refreshed with new data.

7.3. Robustness

In this section we discuss the results of the tests we made to assess the robustness of Cube+AI with respect to the workload, by considering queries that differ from those in W either in their structure or in the natural language expressions they use. The complete set of queries we used can be found at our GitHub repository.

7.3.1. User queries

First of all, we tried queries that might be formulated by users not aware of the details of Cube+AI. Specifically, we asked a small team of researchers with some experience in OLAP to express in natural language some analytical queries that they believe could be interesting in the restaurant domain. We just gave them a general description of Cube+AI, emphasizing that its goal is to integrate OLAP and AI, and a list of the attributes in the Order Line cube; to avoid introducing biases, no example of the queries in W was shown to them.

Overall, we got 21 queries. Of these, 7 were very similar to those in W – which shows that W is representative of a real one – and were discarded. Of the remaining 14, 9 have the same structure of those in W and require either AI methods (e.g., "*Find the average number of orders for restaurants located close to the sea*") or external knowledge (e.g., "*Find the total quantity ordered for cities having more than 50 000 population*"). These queries were correctly handled and executed by Cube+AI except one: "*Tell me the percentage of restaurants per city where the reviews indicate the restaurants' description being inconsistent*". When translating this query, Gemini erroneously targeted only restaurantDescription, ignoring review.

The remaining 5 queries, while still of analytical nature, require a more complex computation; two examples are "*What is the difference in sales occurring during holidays and working days?*" and "*Are restaurants in the countryside more expensive than those by the seaside?*", each of which requires a comparison between the results of two aggregated queries. Interestingly, with no further instruction, Gemini was capable of going beyond our prompt and correctly translating both queries into SQL.

The SQL generated for the former query is shown in Fig. 12. External knowledge is leveraged to classify months into categories representing either holiday seasons or standard working periods. Each order in the database is then mapped onto these categories based on the corresponding order date. Finally, total order amounts are computed for both categories, and working sales are subtracted from holiday sales.

```
WITH date_classification AS (
    SELECT date_id,
        -- Strict sanitization block applied to the LLM temporal classification
        CAST(REGEXP_REPLACE((REGEXP_MATCH(
                    (ai.openai_chat_complete('llama-3.1-8b-instant', jsonb_build_array(
                            jsonb_build_object(
                                'role', 'user', 'content',
                                'Analyze this date information: Month is ''' || month || '''
                                and Year is ''' || year || '''. Is a generic day in this month
                                and year typically considered a holiday month or a working period?
                                Reply with exactly ''1'' if it leans heavily towards holiday
                                seasons/vacation months or ''0'' if it is a standard working period.
                                Reply with just the number.'
                            )
                        )
                    )->'choices'->0->'message'->>'content')::text, '[0-9.]+'
                ))[1], '[^0-9.]', '', 'g'
            ) AS DOUBLE PRECISION
        ) AS is_holiday
    FROM time
),
sales_by_day_type AS (
    SELECT
        -- Protecting against empty responses using NULLIF before conditional evaluation
        CASE WHEN NULLIF(dc.is_holiday, NULL) = 1 THEN 'holiday' ELSE 'working' END AS day_type,
        SUM(o.amount) AS total_sales
    FROM order_Line o
    JOIN date_classification dc ON o.date_id = dc.date_id
    GROUP BY
        CASE WHEN NULLIF(dc.is_holiday, NULL) = 1 THEN 'holiday' ELSE 'working' END
)
SELECT
    COALESCE(SUM(CASE WHEN day_type = 'holiday' THEN total_sales END), 0) -
    COALESCE(SUM(CASE WHEN day_type = 'working' THEN total_sales END), 0) AS sales_difference
FROM sales_by_day_type;
```

Fig. 12. The SQL code created by Gemini for query "*What is the difference in sales occurring during holidays and working days?*".

7.3.2. Operational queries

The Cube+AI approach is not meant to be used by occasional users (in our example, potential customers) interested in operational queries (for instance, aimed at finding a good restaurant) but by business users (in our example, managers of the restaurant chains) willing to ask analytical, OLAP-style queries. However, it is clearly interesting to check how Cube+AI performs when dealing with queries that do not fall in the category specified by our Definition 4. A query such as "*Find inexpensive restaurants*" is not an analytical and does not fall into Definition 4 (it has no grouping and no aggregation); nevertheless, in principle, it could be answered using the cube data. To investigate this issue, we created a few operational queries focused on finding either restaurants (e.g., "*Find inexpensive restaurants*") or dishes (e.g., "*Where can I find the most authentic sushi in Paris?*"). Surprisingly, Gemini appeared to partially override our prompt and generated, in all cases, correct SQL queries with no aggregation, still using AI methods where necessary.

The SQL code for the former query is shown in Fig. 13. The query embeds the term "inexpensive" and calculates the vector similarity with restaurant descriptions using a fixed threshold of 0.4 as defined in the library.

7.3.3. Queries with unconventional/fuzzy expressions

A check on the TripAdvisor and Reddit platforms revealed that not only are most queries in the restaurant domain operational (as expected), but they also make large use of unconventional or fuzzy natural language expressions to select restaurants. Thus, we extracted from those platforms a set of unconventional (e.g., "*Which restaurants are an absolute tourist trap?*") and fuzzy (e.g., "*Find not-so-expensive dishes*") expressions and used them to create some additional queries.

In most cases the queries were correctly translated into SQL. Clearly, while deciding whether to translate these queries using an AI method or resorting to external knowledge is up to Gemini, the burden of handling unconventional and fuzzy expressions is set on Llama. The only exception is query "*Which restaurant has the absolute worst service?*"; in translating this one, Gemini erroneously targeted restaurantDescription instead of review.

7.3.4. Preference queries

Finally, we wanted to check to what extent Cube+AI can manage preference queries, where users can express multiple requirements and combine them by either prioritization or Pareto operators. Although some attempts have been made to consider preferences in OLAP queries [25], preference queries are more common in operational contexts. Thus, we created two queries that rely on two competing predicates to select restaurants and dishes, combined either via prioritization (e.g., "*Find a restaurant that first of all is cheap, and possibly has a terrace*") or via Pareto-optimality (e.g., "*Find the dishes that offer the best compromise between being cheap and having good reviews*").

Surprisingly, even in this case Gemini was capable of handling the query fairly well without having to modify the prompt. For instance, the code for the Pareto query is shown in Fig. 14. To find the best compromise, the query measures the vector similarity between each review and the term "good", averages this similarity score for each dish, and multiplies it by the dish average price to rank the best-value options.

Overall, the results discussed above suggest that Cube+AI is quite robust even across types of queries different from those for which the framework has been conceived.

```
SELECT restaurant_id, restaurant_description, city,
    -- Calculate the cosine distance (smaller distance indicates higher semantic similarity)
    (ai.ollama_embed(
        'mxbai-embed-large', 'inexpensive restaurant', host => 'http://pgai-ollama:11434'
    )::vector <=> restaurant_description_embedded) AS price_similarity_distance
FROM location
WHERE
    -- Filter out locations that do not semantically align with the cheap/budget concept
    (ai.ollama_embed(
        'mxbai-embed-large', 'inexpensive restaurant', host => 'http://pgai-ollama:11434'
    )::vector <=> restaurant_description_embedded) < 0.4
ORDER BY  price_similarity_distance ASC;
```

Fig. 13. The SQL code created by Gemini for query "*Find inexpensive restaurants*".

```
WITH dish_metrics AS (
    SELECT d.dish_id, d.dish_name,
        -- Metric 1: Average unit price (lower is cheaper)
        AVG(ol.unit_price) AS average_price,
        -- Metric 2: Average distance to 'good' sentiment (lower is more positive)
        AVG(
            ol.review_embedded <=> ai.ollama_embed(
                'mxbai-embed-large', 'good', host => 'http://pgai-ollama:11434'
            )::vector
        ) AS average_good_distance
    FROM dish d
    JOIN order_Line ol ON d.dish_id = ol.dish_id
    WHERE ol.review_embedded IS NOT NULL
    GROUP BY  d.dish_id, d.dish_name
)
SELECT dish_id, dish_name, average_price, average_good_distance,
    -- Best compromise formula (lower score indicates optimal balance of price and rating)
    (average_price * average_good_distance) AS compromise_score
FROM dish_metrics
ORDER BY compromise_score ASC;
```

Fig. 14. The SQL code created by Gemini for query "*Find the dishes that offer the best compromise between being cheap and having good reviews*".

8. Conclusion and open challenges

Recent advances in AI offer a compelling research opportunity to re-imagine multidimensional analyses beyond their traditional focus on structured (categorical and numeric) data. In particular, the integration of multimodal and generative AI models into DBMSs supported by AI-DBMSs gave us a chance to envision the Cube+AI framework, which relies on an extension of the multidimensional model to enable semantically rich queries over text, images, and numeric data. To validate it, we have described an implementation that uses PostgresAI as an AI-DBMS and Gemini as an LLM to support the text-to-SQL process.

While, as shown in Section 7, the Cube+AI framework establishes a robust foundation for integrating AI-driven multimodal analysis into the multidimensional model, some limitations have emerged during research; we list them in the following with their possible mitigations, which constitute a subject of our future work:

1. *Query correctness*. Hallucinations may prevent the LLM from generating the correct query. The adoption of improved reasoning strategies, such as few-shot learning [26] and chain-of-thought [27], may alleviate this problem.
2. *Results correctness*. The data generated via AI (e.g., text summaries and external knowledge) is inherently vulnerable to hallucinations and semantic drift; to ensure the trustworthiness of the results produced, we will develop rigorous evaluation metrics and validation constraints.
3. *Optimization*. As already mentioned, query optimization is out of scope for this paper. As discussed in Section 7.2, different formulations for a query may hugely impact its execution time. Part of our future work will be devoted to studying optimized formulations for different types of queries and teaching them to the LLM.
4. *Scalability*. Relying on static vector embeddings and pre-trained foundation models introduces scalability and efficiency bottlenecks, particularly when processing massive data cubes with high-cardinality textual or visual attributes. To address this issue, the definition of ad-hoc index structures and query execution plans should be investigated.
5. *Portability*. The results obtained with PostgresAI and Gemini are encouraging. However, we plan to test our approach with other AI-DBMSs and LLMs to compare performances, scalability, and AI-based analysis capabilities.
6. *User interaction*. From an architectural point of view, the integration of AI-DBMSs queries within OLAP servers would let Cube+AI benefit from the multidimensional analysis capabilities that the latter natively offer (e.g., hierarchy navigation, complex mathematical functions, etc.); the design of a visual interface based on the pivot table paradigm to support textual and image data as well as natural language is also a promising research issue.
7. *User preferences*. Analytical tasks entail complex OLAP queries, often returning large volumes of facts, sometimes providing little or no information. Thus, as argued in [25], expressing user preferences could be highly valuable in this domain. In that paper, the myOLAP approach has been introduced as a way to express preferences on multidimensional queries with three main features: (i) preferences can be expressed on both numerical and categorical attributes; (ii) they can also be expressed on the grouping level of facts; (iii) they can be combined using both prioritization and Pareto operators. While myOLAP relies on a complex SQL-like syntax to let users express their preferences, integrating myOLAP into the Cube+AI framework would let users benefit of a simpler interaction based on natural

language. However, despite the encouraging results discussed in Section 7.3.4, a full integration would require a large reengineering of the prompt as well as ad hoc algorithms for efficiently executing queries, hence, it is left for future work.

8. *Domain knowledge*. The more the LLM knows about the domain to be analyzed by end-users, the more accurate the text-to-SQL translation will be, even when the natural language formulation of the query is blurry and imprecise. Although we already address this issue by prompting end-users during the translation process to ensure that the generated queries are adherent to their intents, another possible step in this direction could be making a catalog of common multidimensional modeling solutions from different domains available to the LLM.

CRediT authorship contribution statement

Houssam Bazza: Writing – original draft, Validation, Software, Investigation, Data curation. **Sandro Bimonte:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Stefano Rizzi:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Formal analysis, Conceptualization. **Sana Sellami:** Writing – review & editing, Supervision, Methodology, Formal analysis, Conceptualization. **Hassan Badir:** Writing – review & editing, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially funded by the 101226371-HORIZON-MSCA-2024-DN-JD grant 'IoRT Data Management and Analysis for Sustainable Agriculture (GreenFieldData)'.

Data availability

Research Link Provided

Query workload and prompts (Original data) (GitHub)

References

[1] E. Eigner, T. Händler, Determinants of LLM-assisted decision-making, 2024, CoRR arXiv:2402.17385.
[2] J.J. Pan, J. Wang, G. Li, Survey of vector database management systems, VLDB J. 33 (2024) 1591–1615.
[3] M. Golfarelli, S. Rizzi, Data Warehouse Design: Modern Principles and Methodologies, McGraw-Hill, 2009.
[4] S. Bimonte, M. Schneider, O. Boussaid, Business intelligence indicators: Types, models and implementation, Int. J. Data Warehous. Min. 12 (2016) 75–98.
[5] M. Bouakkaz, Y. Ouinten, S. Loudcher, OLAP textual aggregation approach using the Google similarity distance, Int. J. Bus. Intell. Data Min. 11 (2016) 31–48.
[6] L.P. Annibal, J.C. Felipe, C.D. Ciferri, R.R. Ciferri, iCube: A similarity-based data cube for medical images, in: Proc. CBMS, Perth, Australia, 2010, pp. 321–326.
[7] S. Bimonte, C. Boyadjian, S. Rizzi, S. Sellami, Towards AI-powered multi-modal generative OLAP, in: Proc. ER Forum, Poitiers, France, 2025, pp. 190–197.
[8] L. Mei, S. Mo, Z. Yang, C. Chen, A survey of multimodal retrieval-augmented generation, 2025, CoRR abs/2504.08748.
[9] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Zhang, J. Fan, G. Li, N. Tang, Y. Luo, A survey of text-to-SQL in the era of LLMs: Where are we, and where are we going? IEEE Trans. Knowl. Data Eng. (2025).
[10] M. Bouakkaz, Y. Ouinten, S. Loudcher, Y.A. Strekalova, Textual aggregation approaches in OLAP context: A survey, Int. J. Inf. Manag. 37 (2017) 684–692.
[11] J. Mothe, C. Chrisment, B. Dousset, K. Alaux, DocCube: multi-dimensional visualisation and exploration of large document sets, J. Am. Soc. Inf. Sci. Technol. 54 (2003) 650–659.
[12] F. Zhu, J. Han, B. Ding, C.X. Lin, B. Zhao, Text cube: Computing IR measures for multidimensional text database analysis, in: Proc. ICDM, 2008, pp. 905–910.
[13] R. Mihalcea, P. Tarau, TextRank: Bringing order into text, in: Proc. EMNLP, Barcelona, Spain, 2004, pp. 404–411.
[14] J.M. Perez, R. Berlanga, M.J. Aramburu, T.B. Pedersen, Contextualizing data warehouses with documents, Decis. Support Syst. 45 (2008) 77–94.
[15] A.A. Vaisman, E. Zimányi, Mobility data warehouses, ISPRS Int. J. Geo Inf. 8 (2019) 170.
[16] A. Arigon, M. Miquel, A. Tchounikine, Multimedia data warehouses: a multiversion model and a medical application, Multim. Tools Appl. 35 (2007) 91–108.
[17] B.Þ. Jónsson, G. Tómasson, H. Sigurþórsson, Á. Eiríksdóttir, L. Amsaleg, M.K. Lárusdóttir, A multi-dimensional data model for personal photo browsing, in: Proc. MMM, Springer, 2015, pp. 345–356.
[18] H. Lee, S. Park, J.H. Yoo, A data cube model for surveillance video indexing and retrieval, in: Proc. SIGMAP, 2013, pp. 163–168.
[19] M. Francia, E. Gallinucci, M. Golfarelli, COOL: a framework for conversational OLAP, Inf. Syst. 104 (2022) 101752.
[20] M.A. Naeem, I.S. Bajwa, Generating OLAP queries from natural language specification, in: Proc. ICACCI, 2012, pp. 768–773.
[21] S. Bimonte, S. Rizzi, Text-to-MDX: LLM-assisted generation of MDX queries from user questions, in: Proc. ER, Poitiers, France, 2025, pp. 165–181.
[22] D. Petković, Applying Text-to-SQL to SQL/OLAP using LLMs, in: Proc. MLNLP, 2025, pp. 1–13.
[23] X. Liu, Y. Yang, LLM-assisted conversational navigation over multidimensional data cubes, Expert Syst. Appl. 328 (2026) 132958.
[24] A. Abelló, O. Romero, T.B. Pedersen, R. Berlanga, V. Nebot, M.J. Aramburu, A. Simitsis, Using semantic web technologies for exploratory OLAP: a survey, IEEE TKDE 27 (2014) 571–588.
[25] M. Golfarelli, S. Rizzi, P. Biondi, myOLAP: An approach to express and evaluate OLAP preferences, IEEE Trans. Knowl. Data Eng. 23 (2011) 1050–1064.
[26] T.B. Brown, et al., Language models are few-shot learners, in: Proc. NeurIPS, 2020, pp. 1877–1901.
[27] J. Wei, et al., Chain-of-thought prompting elicits reasoning in large language models, in: Proc. NeurIPS, New Orleans, LA, USA, 2022, pp. 24824–24837.