COOL: A Framework for Conversational OLAP

Matteo Francia, Enrico Gallinucci, Matteo Golfarelli*

DISI – University of Bologna, Via dell'Università 50, 47522 Cesena, Italy

Abstract

The democratization of data access and the adoption of OLAP in scenarios requiring hand-free interfaces push towards the creation of smart OLAP interfaces. In this paper, we introduce COOL, a framework devised for COnversational OLap applications. COOL interprets and translates a natural language dialogue into an OLAP session that starts with a GPSJ (Generalized Projection, Selection, and Join) query and continues with the application of OLAP operators. The interpretation relies on a formal grammar and on a repository storing metadata and values from a multidimensional cube. In case of ambiguous or incomplete text description, COOL can obtain the correct query either through automatic inference or user interactions to disambiguate the text. Our tests show very promising results in terms of effectiveness, efficiency, and user experience. Besides adding novel support to the interpretation and translation of complete analytical OLAP sessions, COOL achieves an average accuracy of 94% in the interpretation of GPSJ queries from real datasets.

Keywords: Natural language processing, OLAP

1. Introduction

Nowadays, one of the most popular research trends in computer science is the democratization of data access, analysis, and visualization, which means opening them to end-users lacking the required vertical skills on the services themselves [3, 22, 30]. Smart personal assistants [16] (Alexa, Siri, etc.) and auto-machine-learning services [32] are examples of such research efforts that are now on corporate agendas [6].

In particular, interfacing natural language processing (either written or spoken) to database systems opens to new opportunities for data exploration and querying [20]. Actually, in the area of data warehouse, OLAP (On-Line Analytical Processing) itself is an *"ante litteram"* smart interface, since it supports the users with a "point-and-click" metaphor to avoid writing well-formed SQL

*Corresponding author

Email addresses: `m.francia@unibo.it` (Matteo Francia), `enrico.gallinucci@unibo.it` (Enrico Gallinucci), `matteo.golfarelli@unibo.it` (Matteo Golfarelli)

queries. Nonetheless, the possibility of having a conversation with a smart assistant to run an OLAP session (i.e., a sequence of related OLAP queries) opens to new scenarios and applications. It is not just a matter of further reducing the complexity of posing a query: a conversational OLAP system must also provide feedback to refine and correct wrong queries, and it must have memory to relate subsequent requests. A reference application scenario for this kind of framework is *augmented business intelligence* [12], where hand-free interfaces are mandatory.

In this paper, we propose COOL, a COnversational OLap framework able to convert a natural language text into a GPSJ query and to support query disambiguation and OLAP navigation. GPSJ [17] is the main class of queries used in OLAP since it enables Generalized Projection, Selection, and Join operations over a set of tables. Although some natural language interfaces to databases have already been proposed, to the best of our knowledge this is the first proposal addressing full-fledged OLAP analytical sessions through a natural language interface.

In our vision, the desiderata for an OLAP smart interface are the following.

#1 It must be automated and portable: it must exploit cubes metadata (e.g., hierarchy structures, role of measures, attributes, and aggregation operators) to increase its understanding capabilities and to simplify the user-machine interaction process.

#2 It must handle OLAP sessions rather than single queries: in an OLAP session the first query is fully described by the text, while the following ones are implicitly/partially described by an OLAP operator (e.g., drill down, roll up, slice and dice) and require to handle the context and to have memory of the previous queries.

#3 It must be robust with respect to user inaccuracies in using syntax, OLAP terms, and attribute values; also, it must be able to exploit implicit information.

#4 It must be easy to configure on a data warehouse (DW) without a heavy manual definition of the lexicon.

More technically, our text-to-SQL approach is based on a grammar driving the parsing of natural language descriptions of GPSJ queries. The recognized entities include a set of typical query terms (e.g., group by, select) and the domain-specific terms and values automatically extracted from the DW (see desiderata #1 and #4). Robustness (desiderata #3) is one of the main goals for COOL and is pursued in all the interpretation steps: lexicon identification is based on a string similarity function, multi-word terms are handled through n-grams, and alternative query interpretations are scored and ranked. To sum up, the main contributions of this paper are:

1. a list of features and desiderata for an effective conversational OLAP system;

2. an original approach to translate a natural language analytical session into an OLAP session that starts with a well-formed GPSJ query (`Full query` step in Figure 1) and that refines the GPSJ query with known OLAP operators (`OLAP operator` step in Figure 1); in particular:
   - we discuss the architectural view of the approach;
   - we define a formal grammar for both full GPSJ queries and OLAP operators;
   - we analyze the specificities of natural language interfaces in the OLAP context;
   - we formalize the retrieval and resolution methods of OLAP-specific ambiguities in natural language queries.
3. a set of tests to verify the efficiency and effectiveness of our approach. In particular, we carried out tests with real users to assess how well (i.e., quick, simple, and accurate) COOL supports interactions.

Ultimately, COOL contributes to the democratization of data by extending the number of scenarios to carry out OLAP analysis (e.g., in augmented BI [12], natural language is the only means to express queries) and by reducing the skill level required to analyze the data. In particular, users are able to effectively carry out OLAP queries: (i) without any knowledge of querying languages like SQL or MDX, (ii) without any knowledge of complex analytical tools, and (iii) with minimum education about OLAP and the multidimensional model.

The remainder of the paper is organized as follows. Section 2 provides an overview of COOL by sketching the functional architecture and the interpretation steps. Section 3 presents the contents of the `MR`, while the following sections introduce the core steps within the `Intepretation` step, i.e., `Tokenization & Mapping` (Section 4), `Parsing` (Section 5), and `Checking & Annotation` (Section 6). The remaining steps `Disambiguation & Enhancement` and `SQL generation` are respectively discussed in Section 7 and Section 8. In Section 9, a large set of tests assesses the effectiveness, efficiency, and user experience of COOL. Section 10 discusses related works on natural language interface to database systems. Finally, Section 11 draws the conclusions and discusses the evolution of COOL.

2. Overview of COOL

Figure 1 sketches a functional view of the architecture. Given a `DW`, that is a set of multidimensional cubes together with their metadata, we distinguish between an offline phase (to initialize and configure the system) and an online phase (to enable the user interaction).

2.1. The Offline Phase

The *offline* phase extracts the DW-specific terms used by users to express the queries. Such information is stored in the metadata repository of COOL

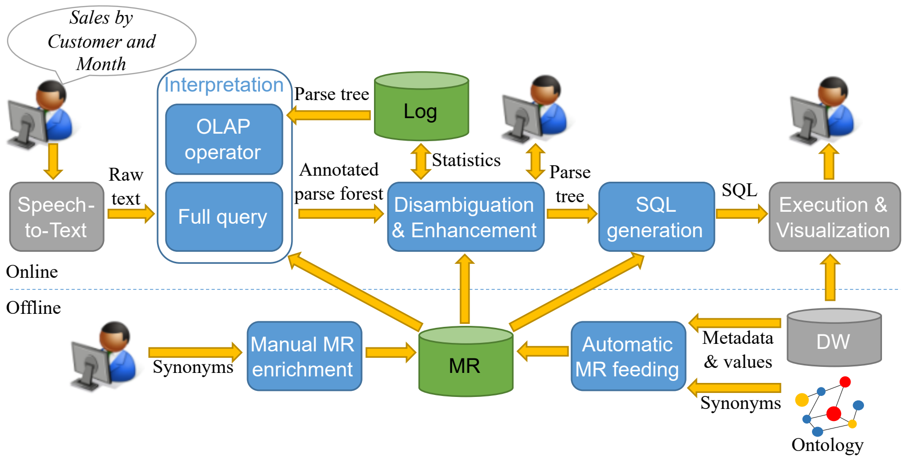


Figure 1: A functional architecture of COOL. Grayed-out elements are out of the paper scope.

(`MR`), which relies on the Dimensional Fact Model (DFM) expressiveness [15]. Noticeably, this phase runs only when the `DW` undergoes modification either in the cube schemas or in their instances. More in detail, the `Automatic MR feeding` process extracts the categorical attribute values and metadata from the cubes (e.g., attribute and measure names, table names, hierarchy structures) and possibly extends them with synonyms, automatically extracted from open data ontologies (Wordnet [26] in our implementation) to widen the language understood by COOL[1]. Besides the domain-specific terminology, the `MR` also includes the set of standard OLAP terms that are domain-independent and that do not require any feeding (e.g., group by, where, select). Further enrichment can be optionally carried out manually (i.e., by the `Manual MR enrichment` step) when the application domain involves a non-standard vocabulary (i.e., when the physical names of tables and columns do not match the words of a standard vocabulary). A closer look to the contents of the `MR` is given in Section 3.

2.2. The Online Phase

The *online* phase runs every time a natural language query is issued to COOL. The spoken query is initially translated to text by the `Speech-to-text` software module. This task is out of scope in our research and we exploited the public Web Speech API in our implementation (`https://wicg.github.io/speech-api/`).

The uninterpreted text is then analyzed by the `Interpretation` step that actually consists of two alternative steps: `Full query` is in charge of interpreting the texts describing full queries (which typically happens when an OLAP session starts), while `OLAP operator` modifies the latest query when the user states an

[1]The automatic procedure extracts every synonym in batch from Wordnet; thereafter, the imported synonyms can be updated or deleted through the manual procedure.

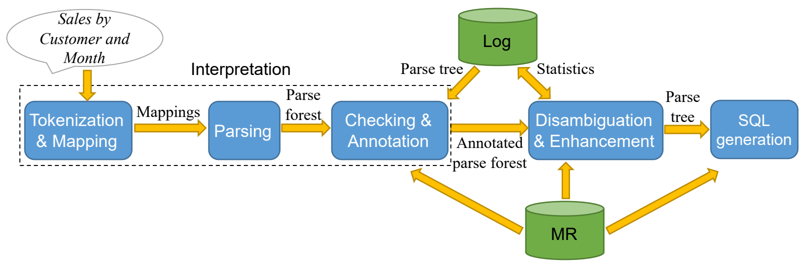


Figure 2: Interpretation of natural language; the `MR` is involved in all steps. The `Interpretation` steps are replicated for both `Full query` and `OLAP operator`.

OLAP operator during an OLAP session. The switch between the two steps to manage the conversation (i.e., a user/COOL dialog) is modeled by two states: *engage* and *navigate*.

- *Engage*: this is the initial state, in which the system expects a full query to be issued and whose interpretation is demanded to `Full query`. When COOL achieves a successful interpretation (i.e., it is able to run the query) it switches to the *navigate* state.

- *Navigate*: the dialogue evolves by iteratively applying OLAP operators that refine the query (i.e., which define an OLAP session). The management of these steps is demanded to `OLAP operator` until a *reset* command is applied, making COOL return to the *engage* state.

On one hand, understanding a single OLAP operator is simpler since it involves less elements than a complete GPSJ query. On the other hand, it requires to have memory of previous queries (stored in the `Log`) and to understand which part of the previous query must be modified. Both `Full query` and `OLAP operator` follow the computational steps represented in Figure 2: (i) `Tokenization & Mapping` (see Section 4), (ii) `Parsing` (see Section 5), and (iii) `Checking & Annotation` (see Section 6), but provide different implementations of (ii) and (iii).

Due to natural language ambiguities, speech-to-text inaccuracies and wrong query formulations (e.g., applying `count` operator on a measure or grouping by a descriptive attribute), part of the text can be misunderstood. The `Disambiguation & Enhancement` step solves ambiguities (if any) by asking appropriate questions to the user. The reasons behind the misunderstandings are manifold, including (but not limited to): ambiguities in the aggregation operator to be used; inconsistency between attribute and value in a selection predicate; identification of relevant elements in the text without understanding their role in the query.

The output of the previous steps is a data structure (i.e., a parse tree) that models the query and that can be automatically translated into an SQL query by exploiting the DW structure stored in the `MR`. Finally, the obtained query is run on the `DW` and the results are reported to the user by the `Execution`

`& Visualization` software module. Such a module could exploit a standard OLAP visualization tool or it could implement voice-based approaches [34] to create an end-to-end conversational solution. The visual interaction could rely on the DFM, which natively provides a graphical representation for multidimensional cubes and queries: such representation is conceptual and user-oriented, and its effectiveness is confirmed by its adoption in commercial tools [18] for both modeling and descriptive purposes. Although we have built a prototype to enable the evaluation of COOL (see Section 9), the discussion of this module is out of the scope of the paper.

3. The Metadata Repository

The Metadata Repository (`MR` in Figure 1) relies on the basic expressiveness of the DFM [15], which includes the following concepts.

- A *fact* (or *cube*) is a concept relevant to decision-making that typically models a set of events. Each event (or cube cell) is defined through a set of coordinates called *dimensions* that define the fact granularity.
- A *hierarchy* is a directed acyclic graph whose nodes are dimensional attributes and whose arcs model many-to-one relationships. A hierarchy is rooted in a dimension.
- A *dimensional attribute* (or simply *attribute*) describes a hierarchy. We can say that an attribute a is *finer/coarser* than an attribute b if there exists a path in the hierarchy from a to b (i.e., a rolls up to b) / b to a (i.e., b drills down to a). An attribute that stores relevant information but on which it makes no sense to aggregate on is called *descriptive attribute.*
- A *measure* is a numerical property of a fact and quantitatively describes an aspect of the fact. Each measure is associated with one or more *aggregation operators* that specify how to aggregate several cells.

The content of `MR` can be divided into *entities* and *structural information.* Entities compose the translated lexicon (i.e., the `Interpretation` step directly looks for their occurrence in the user's text), while structural information supports the interpretation (e.g., patterns necessary to recognize dates and numbers) and enables consistency checks on the interpreted query and the SQL generation (e.g., DW schema). More in detail, an entity $E = \langle t_1, ..., t_r \rangle$ is a sequence of textual words (i.e., a single distinct meaningful element of speech or writing). We refer to the set of *all* entities in the `MR` as $\mathcal{E} = \{E_1, ..., E_m\}$. Additionally, several synonyms can be stored for each entity (Table 1), enabling COOL to cope with slang and different shades of the text.

Orthogonally, entities and structural information are either *domain-agnostic* or *domain-dependent.* The domain-agnostic content includes those keywords and patterns that are typically used to express a query.

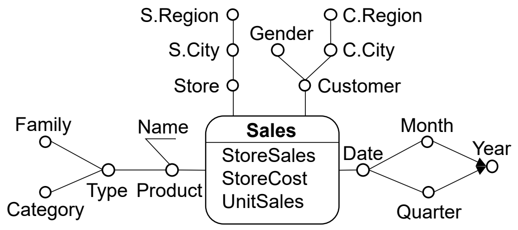


Figure 3: Simplified DFM representation of the Foodmart Sales cube.

- **Intention keywords**: entities expressing the role of the subsequent part of text. Examples of intention keywords are group by, select and where.
- **Operators**: entities including logic (not, and, or), comparison ($=, <>, >, <, \geq, \leq$) and aggregation operators (e.g., sum, avg, min, max).
- **Patterns of dates and numbers**: structures used to automatically recognize dates and numbers in raw text.

The MR domain-dependent content is automatically collected by querying the DW and its data dictionary and is stored in a QB4OLAP [9] compatible repository.

- **DW element names**: entities corresponding to measures, dimensional attributes and fact names.
- **DW element values**: entities corresponding to values from categorical attributes (together with their frequency), and to statistical values (e.g., minimum and maximum) for numerical attributes.
- **Hierarchy structure**: information about the roll up relationships between attributes.
- **Aggregation operators**: information about the operators applicable to each measure and the default one.
- **DB tables**: information about the structure of the database implementing the DW, including table and attribute names, primary and foreign key relationships.

Example 1 (Cube, Entities, and Synonyms). *In the* Sales *fact schema in Figure 3,* Product *and* Store *are dimensions,* Month *is a dimensional attribute and* Name *is a descriptive attribute.* StoreSales *and* UnitSales *are measures. It is possible to aggregate* StoreSales *using* sum *and* avg *aggregation operators. An example of GPSJ query would ask to "return the total quantity sold by month and type only for Italian stores". Drilling down from* Type *to* Product *means grouping on a finer attribute. Conversely rolling up from* Month *to* Year *means grouping on a coarser attribute. Finally, we can further slice and dice by adding*

Table 1: Sample of domain-agnostic and domain-specific entities with their synonyms. Domain-specific entities refer to the Sales cube from Figure 3.

Domain	**Type**	**Entity**	**Synonym samples**
Agnostic	Intention keyword	where	in, on, such that, filter
		group by	by, for each, per
	Operator	$=$; $\geq$	equal to; greater than
		sum	total, amount
		avg	average, medium
Specific	DW element name	Sales	transactions
		UnitSales	quantities
		Gender	sex
	DW element value	*Drink*	beverage

a filter on a specific Category *of products. With reference to the* Sales *cube,* Month *and* UnitSales *are domain-specific entities, while* avg *is a domain-agnostic entity. Examples of synonyms for* avg *are "average" and "medium".* □

4. Tokenization & Mapping

A raw text T can be modeled as a sequence of tokens (i.e., single words) $T = \langle t_1, ..., t_z \rangle$. The goal of this step is to identify in T the entities, i.e., the only elements that will be involved in the `Parsing` step. Turning a text into a sequence of entities means finding a *mapping* between tokens in T and $\mathcal{E}$.

Definition 1 (Mapping function & Mapped sequence). *A mapping function $M(T)$ is a partial function that associates sub-sequences (or n-*grams*)*[2] *from T to entities in $\mathcal{E}$ such that:*

- *sub-sequences of T have length n at most;*
- *the mapping function determines a partitioning of T;*
- *a sub-sequence $T' = \langle t_i, ..., t_l \rangle \in T$ (with $|T'| \leq n$) is associated with an entity E if and only if $Sim(T', E) > \alpha$ (where $Sim()$ is a similarity function, later defined) and $E \in TopN(\mathcal{E}, T')$ (where $TopN(\mathcal{E}, T')$ is the set of N entities in $\mathcal{E}$ that are the most similar to T' according to $Sim(T', E)$).*

The output of a mapping function is a sequence $M = \langle E_1, ..., E_l \rangle$ on $\mathcal{E}$ that we call a target sequence. *A target sequence is said to be* valid *if the fraction of mapped tokens in T is higher than a given threshold β. We call $\mathcal{M}$ the set of valid target sequences.*

[2]The term n-gram is used as a synonym of sub-sequence in the area of text mining.

Several target sequences might be obtained from T since Definition 1 admits sub-sequences of variable lengths (corresponding to different partitionings of T) and associates the top similar entities to each sub-sequence. This increases interpretation robustness (since it allows to choose, in the next steps, the *best text interpretation* out of a higher number of candidates), but it can lead to an increase in computation time. The generated target sequences differ both in the number of entities involved and in the specific entities mapped to a token. In the simple case where multi-token mappings are not possible (i.e., $n = 1$ in Definition 1) the number of generated target sequences for a raw text T, such that $|T| = z$, is:

$$\sum_{i=\lceil z\cdot\beta\rceil}^{z} \binom{z}{i} \cdot N^i \tag{1}$$

The formula counts the possible configurations of sufficient length (i.e., higher or equal to $\lceil z\cdot\beta\rceil$) and, for each length, counts the target sequences determined by the top similar entities. Since the number of candidate target sequences is exponential, we consider only the most significant ones through α, β, and N: α imposes sub-sequence of tokens to be very similar to an entity; N further imposes to consider only the N entities with the highest similarity; finally, β imposes a sufficient portion of the text to be mapped.

The similarity function $Sim()$ is based on the Levenshtein distance (i.e., one of the most used and representative character-based distance functions [37]) and keeps token permutation into account to increase its robustness (e.g., sub-sequences $\langle P., Edgar\rangle$ and $\langle Edgar, Allan, Poe\rangle$ must result similar). Given two token sequences T and W with $|T| = l, |W| = m$ such that $l \leq m$ it is:

$$Sim(\langle t_1, ..., t_l\rangle, \langle w_1, ..., w_m\rangle) = \max_{D\in Disp(l,m)} \frac{\sum_{i=1}^{l} sim(t_i, w_{D(i)}) \cdot \max(|t_i|, |w_{D(i)}|)}{\sum_{i=1}^{l} \max(|t_i|, |w_{D(i)}|) + \sum_{i\in\hat{D}} |w_i|}$$

where $w_{D(i)}$ is a token from W at the index $D(i)$, $D \in Disp(l,m)$ is an l-disposition of $\{1, ..., m\}$, $\hat{D}$ is the subset of values in $\{1, ..., m\}$ that are not present in D, and $sim(t_i, w_{D(i)})$ is the similarity between the pair of tokens, calculated as $1-NLD(t_i, w_{D(i)})$, with $NLD()$ being the normalized Levenshtein distance [21]. Function $Sim()$ weights token similarity based on their lengths (i.e., $\max(|t_i|, |w_{D(i)}|)$) and penalizes similarities between sequences of different lengths that imply unmatched tokens (i.e., $\sum_{i\in\hat{D}} |w_i|$).

Example 2 (Token similarity). *Figure 4 shows some of the possible token dispositions for the two token sequences $T = \langle P., Edgar\rangle$ and $W = \langle Edgar, Allan, Poe\rangle$. The disposition determining the highest similarity is surrounded by a dashed rectangle; the similarity is* 0.46 *and it is calculated as*

$$Sim(T, W) = \frac{sim(P., Poe)|Poe| + sim(Edgar, Edgar)|Edgar|}{|Poe| + |Edgar| + |Allan|}$$

□

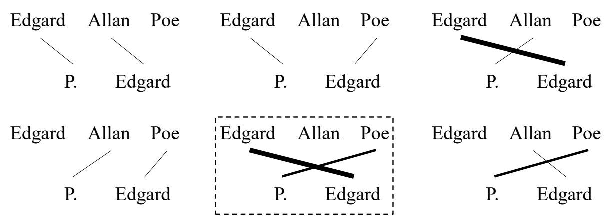


Figure 4: Token dispositions: arcs denote the correspondence of tokens for a specific disposition (the bolder the line, the higher the similarity). The disposition determining the maximum similarity is shown in a dashed rectangle.

We assume the best interpretation of the input text to be the one where (1) all the entities discovered in the text are included in the query (i.e., all the entities are parsed through the grammar), and (2) each entity discovered in the text is perfectly mapped to one sub-sequence of tokens T (i.e., $Sim(T, E_i) = 1$). The two previous statements are modeled through the following score function. Given a target sequence $M = \langle E_1, ..., E_m \rangle$, we define its score as

$$Score(M) = \sum_{i=1}^{m} Sim_M(E_i) \tag{2}$$

where $Sim_M(E_i)$ is an abbreviation to indicate the similarity between E_i and the corresponding sub-sequence of tokens T on M.

The score is higher when M includes several entities with high values of Sim_M. Although at this stage it is not possible to predict if a target sequence will be fully parsed, it is apparent that the higher is its score, the higher is the probability to determine an optimal interpretation. As it will be explained in Section 6, sorting the target sequences by descending score also enables pruning strategies to be applied.

Example 3 (Tokenization & Mapping). *With reference to Table 1, given the set of entities $\mathcal{E}$ and a tokenized text $T = \langle medium, sales, in, 2019, by, the, region \rangle$, examples of target sequences M_1 and M_2 are:*

$$M_1 = \langle \mathsf{avg}, \mathsf{UnitSales}, \mathsf{where}, 2019, \mathsf{group\ by}, \mathsf{region} \rangle$$
$$M_2 = \langle \mathsf{avg}, \mathsf{UnitSales}, \mathsf{where}, 2019, \mathsf{group\ by}, Regin \rangle$$

where "medium" is mapped to avg *since "medium" is a known synonym of the aggregation operator* avg*, "in" is mapped to* where *since "in" can express time-related predicates, "the" is discarded being a stop word, and "region" is mapped to the attribute* Region *in M_1 and to the value "Regin" in M_2 (where "Regin" is a value of attribute* Customer *that holds a sufficient similarity).* □

5. Parsing

Parsing is the process of analyzing a sequence of entities (i.e., a target sequence) to determine its syntactical structure with respect to a formal grammar. Parsing outputs a data structure called parse tree that is used by COOL to translate a target sequence into SQL.

5.1. Full Query Parsing

In `Full query`, `Parsing` is responsible for the interpretation of a complete GPSJ query stated in natural language. Parsing a full query means searching in a target sequence the complex syntax structures (i.e., *clauses*) that build-up the query. Given a target sequence M, the output of a parser is a *parse tree* PT_M, i.e. an ordered tree that represents the syntactic structure of the target sequence according to the grammar described in Figure 5. To the aim of parsing, entities are terminal elements in the grammar.

As a GPSJ query consists of 3 clauses (measure, group by, and selection), in our grammar we identify four types of derivations[3]:

- **Measure clause** $\langle \mathsf{MC} \rangle$: this derivation consists of a list of measure/aggregation operator pairs.
- **Group by clause** $\langle \mathsf{GC} \rangle$: this derivation consists of a sequence of attribute names preceded by the entity "*group by*".
- **Selection clause** $\langle \mathsf{SC} \rangle$: this derivation consists of a Boolean expression of simple selection predicates $\langle \mathsf{SSC} \rangle$ preceded by the entity "*where*".
- **GPSJ query** $\langle \mathsf{GPSJ} \rangle$: this derivation assembles the final query. Only the measure clause is mandatory since a GPSJ could aggregate a single measure with no selections. The order of the clauses is irrelevant; this implies the proliferation of derivations due to permutations.

As shown in Figure 5, some derivations admit also partial forms (e.g., a measure clause $\langle \mathsf{MC} \rangle$ missing its aggregation operator $\langle \mathsf{Agg} \rangle$); in these cases, the `Disambiguation & Enhancement` step will try to infer the omitted derivations (see Section 6).

The GPSJ grammar is LL(1)[4] [4], is not ambiguous (i.e., each target sequence admits, at most, a single parse tree PT_M) and can be parsed by a LL(1) parser with linear complexity [4]. If the input target sequence M is fully parsed, PT_M includes all the entities as leaves. Conversely, if only a portion of the input belongs to the grammar, an LL(1) parser produces a partial parsing, meaning that it returns a parse tree including the portion of the input target sequence that belongs to the grammar (i.e., the PT rooted in $\langle \mathsf{GPSJ} \rangle$). The remaining entities can be either singletons or complex clauses that were not possible to

[3]A derivation in the form $\langle \mathsf{X} \rangle ::= e$ represent a substitution for the non-terminal symbol $\langle \mathsf{X} \rangle$ with the given expression e. Symbols that never appear on the left side of ::= are named terminals. Non-terminal symbols are enclosed between ⟨⟩, while terminal symbols are enclosed between " ".

[4]The rules presented in Figure 5 do not satisfy LL(1) constraints for readability reasons. It is easy to turn such rules into an LL(1) compliant version, but the resulting rules are much harder to be read and understood.

$$
\begin{array}{rcl}
\langle\mathsf{GPSJ}\rangle & ::= & \langle\mathsf{MC}\rangle\langle\mathsf{GC}\rangle\langle\mathsf{SC}\rangle \mid \langle\mathsf{MC}\rangle\langle\mathsf{SC}\rangle\langle\mathsf{GC}\rangle \mid \langle\mathsf{SC}\rangle\langle\mathsf{GC}\rangle\langle\mathsf{MC}\rangle \mid \langle\mathsf{SC}\rangle\langle\mathsf{MC}\rangle\langle\mathsf{GC}\rangle \\
 & & \mid \langle\mathsf{GC}\rangle\langle\mathsf{SC}\rangle\langle\mathsf{MC}\rangle \mid \langle\mathsf{GC}\rangle\langle\mathsf{MC}\rangle\langle\mathsf{SC}\rangle \mid \langle\mathsf{MC}\rangle\langle\mathsf{SC}\rangle \mid \langle\mathsf{MC}\rangle\langle\mathsf{GC}\rangle \\
 & & \mid \langle\mathsf{SC}\rangle\langle\mathsf{MC}\rangle \mid \langle\mathsf{GC}\rangle\langle\mathsf{MC}\rangle \mid \langle\mathsf{MC}\rangle \\
\langle\mathsf{MC}\rangle & ::= & (\langle\mathsf{Agg}\rangle\langle\mathsf{Mea}\rangle \mid \langle\mathsf{Mea}\rangle\langle\mathsf{Agg}\rangle \mid \langle\mathsf{Mea}\rangle \mid \langle\mathsf{Cnt}\rangle\langle\mathsf{Fct}\rangle \mid \langle\mathsf{Fct}\rangle\langle\mathsf{Cnt}\rangle \\
 & & \mid \langle\mathsf{Cnt}\rangle\langle\mathsf{Attr}\rangle \mid \langle\mathsf{Attr}\rangle\langle\mathsf{Cnt}\rangle)+ \\
\langle\mathsf{GC}\rangle & ::= & \text{“}\textit{group by}\text{”}\ \langle\mathsf{Attr}\rangle+ \\
\langle\mathsf{SC}\rangle & ::= & \text{“}\textit{where}\text{”}\ \langle\mathsf{SCA}\rangle \\
\langle\mathsf{SCA}\rangle & ::= & \langle\mathsf{SCN}\rangle\ \text{“}\textit{and}\text{”}\ \langle\mathsf{SCA}\rangle \mid \langle\mathsf{SCN}\rangle \\
\langle\mathsf{SCN}\rangle & ::= & \text{“}\textit{not}\text{”}\ \langle\mathsf{SSC}\rangle \mid \langle\mathsf{SSC}\rangle \\
\langle\mathsf{SSC}\rangle & ::= & \langle\mathsf{Attr}\rangle\langle\mathsf{Cop}\rangle\langle\mathsf{Val}\rangle \mid \langle\mathsf{Val}\rangle\langle\mathsf{Cop}\rangle\langle\mathsf{Attr}\rangle \mid \langle\mathsf{Attr}\rangle\ \text{“}\textit{in}\text{”}\ \langle\mathsf{Val}\rangle+ \\
 & & \mid \langle\mathsf{Val}\rangle+\ \text{“}\textit{in}\text{”}\ \langle\mathsf{Attr}\rangle \mid \langle\mathsf{Attr}\rangle\langle\mathsf{Val}\rangle+ \mid \langle\mathsf{Val}\rangle+\langle\mathsf{Attr}\rangle \mid \langle\mathsf{Val}\rangle \\
\langle\mathsf{Cop}\rangle & ::= & \text{“}=\text{”} \mid \text{“}<>\text{”} \mid \text{“}>\text{”} \mid \text{“}<\text{”} \mid \text{“}\geq\text{”} \mid \text{“}\leq\text{”} \\
\langle\mathsf{Agg}\rangle & ::= & \text{“}\textit{sum}\text{”} \mid \text{“}\textit{avg}\text{”} \mid \text{“}\textit{min}\text{”} \mid \text{“}\textit{max}\text{”} \mid \text{“}\textit{stdev}\text{”} \\
\langle\mathsf{Cnt}\rangle & ::= & \text{“}\textit{count}\text{”} \mid \text{“}\textit{count distinct}\text{”} \\
\langle\mathsf{Fct}\rangle & ::= & \textit{Domain-specific facts} \\
\langle\mathsf{Mea}\rangle & ::= & \textit{Domain-specific measures} \\
\langle\mathsf{Attr}\rangle & ::= & \textit{Domain-specific attributes} \\
\langle\mathsf{Val}\rangle & ::= & \textit{Domain-specific values}
\end{array}
$$

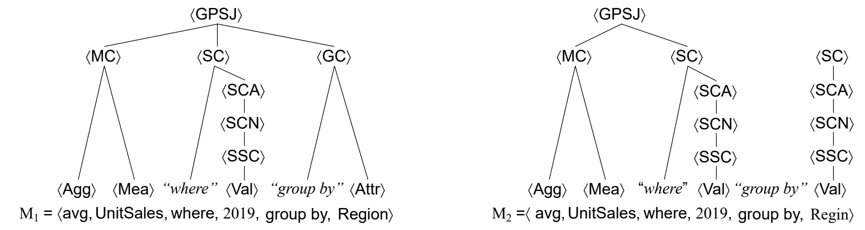
Figure 5: Backus-Naur representation of the `Full query` grammar. Entities from the `MR` are terminal symbols. "+" identifies a list of at least 1 symbol.

connect to the main parse tree. We will call *parse forest* PF_M the union of the parse tree with residual clauses. Obviously, if all the entities are parsed, it is $PF_M = PT_M$. Considering the whole forest rather than the simple parse tree enables disambiguation and errors to be recovered in the `Disambiguation & Enhancement` step (see Section 7). To keep the terminology simple, we will refer to the parser's output as a parse forest independently of the presence of residual clauses.

Example 4 (Parsing). *Figure 6 reports the parsing outcome for the two target sequences in Example 3. M_1 is fully parsed, thus its parse forest corresponds to the parse tree (i.e., $PT_{M_1} = PF_{M_1}$). Conversely, in M_2 the last token is wrongly mapped to the attribute value* Regin *rather than to the attribute name* Region. *This prevents the full parsing and the parse tree PT_{M_2} does not include all the entities in M_2 (i.e., $PT_{M_2} \neq PF_{M_2}$).* □

5.2. OLAP Operator Parsing

In `OLAP operator`, `Parsing` is responsible for searching in a target sequence the syntactic structures of the OLAP operators that build-up the conversation.


⟨GPSJ⟩
⟨MC⟩
⟨SC⟩
⟨GC⟩
⟨SCA⟩
⟨SCN⟩
⟨SSC⟩
⟨Agg⟩ ⟨Mea⟩ "where" ⟨Val⟩ "group by" ⟨Attr⟩
M1 = ⟨avg, UnitSales, where, 2019, group by, Region⟩
⟨GPSJ⟩
⟨MC⟩
⟨SC⟩
⟨SC⟩
⟨SCA⟩
⟨SCN⟩
⟨SSC⟩
⟨Agg⟩ ⟨Mea⟩ "where" ⟨Val⟩ "group by" ⟨Val⟩
M2 =⟨ avg, UnitSales, where, 2019, group by, Regin⟩

(a) The entire target sequence is parsed (i.e., $PF_M = PT_M$).

(b) "*group by*" ⟨Val⟩ cannot be parsed.

Figure 6: Parse forests from Example 3.

$$
\begin{aligned}
\langle \mathsf{OPERATOR} \rangle ::= {} & \langle \mathsf{DRILL} \rangle \mid \langle \mathsf{ROLLUP} \rangle \mid \langle \mathsf{SAD} \rangle \mid \langle \mathsf{ADD} \rangle \mid \langle \mathsf{DROP} \rangle \\
& \mid \langle \mathsf{REPLACE} \rangle \\
\langle \mathsf{DRILL} \rangle ::= {} & \text{“}\textit{drill}\text{”}\ \langle \mathsf{Attr} \rangle_{from}\ \text{“}\textit{to}\text{”}\ \langle \mathsf{Attr} \rangle_{to} \mid \text{“}\textit{drill}\text{”}\ \langle \mathsf{Attr} \rangle \\
\langle \mathsf{ROLLUP} \rangle ::= {} & \text{“}\textit{rollup}\text{”}\ \langle \mathsf{Attr} \rangle_{from}\ \text{“}\textit{to}\text{”}\ \langle \mathsf{Attr} \rangle_{to} \mid \text{“}\textit{rollup}\text{”}\ \langle \mathsf{Attr} \rangle \\
\langle \mathsf{SAD} \rangle ::= {} & \text{“}\textit{slice}\text{”}\ \langle \mathsf{SSC} \rangle \\
\langle \mathsf{ADD} \rangle ::= {} & \text{“}\textit{add}\text{”}\ (\langle \mathsf{MC} \rangle \mid \langle \mathsf{Attr} \rangle \mid \langle \mathsf{SSC} \rangle) \\
\langle \mathsf{DROP} \rangle ::= {} & \text{“}\textit{drop}\text{”}\ (\langle \mathsf{MC} \rangle \mid \langle \mathsf{Attr} \rangle \mid \langle \mathsf{SSC} \rangle) \\
\langle \mathsf{REPLACE} \rangle ::= {} & \text{“}\textit{replace}\text{”}\ (\langle \mathsf{MC} \rangle_{old}\ \text{“}\textit{with}\text{”}\ \langle \mathsf{MC} \rangle_{new} \\
& \mid \langle \mathsf{Attr} \rangle_{old} \text{“}\textit{with}\text{”}\ \langle \mathsf{Attr} \rangle_{new} \mid \langle \mathsf{SSC} \rangle_{old}\ \text{“}\textit{with}\text{”}\ \langle \mathsf{SSC} \rangle_{new})
\end{aligned}
$$

Figure 7: Backus-Naur representation of the `OLAP operator` grammar. Entities from the `MR` are terminal symbols. We omit the derivations ⟨MC⟩, ⟨Attr⟩, ⟨SSC⟩ that are in common with Figure 5. Decoration of non-terminals with subscript is used for the sake of description.

Our conversation steps are inspired by well-known OLAP visual interfaces (e.g., Tableau[5]). To apply an OLAP operator, COOL must be in state *navigate* (i.e., a full GPSJ query has been already successfully interpreted). By definition, the previously interpreted query corresponds to a parse tree PT_C that acts as a context for the operator; in particular, PT_C is used by the `Checking & Annotation` step to verify the consistency of the OLAP operator and by the `Disambiguation & Enhancement` step to implicitly solve ambiguities. For the sake of explanation, we assume that PT_C takes the form ⟨GPSJ⟩ ::= ⟨MC⟩⟨GC⟩⟨SC⟩ (even if ⟨GC⟩ and/or ⟨SC⟩ can be missing), and with a slight abuse of notation we adopt the set notation to denote that a clause is contained in another clause (e.g., ⟨Attr⟩ ∈ ⟨GC⟩ and ⟨GC⟩ ∈ ⟨GPSJ⟩).

The whole grammar is described in Figure 7 (we do not report grammar derivations ⟨Attr⟩, ⟨MC⟩ and ⟨SSC⟩ in common with Figure 5) and introduces the following derivations.

- **Drill down** ⟨DRILL⟩: this derivation substitutes the coarser attribute

[5]`https://www.tableau.com/`

$\langle \mathsf{Attr} \rangle_{from} \in \langle \mathsf{GC} \rangle$ with a finer attribute $\langle \mathsf{Attr} \rangle_{to}$.

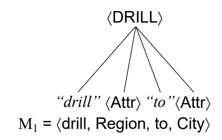
- **Roll up** $\langle \mathsf{ROLLUP} \rangle$: this derivation substitutes the finer attribute $\langle \mathsf{Attr} \rangle_{from} \in \langle \mathsf{GC} \rangle$ with a coarser attribute $\langle \mathsf{Attr} \rangle_{to}$.
- **Slice and dice** $\langle \mathsf{SAD} \rangle$: this derivation specifies a new selection predicate $\langle \mathsf{SSC} \rangle \in \langle \mathsf{SC} \rangle$.
- **Add** $\langle \mathsf{ADD} \rangle$: this derivation adds a measure/attribute/selection clause to the corresponding clause $\langle \mathsf{MC} \rangle / \langle \mathsf{GC} \rangle / \langle \mathsf{SC} \rangle \in \langle \mathsf{GPSJ} \rangle$.
- **Drop** $\langle \mathsf{DROP} \rangle$: this derivation drops a measure/attribute/selection clause from the corresponding clause $\langle \mathsf{MC} \rangle / \langle \mathsf{GC} \rangle / \langle \mathsf{SC} \rangle \in \langle \mathsf{GPSJ} \rangle$.
- **Replace** $\langle \mathsf{REPLACE} \rangle$: this derivation combines **Add** and **Drop** to substitute an existing *old* clause, either measure, attribute or selection, for a *new* one.

Similarly to the GPSJ grammar (Figure 5), the OLAP operator grammar admits partial forms for **Drill down** and **Roll up** (e.g., only a single attribute $\langle \mathsf{Attr} \rangle$ is provided in a $\langle \mathsf{DRILL} \rangle$ clause) that the `Checking & Annotation` step will try to complete (see Section 7). Also note that the **Add** and **Drop** operators respectively behave as **Drill down** and **Roll up** when applied to attributes; otherwise, they behave as **Slice and dice** when applied to selection clauses. Finally, when **Slice and dice** or **Add** refer to an already existing selection clause, we append the new member to such clause (if needed, we replace the comparison operator " = " with "*in*").

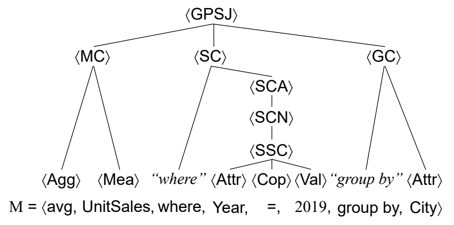
Example 5 (Conversational OLAP Session). *Given the parse tree from Figure 6a, an example of conversation is the following (Figure 8). At first the user issues the natural language sentence "Drill down region to city", COOL parses such sentence (Figure 8a), recognizes a drill down operation and modifies the previous parse tree (Figure 8b). Then, the user asks to "replace the unit sales with the sum of store sales". Despite the aggregation operation is not specified, COOL recognizes the measure clause (Figure 8c) to be substituted and replaces it with the new one (Figure 8d). Finally, the user asks to "slice on milk" (Figure 8e): COOL automatically infers the attribute containing the value "milk" and combines the new condition with the previous existing clause (Figure 8f).* □

6. Parse Forest Checking and Annotation

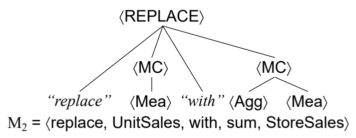
The `Parsing` step does not always output a parse forest that can be directly translated into executable SQL code. Indeed, syntactic adherence to the grammar does not guarantee the conformance of the full query (or OLAP operator) with multidimensional structure and constraints. As it happens for compilers, parsing, and type checking (i.e., verifying and enforcing the constraints of data types) are kept separated to reduce the overall complexity.



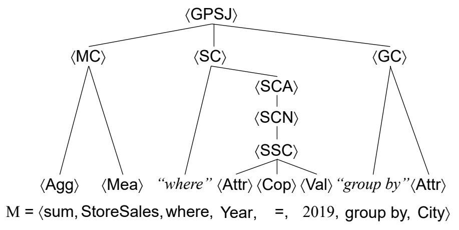
(a) Parse "Drill down region to city".


⟨GPSJ⟩
⟨MC⟩ ⟨SC⟩ ⟨GC⟩
⟨SCA⟩
⟨SCN⟩
⟨SSC⟩
⟨Agg⟩ ⟨Mea⟩ "where" ⟨Attr⟩ ⟨Cop⟩ ⟨Val⟩ "group by" ⟨Attr⟩
M = ⟨avg, UnitSales, where, Year, =, 2019, group by, City⟩

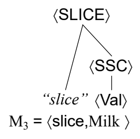
(b) Drill down Region to City w.r.t. Figure 6a.


⟨REPLACE⟩
⟨MC⟩ ⟨MC⟩
"replace" ⟨Mea⟩ "with" ⟨Agg⟩ ⟨Mea⟩
M₂ = ⟨replace, UnitSales, with, sum, StoreSales⟩

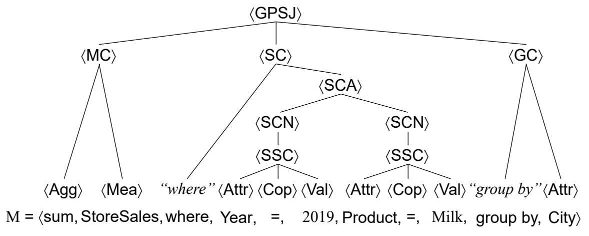
(c) Parse "Replace unit sales with the sum of store sales".


⟨GPSJ⟩
⟨MC⟩ ⟨SC⟩ ⟨GC⟩
⟨SCA⟩
⟨SCN⟩
⟨SSC⟩
⟨Agg⟩ ⟨Mea⟩ "where" ⟨Attr⟩ ⟨Cop⟩ ⟨Val⟩ "group by" ⟨Attr⟩
M = ⟨sum, StoreSales, where, Year, =, 2019, group by, City⟩

(d) Replace Avg UnitSales with Sum StoreSales.



(e) Parse "Slice on milk".


⟨GPSJ⟩
⟨MC⟩ ⟨SC⟩ ⟨GC⟩
⟨SCA⟩
⟨SCN⟩ ⟨SCN⟩
⟨SSC⟩ ⟨SSC⟩
⟨Agg⟩ ⟨Mea⟩ "where" ⟨Attr⟩ ⟨Cop⟩ ⟨Val⟩ ⟨Attr⟩ ⟨Cop⟩ ⟨Val⟩ "group by" ⟨Attr⟩
M = ⟨sum, StoreSales, where, Year, =, 2019, Product, =, Milk, group by, City⟩

(f) Add a new slice on Product = $Milk$.

Figure 8: A conversational session modifies the parse tree from Figure 6a.

In this step, COOL seeks for these problems, and annotates the parse forest accordingly. Two types of annotations are possible.

- ***Ambiguity***: an inconsistency solvable through disambiguation. Ambiguities are solvable either automatically by COOL or by interacting with the user (e.g., *"sum unit sales for Salem"*, but *Salem* is member of both City and StoreCity);
- ***Error***: an inconsistency that does not admit solution and that can be only notified to the user (e.g., *"remove unit sales"*, but the measure UnitSales is not included in ⟨GPSJ⟩).

`Checking & Annotation` is responsible for searching problematic clauses (i.e., subtrees) in the parse forest and annotating them. Depending on the type of the clause, `Checking & Annotation` evaluates the conformance of the clause to the multidimensional structure and constraints and marks the subtrees failing such constraints.

We now describe the allowed annotations for the parse forests produced by `Full query` and `OLAP operator`.

Table 2: `Full query` annotations.

Type	**Name**	**Gen. derivation example**	**Example**
Ambiguity	AA	⟨SSC⟩ ::= ⟨Val⟩	*"sum unit sales for Salem"*, but *Salem* is member of City and StoreCity
Ambiguity	AAO	⟨MC⟩ ::= ⟨Mea⟩	*"unit sales by product"*, but sum and avg are valid aggregations
Ambiguity	AVM	⟨SSC⟩ ::= ⟨Attr⟩⟨Cop⟩⟨Val⟩	*"sum unit sales for product New York"*, but *New York* is not a Product
Ambiguity	MV	⟨MC⟩ ::= ⟨Agg⟩⟨Mea⟩	*"sum prices per store"*, but Price is not additive
Ambiguity	AV	⟨GC⟩ ::= *"group by"* ⟨Attr⟩+	*"average prices by name"*, but Product is not in ⟨GC⟩
Ambiguity	UC	–	*"average unit sales by Regin"*, but *Regin* is not an attribute

6.1. Full Query Annotation

In `Full query`, `Checking & Annotation` searches PT for problematic clauses (i.e., subtrees) in a depth-first fashion [33]. Table 2 reports the annotations resulting from failing checks.

- **Ambiguous attribute (AA)**: the ⟨SSC⟩ clause has an implicit attribute but the parsed value belongs to multiple attribute domains.
- **Ambiguous aggregation operator (AAO)**: the ⟨MC⟩ clause has an implicit aggregation operator but the measure is associated with multiple aggregation operators.
- **Attribute-value mismatch (AVM)**: the ⟨SSC⟩ clause includes a value that does not belong to the domain of the specified attribut.
- **Violation of a multidimensional constraint on a measure (MV)**: the ⟨MC⟩ clause contains an aggregation operator that is not allowed for the specified measure.
- **Violation of a multidimensional constraint on an attribute (AV)**: the ⟨GC⟩ clause contains a descriptive attribute without the corresponding dimensional attribute[6].
- **Unparsed clause (UC)**: A clause has been properly parsed, but the parser was not able to connect it to the ⟨GPSJ⟩ derivation. Thus, a parse forest including a ⟨GPSJ⟩ derivation and one or more dangling clauses has been returned.

6.2. OLAP Operator Annotation

Although the `OLAP operator` grammar returns simpler parse trees than `Full query`, annotating an OLAP operator is more complex since the parse tree must be internally coherent and *also* compliant to the previous query context, i.e. the

[6]According to DFM a *descriptive attribute* is an attribute that further describes a dimensional level (i.e., it is related one-to-one with the level), but that can be used for aggregation only in combination with the corresponding level.

latest full query to which the OLAP operator should be applied. Let PT_C be the parse tree for the previous query context, and let PT be the parse tree of the OLAP operator. In `OLAP operator`, `Checking & Annotation` extends the checks in Section 6.1 by searching also for possible inconsistencies between each clause in PT_C and PT. Checking is achieved in a depth-first fashion [33] and exploits the PT_C structure to reduce the search space (e.g., it is meaningful to search for a $\langle \mathsf{SSC} \rangle$ only within the $\langle \mathsf{SC} \rangle \in \langle \mathsf{GPSJ} \rangle$). As a consequence, checking an OLAP operator requires additional constraints and annotations with respect to the ones defined in Section 6.1 (Table 3):

- **Not Existing Clause (NEC)**: PT references a clause that should be present in PT_C, but that is missing. For example, $\langle \mathsf{Attr} \rangle_{from}$ in the $\langle \mathsf{DRILL} \rangle$ and $\langle \mathsf{ROLLUP} \rangle$ derivations must exist in $\langle \mathsf{GC} \rangle$ of PT_C;
- **Already Existing Clause (AEC)**: PT cannot be applied as it references an element that is already present in PT_C, while it should not be there. For example, $\langle \mathsf{Attr} \rangle_{to}$ in the $\langle \mathsf{DRILL} \rangle$ and $\langle \mathsf{ROLLUP} \rangle$ derivations cannot exist in $\langle \mathsf{GC} \rangle$ of PT_C;
- **Invalid drill (ID)**: it is impossible to drill down on $\langle \mathsf{Attr} \rangle_{from}$ in a $\langle \mathsf{DRILL} \rangle$ derivation, since $\langle \mathsf{Attr} \rangle_{from}$ is already the finest attribute in the hierarchy.
- **Forbidden Attribute (FA)**: it is impossible to drill down/roll up on a descriptive attribute.
- **Hierarchy mismatch (HM)**: $\langle \mathsf{Attr} \rangle_{from}$ and $\langle \mathsf{Attr} \rangle_{to}$ in the $\langle \mathsf{DRILL} \rangle$ (or $\langle \mathsf{ROLLUP} \rangle$) derivation belong to two different hierarchies.
- **Hierarchy structure mismatch (HSM)**: $\langle \mathsf{Attr} \rangle_{from}$ in the $\langle \mathsf{DRILL} \rangle$ (or $\langle \mathsf{ROLLUP} \rangle$) derivation is finer (or coarser) than $\langle \mathsf{Attr} \rangle_{to}$.
- **Branching ambiguity (TO)**: it is impossible to infer $\langle \mathsf{Attr} \rangle_{to}$ in a $\langle \mathsf{DRILL} \rangle$ (or $\langle \mathsf{ROLLUP} \rangle$) derivation due to the presence of a branch in the hierarchy.
- **Rolling ambiguity (FR)**: it is impossible to infer $\langle \mathsf{Attr} \rangle_{from}$ in a $\langle \mathsf{DRILL} \rangle$ (or $\langle \mathsf{ROLLUP} \rangle$) derivation due to the presence of multiple finer (or coarser) attributes in $\langle \mathsf{GC} \rangle$.

6.3. Parse Forest Scoring

A textual query generates several parse forests, one for each target sequence. In our approach, only the most promising one is proposed to the user in the `Disambiguation & Enhancement` step. This choice comes from two main motivations:

- Proposing more than one alternative queries to the user can be confusing and makes it very difficult to contextualize the disambiguation questions.

Table 3: `OLAP operator` annotations.

Type	**Name**	**Gen. derivation example**	**Example**
Ambiguity	TO	⟨DRILL⟩ ::= "*drill*" ⟨Attr⟩	*"drill down from year"*
Ambiguity	FR	⟨DRILL⟩ ::= "*drill*" ⟨Attr⟩	*"drill down to year"* but ⟨GC⟩ contains Month and Quarter
Error	NEC	⟨DROP⟩ ::= "*drop*" ⟨MC⟩	*"remove unit sales"* but UnitSales is not in ⟨GPSJ⟩
Error	AEC	⟨ADD⟩ ::= "*add*" ⟨MC⟩	*"add max unit sales"* but max UnitSales is already in ⟨GPSJ⟩
Error	ID	⟨DRILL⟩ ::= "*drill*" ⟨Attr⟩	*"drill down from product"*
Error	FA	⟨DRILL⟩ ::= "*drill*" ⟨Attr⟩	*"drill down from name"*
Error	HM	⟨DRILL⟩ ::= "*drill*" ⟨Attr⟩$_{from}$ "*to*" ⟨Attr⟩$_{to}$	*"drill down from product to city"*
Error	HSM	⟨DRILL⟩ ::= "*drill*" ⟨Attr⟩$_{from}$ "*to*" ⟨Attr⟩$_{to}$	*"drill down from product to type"*

- Proposing only the most promising choice makes it easier to create a baseline query, even though the optimal derivation could be missed. The baseline query can still be improved by adding or removing clauses through further interactions enabled by the `OLAP operator` step.

Definition 2 (Parse Forest Score). *Given a target sequence M and the corresponding parse forest PF_M, we define its score as $Score(PF_M) = Score(M')$ where M' is the sub-sequence of M belonging to the parse tree PT_M that includes non-annotated and ambiguous entities only.*

The parse forest holding the highest score is the one proposed to the user. This ranking criterion is based on an *optimistic-pessimistic* forecast of the outcome of the `Disambiguation & Enhancement` step. On one hand, we optimistically assume that the ambiguities belonging to PT_M will be positively solved in the `Disambiguation & Enhancement` step and the corresponding clauses and entities will be kept. On the other hand, we pessimistically assume that non-parsed clauses belonging to PF_M will be dropped. Errors (i.e., OLAP operators that cannot be applied to a full query) do not contribute to the score.

The rationale of our choice is that an ambiguous clause in the parse tree is more likely to be a proper interpretation of the text. As shown in Figure 10, a totally pessimistic criterion (i.e., excluding from the score all the annotated entities) would carry forward a too simple, but non-ambiguous, forest; conversely, a totally optimistic criterion (i.e., considering the score of all the entities in PF_M) would make preferable a large but largely non-parsed forest. Please note that the bare score of the target sequence (i.e., the one available before parsing) corresponds to a totally optimistic choice since it sums up the scores of all the entities in the target sequence.

The ranking criterion defined above enables the pruning of the target sequences to be parsed as shown by Algorithm 1. Reminding that target sequences are parsed in descending score order, let us assume that, at some step, the best parse forest is $PF_{M'}$ with score $Score(PF_{M'})$. If the next target sequence to be parsed, M'', has score $Score(M'') < Score(PF_{M'})$, we can stop the algorithm and return $PF_{M'}$ since the score of M'' is an upper bound to the (optimistic) score of the corresponding parse forest.

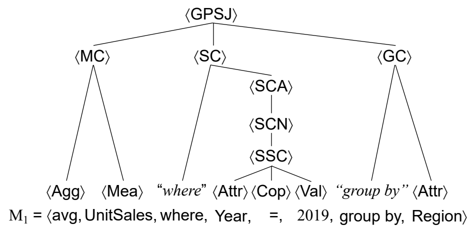


Figure 9: Parse forest enhancement: the implicit attribute Year has been added to M_1 from Figure 6a.

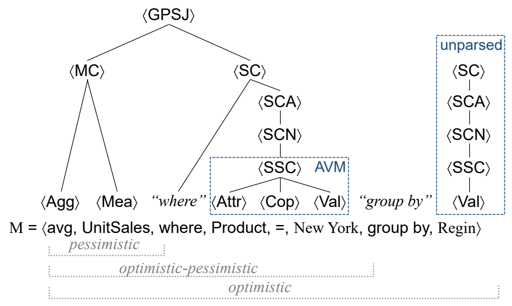


Figure 10: Portion of the parse forest contributing to its score depending on the adopted scoring criterion.

Algorithm 1 works as follows. At first, target sequences are sorted by their score (Line 1), the best parse forest is initialized, and the iteration begins. While the set of existing target sequences is not exhausted (Line 3), the best target sequence is picked, removed from the set of candidates, and its parse forest is generated (Lines 4–6). The current forest replaces the best one if it proves to be better than the latter (Line 7), i.e., if $score(PF_M) > score(PF^*)$ or, in case of a tie, if the number of annotations in PF_M is lower than the one in PF^*. In this case, the current forest is stored (Line 8) and all the target sequences with a lower score are removed from the search space (Line 9) as the pruned target sequences cannot produce parse forests with a score greater than what has been already parsed.

Note that, as $Score()$ requires a parse forest and the `Parsing` step always produces a parse forest (which might coincide with the parse tree), $Score()$, ranking and pruning work for both `Full query` and OLAP operator steps.

7. Parse Forest Disambiguation and Enhancement

If no ambiguities or errors were found in the previous steps, the parse forest coincides with a parse tree[7] that can be directly translated into an executable

[7]As unparsed clauses are annotated as ambiguities, if no ambiguities are found then all clauses are included in the parse tree.

Algorithm 1 Selection of the parse forest

```
Require: $\mathcal{M}$: set of valid target sequences
Ensure: $PF^*$: best parse forest
1: $\mathcal{M} \leftarrow sort(\mathcal{M})$                          ▷ sort target sequences by score
2: $PF^* \leftarrow \varnothing$
3: while $\mathcal{M} \neq \varnothing$ do                   ▷ while search space is not exhausted
4:     $M \leftarrow head(\mathcal{M})$             ▷ get the target sequence with highest score
5:     $\mathcal{M} \leftarrow \mathcal{M} \setminus \{M\}$                                  ▷ remove it from $\mathcal{M}$
6:     $PF_M \leftarrow parse(M)$                          ▷ parse the target sequence
7:     if $PF_M$ is better than $PF^*$ then
8:         $PF^* \leftarrow PF_M$                        ▷ store the new parse forest
9:         $\mathcal{M} \leftarrow \mathcal{M} \setminus \{M' \in \mathcal{M}, score(M') \leq score(PF^*)\}$
                            ▷ remove target sequences with lower scores from $\mathcal{M}$
   return $PF^*$
```

SQL query. Conversely, if COOL detects an error, the only possible solution is to return an informative warning to the user in order to help her resubmit a correct command. If none of the above apply, an annotated parse forest is returned. An annotation does not necessarily imply an interaction with the user to be solved. Indeed, COOL tries to minimize the number of such interactions by exploiting the `MR`, previous activities of the user stored in the `Log`, and (when applicable) the parse tree of the previous query PT_C. At the end of the `Disambiguation & Enhancement` step, the ambiguous parse forest is reduced to a non-ambiguous parse tree as all the ambiguities are solved in this step (existing unparsed clauses are either added to ⟨GPSJ⟩ or dropped).

We identify three ways to solve ambiguities: implicit, default-based, and user-based.

Implicit. Refers to the cases where the parse forest does not include all the information necessary to produce the SQL code, but the missing parts can be automatically inferred either from PT_C or the `MR` since only one solution is possible. In this case, the parse forest is automatically completed and no interaction with the user is required. Examples of automatic inference are the following.

- **Implicit aggregation operator in ⟨MC⟩**: the aggregation operator in a ⟨MC⟩ clause is implicit and the measure is associated with a single aggregation operator.
- **Implicit attribute in ⟨SSC⟩**: the ⟨SSC⟩ clause has an implicit attribute and the parsed member belongs to a single attribute domain.
- **Implicit comparison operator in ⟨SSC⟩**: the ⟨SSC⟩ clause has both attribute and member(s). We assume " = " as ⟨COP⟩ if ⟨SSC⟩ has a single member, otherwise we assume "*in*" as ⟨COP⟩ (Figure 9).

Table 4: User interaction templates and actions for *user-based* ambiguity solution. The line splits annotations related to full-query interpretation from those related to OLAP-operator interpretation.

Name	**Template**	**Action**
AA	$\langle$Val$\rangle$ is member of these attributes [...]	Pick attribute/drop
AAO	$\langle$Mea$\rangle$ allows these operators [...]	Pick operator/drop
AVM	$\langle$Attr$\rangle$ and $\langle$Val$\rangle$ domains mismatch, possible values are [...]	Pick value/drop
MV	$\langle$Mea$\rangle$ does not allow $\langle$Agg$\rangle$, possible operators are [...]	Pick operator/drop
AV	Impossible to group by on $\langle$Attr$\rangle$ without $\langle$Attr$\rangle$	Add attribute/drop
UC $\langle$GC$\rangle$	There is a dangling grouping clause $\langle$GC$\rangle$	Add to $\langle$GPSJ$\rangle$/drop
UC $\langle$MC$\rangle$	There is a dangling measure clause $\langle$MC$\rangle$	Add to $\langle$GPSJ$\rangle$/drop
UC $\langle$SC$\rangle$	There is a dangling predicate clause $\langle$SC$\rangle$	Add to $\langle$GPSJ$\rangle$/drop
TO	$\langle$Attr$\rangle_{from}$ can be generalized/specialized to [...]	Pick attribute/drop
FR	$\langle$Attr$\rangle_{to}$ can be specialized/generalized from [...]	Pick attribute/drop
NEC	The specified clause does not exist in $\langle$GPSJ$\rangle$	Alert & drop
AEC	The specified clause already exists in $\langle$GPSJ$\rangle$	Alert & drop
ID	Cannot drill down from the finest attribute $\langle$Attr$\rangle$	Alert & drop
FA	Cannot roll up/drill down on descriptive attribute $\langle$Attr$\rangle$	Alert & drop
HM	$\langle$Attr$\rangle_{from}$ and $\langle$Attr$\rangle_{to}$ belong to different hierarchies	Alert & drop
HSM	$\langle$Attr$\rangle_{from}$ is coarser/finer than $\langle$Attr$\rangle_{to}$	Alert & drop

- **Implicit attribute in $\langle$DRILL$\rangle$ (or $\langle$ROLL$\rangle$)**: the $\langle$DRILL$\rangle$ (or $\langle$ROLL$\rangle$) clause has an implicit attribute, and, based on the previous query context PT_C, it is necessary to infer its role (*from* or *to*) to complete the OLAP operator. Given the attribute $\langle$Attr$\rangle$ in the $\langle$DRILL$\rangle$ (or $\langle$ROLL$\rangle$) operator, if $Attr \in \langle$GC$\rangle$, we assume *from* as the role of $\langle$Attr$\rangle$, and $\langle$Attr$\rangle_{to}$ is inferred as the finer (or coarser) attribute of $\langle$Attr$\rangle$. Otherwise, if $Attr \notin \langle$GC$\rangle$, we assume *to* as the role of $\langle$Attr$\rangle$, and $\langle$Attr$\rangle_{from}$ is inferred as the coarser (or finer) attribute of $\langle$Attr$\rangle$.

Default-based. Refers to the cases where more than one solution can be applied to complete the parse tree but either a default solution has been defined in the `MR` or it can be inferred from the `Log` according to the previous user disambiguations. In this case, the preferred solution is applied and the user interaction is limited to evidencing it as *hint*. The user can *optionally* manually refine this solution. More details on this case are provided later in this section.

User-based. Refers to the remaining cases where no automatic solutions apply. In this case, a user interaction is required to disambiguate. Table 4 reports the user interaction templates and actions for *user-based* ambiguity resolutions. Each template allows to either provide the missing information or to drop the annotated clause. Templates are standardized and user choices are limited to keep the interaction easy. This allows also unskilled users to obtain a baseline query.

As for default-based ambiguities, if the `Log` highlights a recurring choice in the way of solving ambiguities, COOL infers a preference and this knowledge can be leveraged to reduce the number of user interactions, turning a user-based solution in a smoother default-based one. To this end, similarly to [12], at each disambiguation step d we store in the `Log` L a triple $d = (A, T, T^*)$, where $d.A$ is the annotation name, $d.T$ is the annotated subtree and $d.T^*$ is the disambiguated subtree (eventually empty if the user dropped T). Given L, the

Algorithm 2 Log-based Disambiguation

```
Require: $PF_M$: annotated parse forest, $\tau$: frequency threshold
Ensure: $PF^*_M$: partially solved annotated parse forest
1: $PF^*_M \leftarrow PF_M$                                    ▷ Initialize the parse forest
2: for each $d \in annotations(PF_M)$ do        ▷ Iterate over its annotated subtrees
3:     $T^* = argmax_{T'} f((d.A, d.T, T'))$      ▷ Get most frequent disambiguation
4:     if $f((d.A, d.T, T^*)) \geq \tau$ then      ▷ If frequency is higher than threshold
5:         Replace $d.T$ with $T^*$ in $PF^*_M$                            ▷ ... replace it
6:         Annotate $T^*$ as hint                    ▷ ... annotate the subtree as hint
7: return $PF^*_M$
```

log-based disambiguation frequency is defined as the ratio between the number of times in which a certain replacement is chosen to resolve an ambiguity and the number of times in which the ambiguity occurred.

$$f(d) = \frac{|\{d' \in L \ s.t. \ d'.A = d.A, d'.T = d.T, d'.T^* = d.T^*\}|}{1 + |\{d' \in L \ s.t. \ d'.A = d.A, d'.T = d.T\}|}$$

Given the frequency threshold τ, Algorithm 2 shows the automatic disambiguation process. Given an annotated parse forest (Line 1), the algorithm iterates over the annotated subtrees (Line 2). For each annotation, the algorithm picks the disambiguation with the highest frequency (Line 3). If the frequency is higher than a given threshold (Line 4), then the algorithm replaces the annotated subtree with the frequent one within the parse forest (Line 5) and marks it as a hint (Line 6).

Example 6 (Log-based disambiguation). *Given the target sequence* $M = \langle \mathsf{sum}, \mathsf{UnitSales}, \mathsf{where}, New\ York\rangle$*,* `Parsing` *outputs a parse forest where the subtree* $T' = SSC(New\ York)$[8] *corresponding to the SSC clause* $New\ York$ *(where* $New\ York$ *is a member of both* StoreCity *and* CustomerCity*) is annotated with an ambiguous attribute AA annotation. Given a threshold* $\tau = 0.5$ *and the* Log

$$\begin{aligned} L = \{&(AA,\ SSC(New\ York),\ SSC(\mathsf{StoreCity}, =, New\ York)),\\ &(AA,\ SSC(New\ York),\ SSC(\mathsf{StoreCity}, =, New\ York)),\\ &(AA,\ SSC(New\ York),\ SSC(\mathsf{CustomerCity}, =, New\ York)),\ ...\}\end{aligned}$$

the subtree T' *is automatically replaced with* $T^* = SSC(\mathsf{StoreCity}, =, New\ York)$ *as* $f((AA,\ SSC(New\ York),\ SSC(\mathsf{StoreCity}, =, New\ York))) = \frac{2}{4} = 0.5 \geq \tau$. □

[8]Elements between brackets represent children of the parent node. The node SCC is the parent of the node $New\ York$.

8. SQL Generation

SQL generation translates a full-query parse tree into an executable SQL query[9]. If an OLAP operator has been submitted, the context parse tree PT_C must be updated according to the OLAP operator parse tree PT. All the OLAP operators can be implemented atop the addition/removal of new/existing nodes in PT_C. As in Section 6.2, we apply a depth-first search algorithm to retrieve the clauses interested by the OLAP operator. We recall that adding a new clause to $\langle \mathsf{GPSJ} \rangle$ (e.g., "add city" requires to add the attribute City) requires to append the new clause to the existing $\langle \mathsf{MC} \rangle$/$\langle \mathsf{GC} \rangle$/$\langle \mathsf{SC} \rangle$ (and to create it if it does not exist in $\langle \mathsf{GPSJ} \rangle$). Note that an $\langle \mathsf{SSC} \rangle$ is appended with the Boolean and operator as in Figure 8f. Conversely, when dropping a clause produces an invalid parent clause (e.g., an empty $\langle \mathsf{GC} \rangle$ or an unbalanced $\langle \mathsf{SSA} \rangle$) in the parse tree, it is sufficient to remove the parent clause from $\langle \mathsf{GPSJ} \rangle$.

Given a full query parse tree PT, the generation of its corresponding SQL requires to fill in the `SELECT`, `WHERE`, `GROUP BY` and `FROM` statements. The SQL generation applies to both star and snowflake schemas [15] and is done as follows:

- `SELECT`: measures and aggregation operators from $\langle \mathsf{MC} \rangle$ are added to the query selection clause together with the attributes in the group by clause $\langle \mathsf{GC} \rangle$;
- `WHERE`: predicates from the selection clause $\langle \mathsf{SC} \rangle$ (i.e., values and their respective attributes) are added to the query predicate;
- `GROUP BY`: attributes from the group by clause $\langle \mathsf{GC} \rangle$ are added to the query group by set;
- `FROM`: measures and attributes/values identify, respectively, the fact and the dimension tables involved in the query. Given these tables, the join path is identified by following the referential integrity constraints (i.e., by following foreign keys from dimension tables imported in the fact table).

Example 7 (SQL generation). *Given the GPSJ query "sum the unit sales by type in the month of July", its corresponding SQL is:*

```
SELECT Type, sum(UnitSales)
FROM Sales s JOIN Product p ON (s.pid = p.id)
             JOIN Date d ON (s.did = d.id)
WHERE Month = "July"
GROUP BY Type
```

□

[9]Notice that a full-query parse tree is also translatable into MultiDimensional eXpressions (MDX), i.e., the standard OLAP query language [35]; in this paper, we translate it into SQL since the latter is more widely supported and our experiments rely on a relational OLAP (ROLAP) implementation.

9. Experimental Tests

In this section, we evaluate COOL in terms of effectiveness, efficiency, and user experience. On one hand, effectiveness and efficiency evaluate COOL's capabilities to correctly interpret a text in a time compatible with real-time interaction. On the other hand, user experience evaluates how well COOL helps the user in formulating OLAP sessions. Indeed, there is no point in creating a natural language interface that is not easy to use (also by inexperienced users).

To evaluate COOL against heterogeneous cubes, tests are carried out on Foodmart[10] and SSB [27] schemas, with $8.7 \cdot 10^4$ and $6 \cdot 10^6$ facts, respectively. As to Foodmart, the `Automatic MR feeding` module populated the `MR` with 1 cube, 39 attributes, $1.3 \cdot 10^4$ entities. As to SSB, the `Automatic MR feeding` module populated the `MR` with 1 cube, 50 attributes, and $3.3 \cdot 10^5$ entities. 50 additional synonyms are added in the `Manual MR enrichment` step for both datasets (e.g., *"for each"*, *"for every"*, *"per"* are synonyms of the `group by` statement).

To the best of our knowledge, no standard benchmark exists for natural language GPSJ queries. Nonetheless, [8] describes a real-word benchmark for generic analytics queries. 110 queries out of 147 (i.e., 75% of the benchmark) turned out to meet the GPSJ expressiveness, confirming how general and standard GPSJ queries are. Since queries in [8] refer to private datasets, we mapped them to both the Foodmart and SSB schemas; we preserved the structure of the original queries (e.g., word order, typos, etc.). For each query, we manually defined the ground truth, i.e. the parse tree resulting from the correct text interpretation. More than one correct parse trees might exist; for instance, once parsed, *"sum sales by category and month"* and *"sum sales by month and category"* produce two different parse trees representing the same full query.

9.1. Effectiveness

Effectiveness is evaluated as the parse tree similarity $TSim(PT, PT^*)$ between the parse tree PT produced by COOL and the manually-defined ground truth PT^*. Parse tree similarity is based on the tree distance [38]: it ranges in $[0, 1]$ and it keeps into account both the number of correctly parsed entities (i.e., the parse tree leaves) and the tree (i.e., query) structure (e.g., the selection clauses *"(A and B) or C"* and *"A and (B or C)"* refers to the same parsed entities but underlie different tree structures).

Table 5 summarizes the parameters considered in our approach: token sub-sequences have maximum length n, each sub-sequence is associated with the top N similar entities with similarity higher than α, and only the target sequences covering at least a percentage β of the tokens in T is covered. We set n to the maximum number of words representing an entity in the `MR`; e.g., for Foordmart,

[10]A public dataset about food sales between 1997 and 1998 (`https://github.com/julianhyde/foodmart-data-mysql`).

Table 5: Parameter values for testing.

Symbol	**Meaning**
N	Num. of top similar entities
α	Token/entity minimum similarity
β	Sentence coverage threshold
n	Maximum sub-sequence length

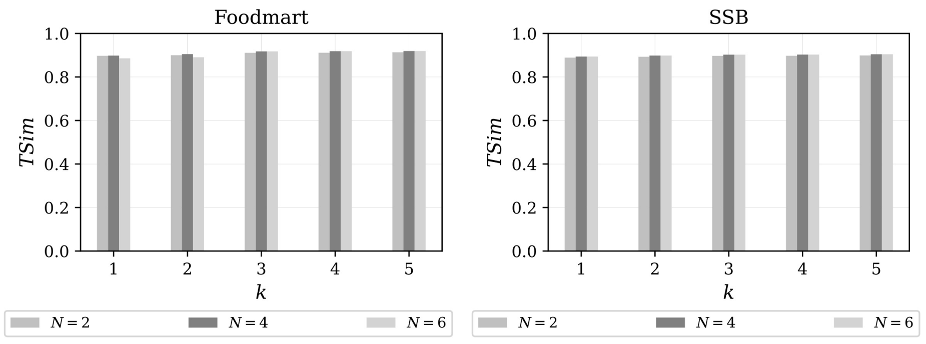
$n = 4$ as no entity in the `MR` is longer than 4 words. The value of β is fixed to 70% based on an empirical evaluation of the benchmark queries.

Since COOL is the first proposal addressing OLAP sessions, there is no direct competitor to base the comparison. Although the approaches proposed in [20, 36, 29] are potentially applicable to our domain, it was impossible to run a comparison against them since (i) the implementations are private and the provided descriptions are far from making them reproducible, (ii) despite the availability of natural language datasets (e.g., the ones used in [20]), these datasets are hardly compatible with multidimensional constraints and GPSJ queries, and (iii) some of these approaches require ad-hoc knowledge (e.g., domain-specific ontologies) that are not publicly available.

9.1.1. Effectiveness Without Disambiguation

Figures 11 and 12 depict the performance of our approach varying the number of retrieved top similar entities (i.e., $N \in \{2, 4, 6\}$) or the similarity threshold (i.e., $\alpha \in \{0.4, 0.5, 0.6, 0.7\}$). Values are reported to the best of the top-k trees (i.e., the k trees with the highest score). We remind that only one parse forest is involved in the `Disambiguation & Enhancement` step; nonetheless, for testing purposes, it is interesting to see if the best parse tree belongs to the top-k ranked ones. For the Foodmart dataset, effectiveness slightly changes by varying N and α, and it ranges respectively in $[0.88, 0.92]$ and $[0.86, 0.92]$. Similarly, for the SSB dataset, effectiveness ranges in $[0.88, 0.90]$ (when varying N) and in $[0.88, 0.92]$ (when varying α). In both cases, the best results are obtained when more similar entities are admitted (i.e., lower α, higher N), and more candidate target sequences are generated. Independently of the chosen thresholds, the results of COOL are very stable (i.e., the effectiveness variations are limited), even by considering only the top-1 query. Noticeably, Foodmart and SSB schemas provided very similar results.

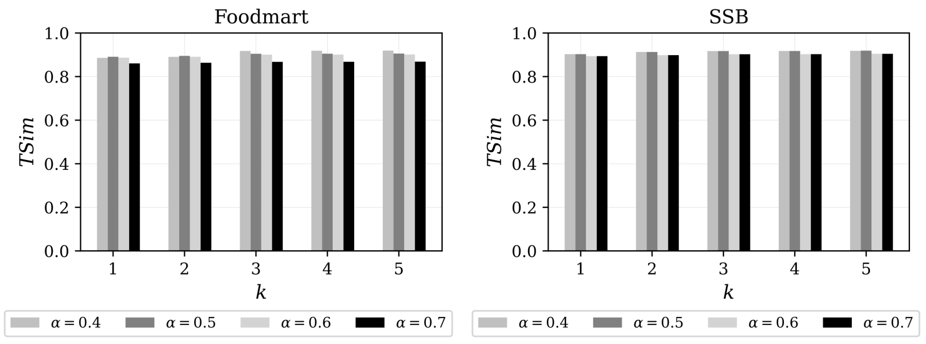
Even in the presence of typos, the robustness of our similarity function allows COOL to match the correct entities. However, since entities in the metadata repository can cause conflicts while selecting the correct ones for the interpretation, Figure 13 depicts how effectiveness changes varying the entities in the `MR`. For both datasets, we manually ensured that the `MR` contains the entities necessary to correctly interpret the dataset—almost 100 considering members, attributes, operators, etc.—and randomly add the remaining entities until all are considered (the average results are shown). Noticeably, increasing the `MR` size has no impact on the average effectiveness; the correct entities are not al-



(a) Foodmart (with $\alpha = 0.4$).

(b) SSB (with $\alpha = 0.7$).

Figure 11: Effectiveness varying the number of top-N similar entities and top-k queries returned.



(a) Foodmart (with $N = 6$).

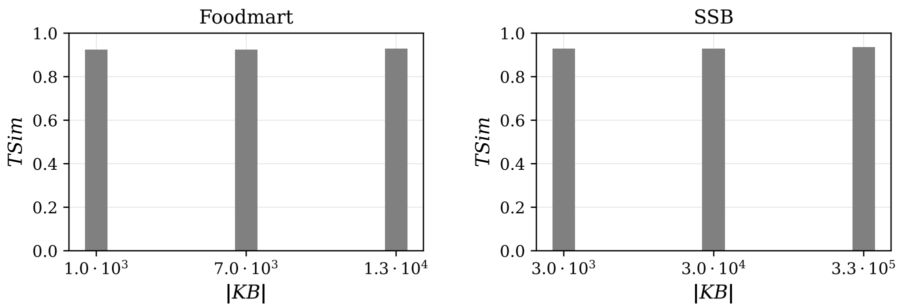
(b) SSB (with $N = 6$).

Figure 12: Effectiveness varying the similarity threshold α and the number of top-k queries returned.

tered by the presence of other entities. Although it is true that a textual token likely matches more entities, this really depends on the dataset and benchmark at hand. When the quality of the spoken/written English is high (e.g., few typos or text matches DW entities with high similarity) it is intuitive to verify that COOL matches the *right* entities (we remind that queries from [8] are a real-world benchmark).

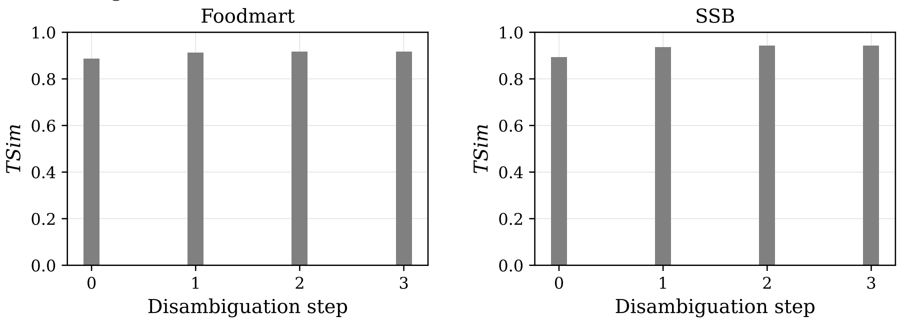
These results confirm the following points.

- The choice of proposing only one query to the user does not negatively impact on performance (while it positively impacts on interaction complexity and efficiency).
- Our scoring function properly ranks parse tree similarity to the correct interpretation for the query since the best ranked is in most cases the most similar to the correct solution.



(a) Foodmart (with $k = 1$, $N = 6$, and $\alpha = 0.4$). (b) SSB (with $k = 1$, $N = 6$, and $\alpha = 0.7$).

Figure 13: Effectiveness as a function of the number of entities in the `MR`.



(a) Foodmart (with $k = 1$, $N = 6$, and $\alpha = 0.4$). (b) SSB (with $k = 1$, $N = 6$, and $\alpha = 0.7$).

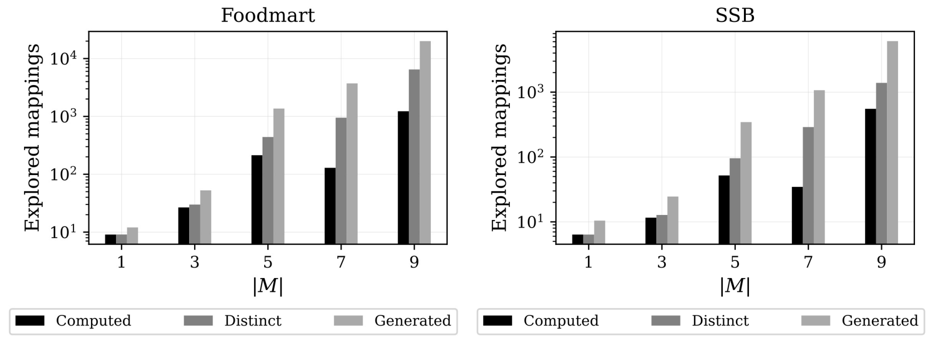
Figure 14: Effectiveness as a function of the number of required disambiguation steps.

- COOL preserves stable effectiveness even against different schemas.
- Our similarity function is robust enough to rule out entities that are similar to the correct ones (even in case of natural language with typos).

9.1.2. Effectiveness With Disambiguation

Only 58 queries out of 110 are not ambiguous and produce parse trees that can be fed *as-is* to `SQL generation` and `Execution & Visualization`. This means that 52 queries—despite being very similar to the correct tree, as shown by the aforementioned results—are not directly executable without disambiguation (we recall the ambiguities from Table 2). Indeed, of these 52 queries, 38 contain one ambiguity annotation, 12 contain two ambiguity annotations, and 2 contain three or more ambiguity annotations.

Figure 14 depicts the performance when the best parse tree undergoes iterative disambiguation (i.e., consequent correcting actions are applied). To do so, we consider the best configuration with near-real-time efficiency (see Section 9.2). As to Foodmart, given $N = 6$ and $\alpha = 0.4$, the effectiveness increases from 0.89 up to 0.93. As to SSB, given $N = 6$ and $\alpha = 0.7$, the effectiveness increases from 0.89 up to 0.94. Overall, unsolved differences between user parse trees PT and the ground truth PT^* are mainly due to missed entities in



(a) Foodmart (with $N = 6$ and $\alpha = 0.4$). (b) SSB (with $N = 6$ and $\alpha = 0.7$).

Figure 15: Number of generated, distinct and parsed target sequences varying the number $|M|$ of entities in the optimal tree. Given *all* the target sequences generated in the `Tokenization & Mapping` step, some target sequences are identical and only the *distinct* ones are kept. The ranking and scoring functions further prune the actually *computed* target sequence.

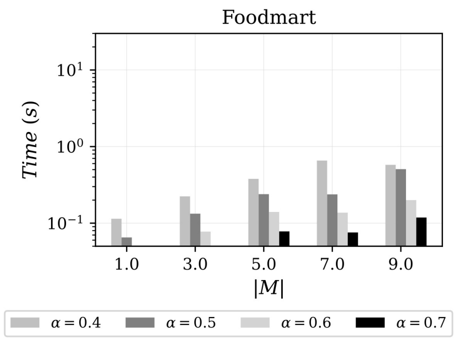
the target sequences (e.g., a word contains too many typos to be resolved or a synonym is missing in the `MR`).

9.2. Efficiency

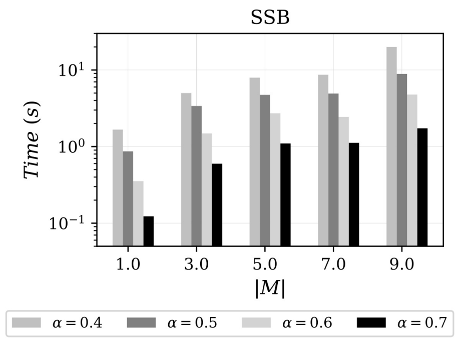
In Section 4, we have shown that the search space of target sequences increases exponentially in the text length. Figure 15 confirms this result on both datasets, showing the number of target sequences as a function of the number of entities $|M|$ included in the optimal parse tree PT^*. Note that $|M|$ is strictly related to the number of tokens in the text $|T|$. Since a target sequence can be generated by multiple mappings, we only consider distinct target sequences. Then, we only parse the most promising distinct target sequences by exploiting our scoring function. Noticeably, pruning rules strongly limit the number of target sequences to be actually parsed.

We ran the tests on a machine equipped with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz CPU and 8GB RAM, with COOL implemented in Java. Tests are run in the worst-case scenario, i.e. when all the entities in the `MR` are present. Queries are executed against enterprise data marts hosted on Oracle 11g and their performance depends on the underlying multidimensional engine and on the complexity of the query itself. We emphasize that the execution time corresponds to the time necessary for the interpretation, and not to the time to execute the queries. Since the execution time increases with the number of entities $|M|$ included in the optimal parse tree, we assess how efficiency changes by increasing $|M|, N$, and α.

Since the similarity function (introduced in Section 4) between a string T and an entity W is based on the Levenshtein distance (with complexity $O(|T| \cdot |W|)$), we want to reduce the number of similarity comparisons necessary to get the entities similar to T. In the case of linear search, such number is $O(|\texttt{MR}|)$. To reduce this complexity, we leverage the BK-tree data structure [35] to perform

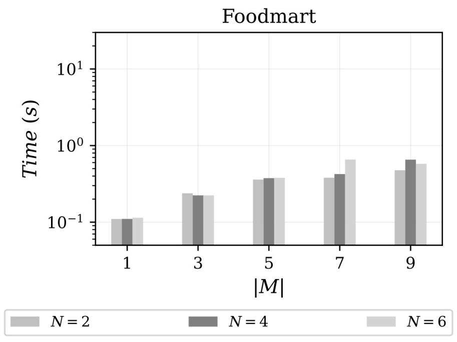


(a) Foodmart (with $k = 1$ and $N = 6$).

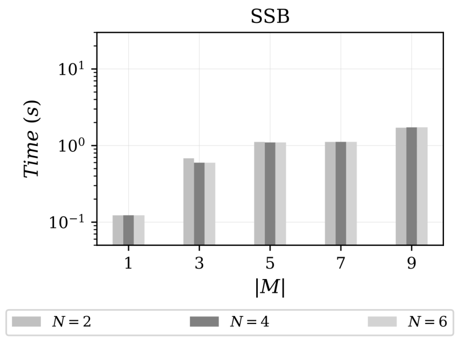


(b) SSB (with $k = 1$ and $N = 6$).

Figure 16: Efficiency as a function of the minimum similarity threshold α.



(a) Foodmart (with $k = 1$ and $\alpha = 0.4$).



(b) SSB (with $k = 1$ and $\alpha = 0.7$).

Figure 17: Efficiency as a function of the top N similar entities.

approximate string matching with a complexity of $O(log(|\texttt{MR}|))$ (note that this tree is computed during the offline phase). As a result, α affects the overall efficiency: the lower is α, the more entities are taken into account. Figure 16 shows the average execution time by varying $|M|$ and α. Since the execution increases for lower similarity values (α), for bigger datasets (such as SSB) it is necessary to increase the minimum α (e.g., to 0.7) to keep the overall efficiency in the order of (a few) seconds. As shown in Section 9.1, this has a slight impact on effectiveness.

Noticeably, Figure 17 shows that the effect of selecting more entities (i.e., higher N) on the average execution time is almost negligible.

While we run our tests against the entire MR, it is worth reasoning on what happens by increasing the number of attributes/members. Since the performance of COOL is based on two aspects (i.e., mapping and parsing), increasing the number of attributes/members could affect the number of mappings due to the wider search space for entity similarity (Equation (1)). However, as we pick a fixed number of synonyms, this does not affect parsing since the number of

Table 6: Results of user evaluation in terms of average accuracy $TSim()$, average elapsed time T (in seconds), and average number of user interactions I; the Skill column distinguishes users based on their familiarity with OLAP.

		`Full query`			`OLAP operator`		
Id	Skills	$TSim(PT^u_s, PT^*_s)$	$T_s(s)$	I_s	$TSim(PT^u_e, PT^*_e)$	$T_{op}(s)$	I_{op}
s_a	Low	0.98	141	0.57	0.96	144	0.29
	High	0.94	145	0.33	0.93	128	0.39
s_b	Low	0.92	92	1.43	0.70	102	0.29
	High	0.93	95	0.56	0.79	114	0.18
s_c	Low	0.91	98	1.58	0.90	51	0.00
	High	0.94	99	0.78	0.93	56	0.12
s_d	Low	0.84	200	2.60	0.84	-	-
	High	0.79	129	2.00	0.87	45	0.14
s_e	Low	1.00	49	0.20	1.00	-	-
	High	1.00	40	0.14	1.00	-	-
s_f	Low	0.79	271	3.50	0.79	-	-
	High	0.82	121	1.57	0.89	39	0.14
s_g	Low	0.91	136	3.25	0.86	114	0.25
	High	0.93	50	0.50	1.00	46	0.08
Overall	Low	0.91	141	1.88	0.86	102	0.21
	High	0.91	97	0.84	0.92	71	0.18

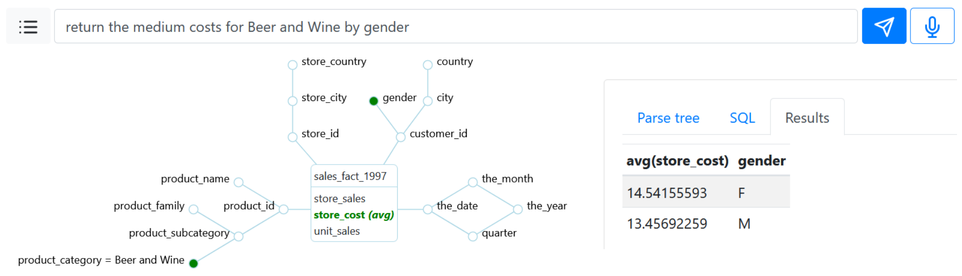
generated target sequences does not change.

9.3. User Experience Evaluation

The main goal of a natural language interface is to enable a user to easily formulate a command. To this end, we tested COOL with 55 users, mainly students in data science and in engineering management, with knowledge of business intelligence and data warehousing varying from none to advanced. On a scale from 1 (very poor) to 5 (very high), on average, users scored 3.60 ± 0.7 their familiarity with the English language. On a scale from 1 (none) to 5 (very high), 16 users showed no familiarity with the OLAP paradigm, while 39 users showed medium to very-high familiarity.

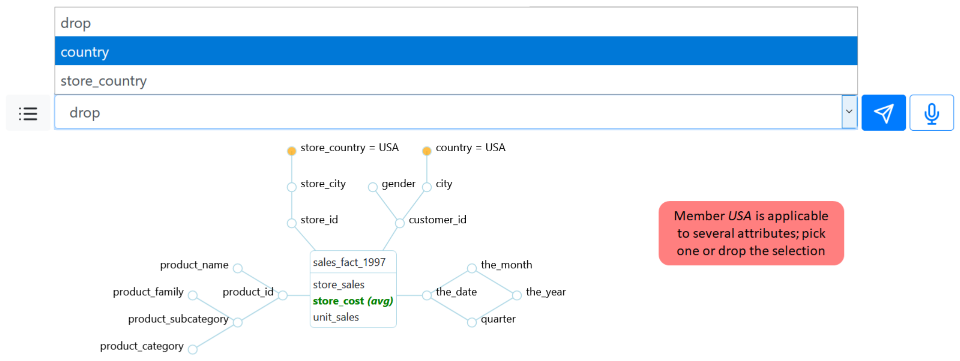
For the sake of testing, we implemented a web application (Figure 18) in which users submit written or spoken descriptions of full queries/OLAP operators. For this test, we only considered the Foodmart schema. User interaction is also supported by the DFM visual representation of the Foodmart schema. From our experience, the DFM easily allows non-expert users to visualize the entire cube expressivity without the need to know the underlying physical schema, such as which measures and attributes can be queried, as well as to intuitively infer how attributes aggregate (without the need to know the meaning of functional dependencies). Additionally, this visual metaphor combines well with the visualization of ambiguously/correctly interpreted entities (as later discussed).

Two types of tests (*formal-based* and *goal oriented*) have been conducted with an increasing complexity. Formal-based tests assess the capability of users to generate natural language OLAP sessions given the formal content of each query. By doing so, no natural language bias is introduced by the tests. Goal-oriented tests assess the capability of users to formulate natural language OLAP


return the medium costs for Beer and Wine by gender

avg(store_cost)	gender
14.54155593	F
13.45692259	M

(a) The non-ambiguous query *"return the medium costs for Beer and Wine by gender"* is issued. Elements highlighted in green have been understood by COOL.



Member *USA* is applicable to several attributes; pick one or drop the selection

(b) The ambiguous query *"return the medium costs in USA"* is issued. COOL asks to the user if *USA* is member of Country or StoreCountry or if *USA* should be dropped (ambiguity AA in Table 4). Elements highlighted in yellow require user-based disambiguation.

Figure 18: User interface of COOL implemented for the testing purpose.

sessions *in English* after being provided with the description of analytic goals *in Italian* (users were asked not to use translating tools). By doing so, no formal bias is introduced by the tests, as users are required to understand and submit the OLAP sessions necessary to achieve each of the goals. A 10-minute tutorial was presented to show the users how COOL works and how the tests are organized. We emphasize that—besides the content of a GPSJ query—users are not required to be knowledgeable of any other formal concept or programming language (e.g., SQL or MDX). To support this (democratization) claim, we split users into two groups depending on their familiarity (low or high) with OLAP. All KPIs are averaged on users within the same group (Table 6). Following the previous effectiveness evaluation, we set $\alpha = 0.4$, $\beta = 70\%$, $N = 6$ and $n = 4$.

We first introduce the two types of tests and we finally draw the overall evaluation.

9.3.1. Formal-based

The following testing protocol has been adopted.

- Users undergo 3 OLAP sessions $\{s_a, s_b, s_c\}$ with increasing complexity.

Each OLAP session is composed of three steps (q, op', op''), where q is a full query and op' and op'' are the OLAP operators that are iteratively applied to the full query. In each OLAP session, we provided three formal queries (q, q', q''), with q' (or q'') slightly changing from q (or q'). At first, users were asked to produce the natural language description of the full query q. Then, by difference with the following query, users were asked to understand which OLAP operator op changed q (or q') to q' (or q''), and to issue the natural language description of op to COOL.

- At each step, users are asked to issue the natural language description in English to COOL. If fully parsed, COOL visualizes the query result as a pivot table (Figure 18a). Otherwise, the disambiguation process starts, and users are asked to disambiguate what COOL was not capable of inferring automatically (Figure 18b).
- When all ambiguities (if any) are solved, the next step in the session is proposed.
- Users explicitly end the OLAP session after running q''.

Example 8 (OLAP session). *Given the formal full query*

$$q = \{\{\textsf{avg StoreCost}\}, \{\textsf{gender}\}, \textsf{category} = \text{“}\textit{Beer and Wine}\text{”}\}$$

a user is asked to produce a natural language description of the query, such as “return the medium costs for Beer and Wine by gender”. As COOL is capable of fully interpreting this query (Figure 18a), the next formal query

$$q' = \{\{\textsf{avg StoreCost}\}, \{\textsf{gender}, \textsf{month}\}, \textsf{category} = \text{“}\textit{Beer and Wine}\text{”}\}$$

is presented to the user. The user is asked to understand which OLAP operator can be applied to q to produce q' and to issue its natural language description to COOL. Note that the applicable OLAP operator is not unique. For instance, depending on the familiarity with the OLAP paradigm, the user may choose either to “drill down to month” or to “add the month”. □

We call PT^u_s the parse tree of the first full query issued by the user (i.e., the user description of q), and PT^u_e the parse tree of the last full query modification (i.e., when the user ends the session). We call PT^*_s and PT^*_e our ground truth, i.e. the parse trees corresponding to q and q''. Note that there is not a single correct/predefined path driving the OLAP session. For instance the OLAP session can diverge/converge from/to the ground truth (e.g., a full query that does not comprehend all the necessary entities can be completed with consequent OLAP operators); users can concatenate ⟨ADD⟩ and ⟨REMOVE⟩ instead of the ⟨REPLACE⟩ operator; or some disambiguation steps might be necessary to complete the session due to ambiguities or inaccuracies.

For each OLAP session, we evaluate both the starting full query and the consequent OLAP operators. We adopt the following key performance indicators

(KPIs): accuracy $TSim()$, interpretation time T, extra user interactions I (in the optimal case $I = 0$; i.e., the interpretation discloses no ambiguities that require user interactions). In detail:

- `Full query`. $TSim(PT^u_s, PT^*_s)$ measures how good is the interpretation of the full query; T_s adds up the time it takes for the user to formulate the query and COOL to interpret it; and I_s counts the number of user interactions to produce a fully-parsed full query.

- `OLAP operator`. $TSim(PT^u_e, PT^*_e)$ measures how close the user query is to the correct query after applying the OLAP operators. For each OLAP operator op, T_{op} sums the time it took for the user to formulate the operator, and for COOL to interpret them and to apply it to the user full query; and I_{op} counts the number of user interactions to produce a fully-parsed OLAP operator.

9.3.2. Goal oriented

This type of test assesses the capability of users to understand four analytic goals (in Italian) and to formulate their own (English) natural language OLAP sessions (s_d, s_e, s_f, and s_g); users are asked not to use translating tools. While the formal-based tests directly provided the analytic steps that users have to follow, in these tests users are free to issue a full query and (optionally) opt for OLAP operators in case of need. In principle, all the analytic goals can be fulfilled via a single full query (i.e., $PT^*_s = PT^*_e$).

Example 9. *An example of an analytic goal to be interpreted into an English natural language query is "Scrivi una query (in inglese) che ti permetta di capire quale sia la famiglia di prodotti che ha venduto più unità nel 1997" (which literally translates to "Write a query (in English) that allows you to understand which is the product family who sold the most units in 1997"). Interestingly, some users provided interpretations aimed to solve the analytic goal in a step. For instance "Get the product family which has the highest unit sales in 1997" or "Show me the sum of unit sales in 1997 by product family". Other users preferred to issue a shorter full query and to enrich it using OLAP operators. For instance the full query "Show the unit sales" has been later refined using "add the product family" and "filter on 1997".* □

9.3.3. Overall Evaluation

The average accuracy for full query interpretation is comparable for both experienced and inexperienced users (for s_a, s_d users with no/low OLAP skills performed even better). Similar considerations on accuracy also hold for the average accuracy of the OLAP operator, where experienced OLAP users perform slightly better (due to the higher familiarity with the OLAP analysis). As to the formal-based tests (s_a, s_b, and s_c), the average accuracy for full query interpretation is 0.94 for both skill levels. The decrease in accuracy $TSim(PT^u_s, PT^*_s)$ from s_a to s_c for inexperienced users is justified by the increasing complexity of the initial full query. These considerations also hold for the average accuracy of the OLAP operator (besides some vocabulary issues on s_b solvable by enriching the metadata repository). As to the goal-oriented tests (s_d, s_e, s_f, and s_g), inexperienced users tend to prefer single and longer full queries, at the cost of higher complexity in expressing the query; indeed, they take almost an interaction more while writing a full query. Conversely, experienced users are also fine with providing smaller full queries and then refining the results by issuing OLAP operators. At the end of the OLAP session, the average similarity for inexperienced users is 0.87 while it is 0.94 for experienced users, showing that—as expected—experienced users have a better capability in understanding and solving analytic tasks. The accuracy drop in s_d, s_f is due to vocabulary issues solvable by enriching the metadata repository.

As to automatic disambiguation (i.e., hints), 79% of ambiguities have been correctly resolved by COOL (i.e., the proposed hints match the user choice). Log-based disambiguation is very effective since most users submit correct queries. This further reinforces log-based disambiguation; its effectiveness could decrease if users' choices were less focused.

The time necessary to interpret a full query is higher for inexperienced users (141 seconds vs 97 seconds). While the formal-based tests provided very similar times for both types of users, goal-oriented tests show that the time necessary to understand the analytic goal is higher for inexperienced users (supporting the claim mentioned before). Noticeably, for all users, the time required for the OLAP operator sensibly decreases along with increasing familiarity with COOL (the same happens for the full query, but the effect is compensated by the increasing complexity).

We conclude that:

- the small number of extra interactions—together with the high accuracy that consolidates the results in Section 9.1—proves COOL's robustness in natural language interpretation (we recall that misinterpreted clauses count as unparsed ambiguities). In turn, this enables users to issue complete queries and OLAP operators in either 1 or 2 interactions (i.e., 0 or 1 extra interactions);
- both experienced and inexperienced users achieve comparable accuracy results, supporting our claim on democratization: no formal background nor programming language is needed to perform OLAP sessions. Still, results show that experienced users are more efficient in understanding and interpreting analytic tasks.

10. Related Works

Conversational business intelligence can be classified as a *natural language interface* (NLI) to *business intelligence* systems to drive analytic sessions. Despite the plethora of contributions in each area, to the best of our knowledge, no approach lies at their intersection.

NLIs to operational databases enable users to specify complex queries without previous training on formal programming languages (such as SQL) and software; a recent and comprehensive survey is provided in [1]. Overall, NLIs are divided into two categories: question answering and dialog. While the former are designed to operate on single queries, only the latter are capable of supporting sequences of related queries as needed in OLAP analytic sessions. However, to the best of our knowledge, no dialog-based system for OLAP sessions has been provided so far. The only contribution in the dialog-based direction is [23], where the authors provide an architecture for querying relational databases; with respect to this contribution we rely on the formal foundations of the multidimensional model to drive analytic sessions (e.g., according to the multidimensional model it is impossible to group by a measure, compute aggregations of categorical attributes, aggregate by descriptive attributes, ensure drill-across validity). Also differently from [23], the results we provide are supported by extensive effectiveness and efficiency performance evaluation that completely lack in [23]. Finally, existing dialog systems, such as [28], address the exploration of linked data. Hence, they are not suitable for analytics on the multidimensional model.

As for question answering, existing systems are well understood and differentiate for the knowledge required to formulate the query and for the generative approach. Domain agnostic approaches solely rely on the database schema. NaLIR [20] translates natural language queries into dependency trees [25] and transforms promising trees by brute force until a valid query can be generated. In our approach, we rely on n-grams instead of dependency trees since the latter cannot be directly mapped to entities in the metadata repository (i.e., they require tree manipulation) and are sensible to the query syntax (e.g., *"sum unit sales"* and *"sum the unit sales"* produce two different trees with the same meaning). SQLizer [36] generates templates over the issued query and applies a "repair" loop until it generates queries that can be obtained using at most a given number of changes from the initial template. Domain-specific approaches add semantics to the translation process through domain-specific ontologies and ontology-to-database mappings. SODA [5] uses a simple but limited keyword-based approach that generates a reasonable and executable SQL query based on the matches between the input query and the database metadata, enriched with domain-specific ontologies. ATHENA [29] and its recent extension [31] map natural language into an ontology representation and exploit mappings crafted by the relational schema designer to resolve SQL queries. Analyza [7] integrates the domain-specific ontology into a "semantic grammar" (i.e., a grammar with placeholders for the typed concepts such as measures, dimensions, etc.) to annotate and finally parse the user query. Additionally, Analyza provides an intuitive interface facilitating user-system interaction in spreadsheets. Unfortunately, by relying on the definition of domain-specific knowledge and mappings, the adoption of these approaches is not plug-and-play as an ad-hoc ontology is rarely available and is burdensome to create.

In the area of business intelligence, the road to conversation-driven OLAP is not paved yet. The recommendation of OLAP sessions to improve data exploration has been well-understood [2] also in domains of unconventional contexts [11] where hand-free interfaces are mandatory. Recommendation systems focus on integrating (previous) user experience with external knowledge to suggest queries or sessions, rather than providing smart interfaces to BI tools. To this end, personal assistants and conversational interfaces can help users unfamiliar with such tools and SQL language to perform data exploration. However, end-to-end frameworks are not provided in the domain of analytic sessions over multidimensional data. QUASL [19] introduces a QA approach over the multidimensional model that supports analytical queries but lacks both the formalization of the disambiguation process (i.e., how ambiguous results are addressed) and the support to OLAP sessions (with respect to QA, handling OLAP sessions requires to manage previous knowledge from the `Log` and to understand whether the issued sentence refines the previous query or is a new one). Complementary to COOL, [34] recently formalized the vocalization of OLAP results.

To summarize, the main differences between our approach and previous work are the following.

1. The implementation of an end-to-end general-purpose *dialog-driven* framework named COOL, supporting full-fledged OLAP sessions.
2. The definition of the framework's functional architecture and the formalization of its steps.
3. The enforcement of multidimensional constraints on GPSJ queries by the means of (i) a formal grammar ensuring syntactic validity, and (ii) a type checker ensuring consistency (e.g., it is impossible to assign the "sum" aggregation operator to a non-additive measure).
4. A plug-and-play implementation that allows COOL to run on top of existing data warehouses with no impact on it; furthermore, the integration with external knowledge is supported but not mandatory.

Being a NLI, this work builds on fundamental notions of Natural Language processing (NLP), such as tokenization, parsing, and disambiguation. We refer the reader to a referential contribution in the areas of Natural Language Processing [24] to further delve into this subject. With respect to the single steps of the approach, we remark the following differences.

- Our parsing technique Section 5 is based on a novel formal grammar, used to interpret OLAP queries and OLAP operations expressed in natural language and to obtain parse trees. Among related approaches: [5] relies on a limited keyword-based technique; [29] uses a similar technique, even though its grammar is not specific to OLAP and is aimed at mapping text tokens to ontology concepts; [20, 36, 7] rely on standard natural language parsers where the produced dependency trees are sensible to small linguistic variations in the user query.

- By analyzing the specificities of natural language interfaces in the OLAP context, we formally define and retrieve the ambiguities that may arise in OLAP queries and operations (Section 6). With respect to related approaches, this allows us to discover and manage ambiguities that are related more to the OLAP metaphor rather than to natural language per se.

- Related work deals differently with the disambiguation activity. [36] relies on a completely automatic and probabilistic method to resolve ambiguities, whereas [7] always prompts the user. [20] adopts a hybrid approach by distinguishing easy ambiguities (automatically solvable) from hard ambiguities (requiring user interaction); [5, 29] also adopt an automatic solving technique, but by relying on a domain-specific ontology. COOL integrates these methods by i) implicitly solving some ambiguities (also thanks to the OLAP specific parsing and annotation techniques), ii) inferring the solution from the log whenever possible, and iii) by asking the user when no solution has been found (Section 7).
- Every related approach defines its own way to generate SQL queries from the data structure parsed from the user query. The technique we discuss in Section 8 is specifically tied to our novel grammar.
- The techniques we adopt for tokenization and mapping (Section 4) are not specific to OLAP; nonetheless, a detailed explanation of tokenization and mapping is important to introduce basic concepts that are later used and to ensure the reproducibility of the approach.

We sketched the idea of Conversational OLAP in [10]; this paper largely extends the previous contribution by (i) proposing and implementing a solution for the interpretation and disambiguation of OLAP operators, (ii) providing a log-based ambiguity resolution mechanism that automatically resolves ambiguities by learning the most frequent disambiguations, (iii) providing a visual interface to handle the interaction, designed on top of the DFM model (see Section 9), and (iv) carrying out extensive tests with real users to assess the usability of COOL.

11. Conclusions and Future Works

In this paper, we proposed COOL, a conversational OLAP framework supporting the translation of a natural language conversation into an OLAP session. COOL supports both the interpretation of GPSJ queries and OLAP operators.

Besides proposing a technical solution and a reference architecture, the contribution of the paper lies in the discussion of specific issues related to conversational OLAP systems. Since its conception, OLAP analysis aims to allow users to analyze data without requiring technological skills. Conversational OLAP represents a step forward in this direction. We believe that conversational OLAP can be particularly useful in the context of hand-free applications such as the ones we proposed in [13]: adding conversational capabilities to an augmented OLAP solution would be highly desirable whenever the user is working in the field (e.g., a warehouse or a factory) and is not in front of a computer. To close the loop, we are working towards enabling access to OLAP results in a conversational and hand free fashion. The idea is to create *query summaries* that can fit an augmented reality device or that can be returned through a short vocal message. In this direction, the main research challenges are about (i) identifying the most interesting information out of the possibly large amount of information returned by the query (i.e., applying mining techniques to reduce the cardinality of the returned results as in [13, 14]); and (ii) identifying the right storytelling and the user-system interactions [34].

References

[1] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. 2019. A comparative survey of recent natural language interfaces for databases. *VLDB J.* 28, 5 (2019), 793–819.

[2] Julien Aligon, Enrico Gallinucci, Matteo Golfarelli, Patrick Marcel, and Stefano Rizzi. 2015. A collaborative filtering approach for recommending OLAP sessions. *Decis. Support Syst.* 69 (2015), 20–30.

[3] Paul Alpar and Michael Schulz. 2016. Self-Service Business Intelligence. *Bus. Inf. Syst. Eng.* 58, 2 (2016), 151–155.

[4] John C. Beatty. 1982. On the relationship between LL(1) and LR(1) grammars. *J. ACM* 29, 4 (1982), 1007–1022.

[5] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. 2012. SODA: Generating SQL for Business Users. *PVLDB* 5, 10 (2012), 932–943.

[6] Kenneth Brant and Svetlana Sicular. 2018. Hype Cycle for Artificial Intelligence, 2018. `http://www.gartner.com/en/documents/3883863/hype-cycle-for-artificial-intelligence-2018`. Published: 24 July 2018, Accessed: 2019-06-21.

[7] Kedar Dhamdhere, Kevin S. McCurley, Ralfi Nahmias, Mukund Sundararajan, and Qiqi Yan. 2017. Analyza: Exploring Data with Conversation. In *Proc. IUI.* ACM, New York, NY, USA, 493–504.

[8] Krista Drushku, Julien Aligon, Nicolas Labroche, Patrick Marcel, and Verónika Peralta. 2019. Interest-based recommendations for business intelligence users. *Inf. Syst.* 86 (2019), 79–93.

[9] Lorena Etcheverry and Alejandro A. Vaisman. 2012. QB4OLAP: A Vocabulary for OLAP Cubes on the Semantic Web. In *Proc. COLD (CEUR Workshop Proceedings)*, Vol. 905. CEUR-WS.org, Aachen, DEU.

[10] Matteo Francia, Enrico Gallinucci, and Matteo Golfarelli. 2020. Towards Conversational OLAP. In *Proc. DOLAP@EDBT/ICDT (CEUR Workshop Proceedings)*, Vol. 2572. CEUR-WS.org, Copenhagen, Denmark, 6–15.

[11] Matteo Francia, Matteo Golfarelli, and Stefano Rizzi. 2019. Augmented Business Intelligence. In *Proc. DOLAP@EDBT/ICDT (CEUR Workshop Proceedings)*, Vol. 2324. CEUR-WS.org, Lisbon, Portugal, 1–10.

[12] Matteo Francia, Matteo Golfarelli, and Stefano Rizzi. 2020. A-BI$^+$: A framework for Augmented Business Intelligence. *Inf. Syst.* 92 (2020), 101520.

[13] Matteo Francia, Matteo Golfarelli, and Stefano Rizzi. 2020. Summarization and visualization of multi-level and multi-dimensional itemsets. *Inf. Sci.* 520 (2020), 63–85.

[14] Matteo Golfarelli, Simone Graziani, and Stefano Rizzi. 2014. Shrink: An OLAP operation for balancing precision and size of pivot tables. *Data & Knowledge Engineering* 93 (2014), 19–41.

[15] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. 1998. The Dimensional Fact Model: A Conceptual Model for Data Warehouses. *Int. J. Cooperative Inf. Syst.* 7, 2-3 (1998), 215–247.

[16] Ramanathan V. Guha, Vineet Gupta, Vivek Raghunathan, and Ramakrishnan Srikant. 2015. User Modeling for a Personal Assistant. In *Proc. WSDM*. ACM, New York, NY, USA, 275–284.

[17] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-Query Processing in Data Warehousing Environments. In *Proc. VLDB*. Morgan Kaufmann, San Francisco, CA, USA, 358–369.

[18] iConsulting S.p.A. 2020. Indyco wewb site. `https://www.indyco.com/`.

[19] Nicolas Kuchmann-Beauger, Falk Brauer, and Marie-Aude Aufaure. 2013. QUASL: A framework for question answering and its Application to business intelligence. In *Proc. RCIS*. IEEE, Paris, France, 1–12.

[20] Fei Li and H. V. Jagadish. 2016. Understanding Natural Language Queries over Relational Databases. *SIGMOD Record* 45, 1 (2016), 6–13.

[21] Yujian Li and Bi Liu. 2007. A Normalized Levenshtein Distance Metric. *IEEE Trans. Pattern Anal. Mach. Intell.* 29, 6 (2007), 1091–1095.

[22] Yunyao Li and Davood Rafiei. 2017. Natural Language Data Management and Interfaces: Recent Development and Open Challenges. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017*. ACM, Chicago, IL, USA, 1765–1770.

[23] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. 2016. Making the Case for Query-by-Voice with EchoQuery. In *Proc. SIGMOD*. ACM, New York, NY, USA, 2129–2132.

[24] Christopher D. Manning and Hinrich Schütze. 2001. *Foundations of statistical natural language processing*. MIT Press.

[25] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proc. ACL (System Demonstrations)*. The Association for Computer Linguistics, Baltimore, MD, USA, 55–60.

[26] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[27] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Proceedings of TPCTC*. Springer, Lyon, France, 237–252.

[28] Amrita Saha, Mitesh M. Khapra, and Karthik Sankaranarayanan. 2018. Towards Building Large Scale Multimodal Domain-Aware Conversation Systems. In *Proc. AAAI*. AAAI Press, New Orleans, Louisiana, USA, 696–704.

[29] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *PVLDB* 9, 12 (2016), 1209–1220.

[30] David S Sawicki and William J Craig. 1996. The democratization of data: Bridging the gap for community groups. *Journal of the American Planning Association* 62, 4 (1996), 512–523.

[31] Jaydeep Sen, Fatma Ozcan, Abdul Quamar, Greg Stager, Ashish R. Mittal, Manasa Jammi, Chuan Lei, Diptikalyan Saha, and Karthik Sankaranarayanan. 2019. Natural Language Querying of Complex Business Intelligence Queries. In *Proc. SIGMOD*. ACM, New York, NY, USA, 1997–2000.

[32] Radwa El Shawi, Mohamed Maher, and Sherif Sakr. 2019. Automated Machine Learning: State-of-The-Art and Open Challenges. *CoRR* abs/1906.02287 (2019).

[33] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[34] Immanuel Trummer, Yicheng Wang, and Saketh Mahankali. 2019. A Holistic Approach for Query Evaluation and Result Vocalization in Voice-Based OLAP. In *Proc. SIGMOD*. ACM, Amsterdam, The Netherlands, 936–953.

[35] Alejandro A. Vaisman and Esteban Zimányi. 2014. *Data Warehouse Systems - Design and Implementation*. Springer.

[36] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL* 1, OOPSLA (2017), 63:1–63:26.

[37] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers Comput. Sci.* 10, 3 (2016), 399–417. `https://doi.org/10.1007/s11704-015-5900-5`

[38] Kaizhong Zhang and Dennis E. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262.