Generating Succinct Descriptions of Database Schemata for Cost-Efficient Prompting of Large Language Models

Immanuel Trummer
Cornell University
Ithaca, New York, USA
itrummer@cornell.edu

ABSTRACT

Using large language models (LLMs) for tasks like text-to-SQL translation often requires describing the database schema as part of the model input. LLM providers typically charge as a function of the number of tokens read. Hence, reducing the length of the schema description saves money at each model invocation. This paper introduces Schemonic, a system that automatically finds concise text descriptions of relational database schemata. By introducing abbreviations or grouping schema elements with similar properties, Schemonic typically finds descriptions that use significantly fewer tokens than naive schema representations.

Internally, Schemonic models schema compression as a combinatorial optimization problem and uses integer linear programming solvers to find guaranteed optimal or near-optimal solutions. It speeds up optimization by starting optimization from heuristic solutions and reducing the search space size via pre-processing. The experiments on TPC-H, SPIDER, and Public-BI demonstrate that Schemonic reduces schema description length significantly, along with fees for reading them, without reducing the accuracy in tasks such as text-to-SQL translation.

PVLDB Reference Format:
Immanuel Trummer. Generating Succinct Descriptions of Database Schemata for Cost-Efficient Prompting of Large Language Models. PVLDB, 17(11): 3511 - 3523, 2024.
doi:10.14778/3681954.3682017

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/itrummer/schemacompression.

1 INTRODUCTION

Large language models (LLMs) such as GPT-4 have a wide range of applications in the context of data management, including tasks like text-to-SQL translation as well as information extraction. Quite often, solving such tasks requires describing the schema of a relational database to the LLM as part of the input prompt (describing the task to solve as natural language text to the model). Language models like OpenAI's GPT or Anthropic's Claude are nowadays able to process large amounts of input, up to hundreds of pages of text. In principle, this enables their use even for databases with large schemata. However, doing so is expensive since LLM providers typically charge processing fees that are proportional to the length of the input (and output) text, measured as the number of tokens (the atomic units used by the LLM for text representation)[1]. This paper addresses the problem of generating concise descriptions of database schemata, suitable as input for LLMs. By reducing the size of the schema description, users save significant amounts of money in each LLM invocation that refers to the compressed schema.

Example 1.1. Text-to-SQL translation is a classical use case for large language models. In this scenario, the model input (the prompt) integrates the text question to translate, as well as a description of the database schema [12]. A longer schema description increases the number of tokens in the prompt and, therefore, for providers such as OpenAI, Anthropic, or Cohere, the cost for each text-to-SQL translation. A common method [33] is to describe schemata by their DDL commands, shown for an example schema in Figure 1a. Using the more concise description of the same schema in Figure 1b instead decreases costs. It uses multiple levels of nesting to describe the database schema. The outermost pair of brackets contains columns associated with the `Students` table. Inner brackets group columns that share the same type (e.g., `varchar(255)`) or the same constraints (`NOT NULL`). As demonstrated in Section 8.3, large language models are able to understand such schema descriptions. Figure 1c reduces costs further by introducing abbreviations. More precisely, it introduces the asterisk symbol (`*`) to abbreviate the column name prefix `UniStu_`. Schemonic takes raw schemata as in Figure 1a as input and produces more concise schema descriptions like the one in Figure 1c as output, enabling cost savings.

This paper introduces Schemonic (a portmanteau of "schema" and "laconic"), a system that finds concise text descriptions of database schemata automatically. Schemonic exploits the opportunities to shorten schema descriptions, illustrated in Figure 1. It models schema compression as a combinatorial optimization problem which (as shown in Section 7) is NP-hard. Often, the schema of a database changes only infrequently. Hence, a concise schema description, generated once, can be reused often. This makes it worthwhile to apply even expensive optimization methods to find solutions with formal optimality or near-optimality guarantees. Motivated by this insight, Schemonic models schema compression as an integer linear programming (ILP) problem and applies sophisticated ILP solvers to find solutions.

Given a new schema to compress, Schemonic first analyzes the schema to identify candidate substrings for abbreviations. Next,

[1]Per-token pricing is used by major providers such as OpenAI, Anthropic, Cohere, AI21, as well as by providers such as IBM and Anyscale that offer open-source models in the Cloud. Schemonic does not directly apply to scenarios with a different cost model, e.g., when running language models locally.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682017

```
CREATE TABLE Students(
    UniStu_ID int primary key,
    UniStu_Name varchar(120) NOT NULL,
    UniStu_Street_Name varchar(255) NOT NULL,
    UniStu_Street_Nr int NOT NULL,
    UniStu_City varchar(255) NOT NULL
);
```

(a) SQL commands creating schema: requires 93 tokens with GPT.

```
Table Students(
    UniStu_ID(int primary key)
    NOT NULL(varchar(255)(
        UniStu_Name UniStu_Street_name UniStu_City)
    int(UniStu_Street_Nr)
))
```

(b) First associated schema text: requires 54 tokens with GPT (after removing tabs and newlines added for readability).

```
* means UniStu_
Table Students(
    *ID(int primary key)
    NOT NULL(varchar(255)(
        *Name *Street_name *City)
    int(*Street_Nr))
)
```

(c) Second associated schema text: requires 39 tokens with GPT (after removing tabs and newlines added for readability).

Figure 1: Example schema and associated representations.

it identifies groups of columns with similar properties, enabling a reduction in search space size for the following optimization steps. Then, Schemonic calculates heuristic solutions and associates search space parts with heuristic priorities. Finally, Schemonic transforms the schema compression problem into an instance of ILP which is solved by a corresponding solver. Heuristic solutions, priority values, and column groups are used to provide the ILP solver with hints. These steps speed up optimization (as shown in the experiments) without compromising optimality guarantees. The ILP solution is then transformed into a concise text description of the input schema. As compression is based on a structured schema representation with sound transformations, the resulting schema description is guaranteed to be equivalent to the input schema.

The experiments compare Schemonic to several baselines on database schemata from the PublicBI [14], TPC-H, and SPIDER [38] benchmarks. On average, Schemonic reduces fees for reading schema descriptions via language models by factor two. At the same time, compression does not negatively impact the ability of language models like GPT to translate questions to SQL queries.

This paper's original scientific contributions are the following:

- The paper introduces the problem of schema compression for LLM prompting.
- It proposes an approach for schema compression based on integer linear programming.
- It formally analyzes the schema compression problem and the proposed solution.
- It reports experimental results comparing the proposed approach to baselines.

The remainder of this paper is organized as follows. Section 2 introduces the problem model and associated terminology. Section 3 gives a high-level overview of the Schemonic system and the context in which it is used. Section 4 describes how Schemonic identifies potentially useful abbreviations. Section 5 describes the transformation from schema compression to ILP and Section 6 describes several optimizations, enabling Schemonic to find ILP solutions faster. Section 7 formally proves the correctness of the ILP transformation and analyzes the complexity of the problem. Section 8 discusses experimental results and Section 9 prior work.

2 FORMAL MODEL

We introduce schema compression and related terminology.

Definition 2.1 (Schema). A schema s is associated with a set of tables, denoted as $s.tables$. Each table t is associated with a name and a set of columns, referred to as $t.name$ and $t.columns$ respectively. Each column c is associated with a name (denoted as $c.name$) and a set of annotations ($c.annotations$), describing the column type or applicable constraints (e.g., column type, uniqueness or not-null constraints, or single-column primary key constraints). Optionally, a table t may be associated with constraints ($t.constraints$) that refer to column groups (e.g., multi-column primary key constraints or multi-column foreign key constraints).

Definition 2.2 (Identifier, Token). Given a schema s, we denote identifier tokens (short: identifiers) of s as $ID(s)$. Eligible identifiers are the set of table names, prefixed by the keyword "table ", the set of column names for each table (the name alone in case of non-ambiguous column names, otherwise the column names prefixed by the associated table name), and the set of annotations used for columns or tables. General tokens include the identifier tokens as well as opening and closing brackets.

Tokens according to the prior definition may or may not correspond to tokens used by specific language models.

Example 2.3. Denote by s the schema created in Figure 1a. Elements in the set $ID(s)$ include `Table Students`, as well as all column names such as `UniStu_ID` and `UniStu_Name`. Also, it includes `int`, `primary key`, `NOT NULL`, `varchar(255)`, and all other column annotations used.

Definition 2.4 (Description Syntax). The empty string ("") is a syntactically valid description. If d_1 and d_2 are valid descriptions then $d_1 d_2$ (i.e., their concatenation) is a valid description. Let t be an identifier token for the relevant schema and d a valid description. Then, $t(d)$ is also a valid description.

We expand the scope of the ID function and also denote by $ID(d)$ the identifier tokens that appear in a schema description d.

Definition 2.5 (Description Semantics). Function $Facts(d)$ denotes a set of facts about a schema that can be inferred from a schema description d. If description d is an empty string, it is $Facts(d) = \emptyset$. If $d = d_1 d_2$ (i.e., the description concatenates descriptions d_1 and d_2), it is $Facts(d) = Facts(d_1) \cup Facts(d_2)$. If $d = t(d')$ then $Facts(d) = Facts(d') \cup \{\{t, id\} | id \in ID(d')\}$.

Definition 2.6 (Accurate Description). A description d of a schema s is accurate iff $Facts(d)$ contains all associations between tables and columns that appear in the schema and connects all tables and columns to all applicable annotations (e.g., type and key constraints). At the same time, the description cannot convey incorrect facts (e.g., incorrect associations between columns and annotations).

Example 2.7. Let d be the schema description illustrated in Figure 1a. Here, all column names appear within the surrounding context `Table Students`. This means facts connecting the table with its columns, e.g., {`Table Students`,`UniStu_ID`}, are contained in $Facts(d)$. For column `UniStu_ID`, all relevant annotations appear in the column context, leading to facts {`UniStu_ID`,`Primary Key`} and {`UniStu_ID`,`int`}. As yet another example, consider the column `UniStu_Name`. As this column appears within two annotation contexts (`NOT NULL` and `varchar(255)`), {`UniStu_Name`,`NOT NULL`} and {`UniSti_Name`,`varchar(255)`} appear in $Facts(d)$. At the same time, note that the description does not introduce any incorrect facts (e.g., erroneous column-table associations or incorrect column annotations). Hence, the schema description is accurate.

Definition 2.8 (Token Mapping). A representation function maps tokens to a text representation, possibly shortening the token name. Consider a set G of such functions. A token mapping for schema s is a function $\mu : ID(s) \mapsto G$ that maps each token to a function used to represent it.

Example 2.9. Figure 1c uses a representation function, g_* in the following, that replaces all occurrences of `UniStu_` by the symbol $*$ (which does not otherwise appear in the schema). The identity function, g_I in the following, is a special case and represents each token by its name. Given the text in Figure 1c, we can infer that $\mu($`UniStu_Name`$) = g_*$ while $\mu($`int`$) = g_I$ for the token mapping μ.

Definition 2.10 (Schema Text). We can map a schema description d with associated token mapping μ to a text description, $Text(d, \mu)$ as follows. Let $G = \cup_{id \in ID(d)} \mu(id)$ the set of representation functions used in μ. It is $Text(d, \mu) = FText(G)SText(d, \mu)$ where $FText$ describes all functions in G as text and $SText$ describes the schema, using the aforementioned functions. It is $SText(d, \mu) =$ "" if d is empty, $SText(t, \mu) = \mu(t)(t)$ for any identifier token t (this expression first maps t to a representation function and then applies that function to t), $SText(d'd'', \mu) = SText(d', \mu)SText(d'', \mu)$, and $SText(d(d'), \mu) = SText(d, \mu)$"("$SText(d', \mu)$")".

Example 2.11. Figure 1c introduces function g_* (replacing occurrences of `UniStu_` by an asterisk symbol) before describing the schema itself. Note that the identity function does not require further explanations (i.e., the definition text is empty).

Definition 2.12 (Size). The aforementioned tokens are in general not equivalent to the tokens used by large language models. Assuming a fixed target model, we use $Size(text)$ to denote the number of tokens used by the model to represent the given $text$.

We are now ready to introduce the problem solved by Schemonic.

Definition 2.13 (Schema Compression). Given a schema s and a set $\mathcal{G}$ of eligible representation functions, find an accurate schema description d and associated token mapping μ, mapping tokens to some subset of $\mathcal{G}$, such that $Text(d, \mu)$ has minimal size (i.e., find $\arg\min_{d,\mu} Size(Text(d, \mu))$).

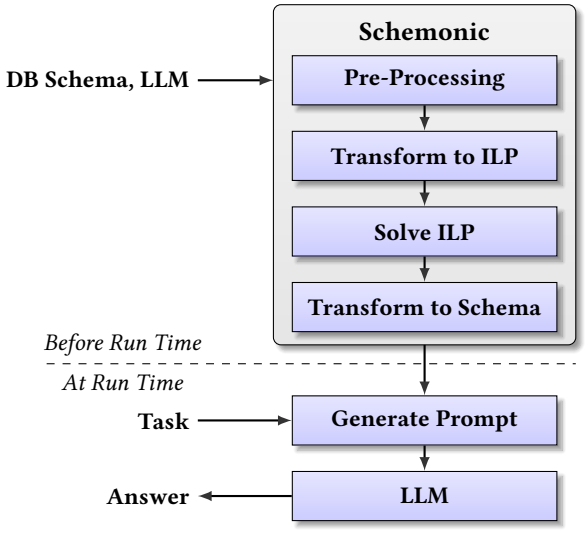

Figure 2: Schema compression and its context.

3 SYSTEM OVERVIEW

Section 3.1 gives a high-level overview of the schema compression approach and the context in which it is used. Section 3.2 describes the top-level algorithm in more detail.

3.1 System Context

Figure 2 shows an overview of the "Schemonic" system and its context. The input is a database schema to compress, as well as a target LLM. Schemonic aims at finding a text representation of the schema that minimizes the number of tokens used, according to the tokenizer used by the target LLM. Internally, Schemonic performs several pre-processing steps, then transforms the schema compression problem into an ILP instance. It solves the resulting ILP via a corresponding solver (currently, it uses the Gurobi solver). In doing so, it considers user-specified bounds on optimization overheads (e.g., a time limit). The resulting solution is transformed into a schema representation in text form. This description is concise and typically uses fewer tokens than the original.

The resulting schema description is meant to be used as part of the input prompt for an LLM, informing the LLM about the database structure without using more tokens than necessary. Reducing the number of tokens is typically equivalent to reducing monetary processing fees. Providers of LLMs such as OpenAI calculate their fees as a function of the number of tokens read and generated. The schema description can be used for any task to be solved by the LLM that relates to the input database. Examples include text-to-SQL translation [23, 38, 40], schema matching [4] and data wrangling tasks [26], or structured information extraction [7].

Generating optimized schema descriptions can be expensive (depending on the size of the database schema and the constraints on optimization overheads). However, assuming that the database schema changes only rarely, the same compressed descriptions can be reused many times. The latest generation of language models is typically used in zero or few-shot scenarios. This means that all relevant information, including the schema description, have

Algorithm 1 Function generating concise schema descriptions.

```
1:  Input: A schema s to compress, a target model LLM, the number k of prefixes to consider, and the maximal nesting depth L.
2:  Output: Compressed representation.
3:  function CompressSchema(s, LLM, k, L)
4:      // Generate candidate prefixes
5:      P ←CandidatePrefixes(s, k)
6:      // Merge columns with same annotations
7:      s ←MergeColumns(s)
8:      // Generate greedy solution as start
9:      g ←Greedy(s)
10:     // Transform to integer linear program
11:     ilp ←TransformToILP(s, LLM, L, P, g)
12:     // Optimize using ILP solver
13:     o ←ILPSolver(ilp)
14:     // Transform solution to description text
15:     text ←ExtractDescription(o)
16:     // Return optimized description
17:     return text
18: end function
```

to be included into the prompt in each invocation. Hence, using a compressed schema reduces the per-invocation costs.

3.2 Main Algorithm

Algorithm 1 describes the schema compression process in more detail. The input is a schema s (to be described concisely), a target model LLM, as well as two configuration parameters k and L. Those parameters restrict the maximal number of prefixes considered during optimization (k) as well as the maximal nesting depth for the generated schema descriptions (L). For schema compression, Schemonic considers opportunities to abbreviate common prefixes (e.g., of column or table names) by newly introduced symbols (which do not appear in the schema otherwise). Algorithm 1 generates a set of potentially useful prefixes in Line 5. To reduce the size of the search space for compression, the algorithm merges together columns with equivalent annotations into column groups (Line 7). Finally, it generates a first schema description using a simple greedy algorithm (Line 9), described in Section 6.2. While this solution is not guaranteed to be optimal (and, as shown in the experiments, is typically sub-optimal) it provides a useful starting point for the ILP solver. In Line 11, Algorithm 1 transforms the schema compression instance into an ILP instance. It solves this instance by a corresponding ILP solver (Line 13). Depending on user constraints, this step ends once an optimal solution is found or once thresholds on optimization overheads (e.g., time limits) are reached. Finally, Algorithm 1 extracts an optimized schema description from the ILP solution.

4 RANKING PREFIXES

To reduce the schema description size, Schemonic considers options to shorten token names by abbreviating common prefixes. This section describes the method used by Schemonic to identify potentially useful prefixes. Considering more prefixes increases the size of the ILP that needs to be solved later on. Hence, Schemonic aims at identifying a limited number of prefixes with high expected utility.

Algorithm 2 Generate candidates for shortcuts (prefixes).

```
1:  Input: A database schema s.
2:  Output: Map prefixes to frequency.
3:  function PrefixFrequency(s)
4:      // Retrieve list of identifiers
5:      I ←IdentifierList(s)
6:      // Initialize frequency counter
7:      F ← {id : 0|id ∈ I}
8:      // Iterate over identifiers
9:      for id ∈ I do
10:         // Iterate over prefix length
11:         for l ← 1..|id| do
12:             // Extract prefix ...
13:             p ← id[: l]
14:             // ... and count it
15:             F[p] ← F[p] + 1
16:         end for
17:     end for
18:     return F
19: end function

20: Input: Map F from prefixes to frequencies.
21: Output: Pruned map from prefixes to counts.
22: function Prune(F)
23:     // Prune out prefixes with a single occurrence
24:     F ← {⟨p, f⟩|⟨p, f⟩ ∈ F, f > 1}
25:     // Iterate over prefixes and frequencies
26:     for ⟨p, f⟩ ∈ F do
27:         // Iterate over substring length
28:         for l ← 1..|p| do
29:             // Retrieve substring
30:             q ← p[: l]
31:             // Is substring not more common?
32:             if ⟨q, g⟩ ∈ F|g ≤ f then
33:                 // Prune dominated substring
34:                 F ← {⟨p, f⟩|⟨p, f⟩ ∈ F, q ≠ p}
35:             end if
36:         end for
37:     end for
38:     // Return pruned prefixes
39:     return F
40: end function

41: Input: A database schema s, number of prefixes k.
42: Output: A set of common prefixes.
43: function CandidatePrefixes(s, k)
44:     // Count prefix frequency
45:     F ←PrefixFrequency(s)
46:     // Prune prefixes
47:     F ←Prune(F)
48:     // Return most frequent prefixes
49:     return k most frequent prefixes in F
50: end function
```

Algorithm 2 shows how Schemonic identifies potentially useful prefixes. The input to Function CandidatePrefixes is the database schema s, along with the number of prefixes to generate (k). Considering more prefixes may possibly lead to more optimal solutions later on. However, considering more prefixes also increases the size of the associated ILP and therefore optimization time.

Function CandidatePrefixes in Algorithm 2 first counts the number of occurrences for each prefix. To that purpose, Function PrefixFrequency first retrieves the list of relevant identifiers. Function IdentifierList (we omit pseudo-code due to space restrictions) retrieves the list of identifier tokens used in the original (i.e., SQL) description of the database schema. This list contains duplicates which is important to identify frequent prefixes. Next, Function PrefixFrequency iterates over all identifier prefixes. To specify prefixes, the function uses a notation inspired by the Python programming language, i.e., $id[:l]$ are the first l characters from the start of string id (or the entire string if it has less than l characters). Each prefix occurrence is counted in a dictionary (F), mapping prefixes to the associated count.

Next, Function CandidatePrefixes prunes prefixes by discarding dominated prefixes. Function Prune first discards prefixes that do not appear repeatedly. Introducing shortcuts for prefixes as a part of the schema description also consumes tokens. Hence, doing so for prefixes that cannot be reused is wasteful. Next, Function Prune compares prefixes and prunes them out if the following condition is met. A prefix is dominated if there is another prefix that is longer and appears at least as often. This avoids situations in which the system retrieves multiple prefixes of varying length for the same group of identifiers (here, typically, it is optimal to use the longest prefix for the associated token group).

The remaining prefixes are ordered by their occurrence frequency. The k most frequent prefixes are returned.

5 ILP TRANSFORMATION

Schemonic transforms the problem of schema compression to ILP problems. The resulting problems can be solved via ILP solvers such as Cplex or Gurobi. The optimal solution to the ILP instance can be transformed back into a schema description of minimal size. This section shows how to transform an instance of schema compression into an ILP instance. An ILP instance is characterized by a set of integer variables, a set of linear constraints on those variables, and a linear objective function of those variables. Section 5.1 describes the variables used and their semantics, Sections 5.2 describes different categories of constraints on those variables, and Section 5.3 describes the objective function. Section 5.4 discusses how to extract a schema representation from the optimal ILP solution.

5.1 Variables

Table 1 summarizes the variables used by the ILP instances. All variables are integer variables with binary domain (i.e., the only admissible values are one and zero). Table 1 summarizes variables into groups, covering different aspects of the schema description. Variables x_{it} capture the schema description itself (but not yet the token mapping). They describe the sequence of tokens selected for the description, including brackets. Tokens are divided into consecutive slots. Each slot contains up to one identifier token (e.g., a column or table name) and up to one (opening or closing) bracket. As discussed in more detail later, introducing slots (as opposed to representing single tokens separately) makes it easier to impose constraints between tokens and brackets (e.g., requiring that opening brackets are combined with tokens). Each variable x_{it} captures whether or not token t is in slot number i ($x_{it} = 1$ iff the token is included).

Table 1: Variables of integer linear program with associated semantics (all variables are binary).

Variable	Semantics
x_{it}	1 iff token t in slot number i
r_{itg}	1 iff function g used for token t at position i
h_g	1 iff we have function g in prompt description
e_i	1 iff no tokens are selected at position i
a_{it}	1 iff token t added to context at position i
c_{ilt}	1 iff token t in context at position i and layer l
$m_{it_1t_2}$	1 iff fact connecting t_1, t_2 mentioned at i
$f_{t_1t_2}$	1 iff fact connecting tokens t_1, t_2 mentioned

Tokens can be represented differently, either via the original token name alone or via a (shortening) transformation. Specifically, Schemonic considers shortening token names by abbreviating common prefixes. Variables r_{itg} capture the representation of selected tokens for slots i, tokens t, and representation functions g (i.e., using the original token name or abbreviating a prefix via a symbol). It is $r_{itg} = 1$ iff function g is used to represent the selected token. All functions used to represent tokens must be introduced. Variables h_g indicate whether function g is introduced (if so, the schema text description contains a corresponding text snippet at the beginning).

Variables e_i, a_{it}, and c_{ilt} are auxiliary variables whose values are directly derived from the values of variables x_{it}. Variables e_i are set to one iff slot number i is empty. As discussed in more detail in Section 2, enclosing a group of tokens (e.g., column names) within brackets, prefixed by another token (e.g., a data type), implicitly associates all tokens in the group with the preceding token. In those cases, we also say that the token group appears within the context of the initial token. If tokens are enclosed by multiple pairs of brackets, the context may contain more than one token (one token for each pair of brackets). Variables c_{ilt} keep track of the surrounding context for each slot. It divides context tokens into layers such that the outermost brackets are associated with the first layer, the innermost brackets are associated with the last used layer. The maximal number of usable layers L (one of the input parameters in Algorithm 1) is equivalent to the maximal number of nested brackets. It is $c_{ilt} = 1$ iff context layer l at slot i contains token t. Connecting a token with an opening bracket adds that token to the first unused context layer. Variable a_{it} is set to one iff token t is added to the context in slot i (i.e., the token will appear in the context starting from the next slot).

Variables $m_{it_1t_2}$ and $f_{t_1t_2}$ are used to associate a schema description with semantics. As discussed in Section 2, facts connect token pairs (e.g., associating a column name with a specific data type). Variables $m_{it_1t_2}$ indicate whether a fact connecting token t_1 and t_2 was mentioned at slot i. More precisely, the mention entails token

Table 2: Values of decision variables for first six slots of the schema description from Figure 1c.

Slot	Tokens	Added	Layer 1	Layer 2	Representation	Facts
1	`Table Students(`	`Table Students`			`Table Students`	
2	`UniStu_ID(`	`UniStu_ID`	`Table Students`		`*ID`	`Table Students-UniStu_ID`
3	`int`		`Table Students`	`UniStu_ID`	`int`	`UniStu_ID-int`
4	`primary key`		`Table Students`	`UniStu_ID`	`primary key`	`UniStu_ID-primary key`
5	`)`		`Table Students`			
6	`NOT NULL(`	`NOT NULL`	`Table Students`		`NOT NULL`	

t_2 appearing in slot i whereas token t_1 is contained in one of the context layers at slot i. A fact is stated if it is mentioned at least once. Variables $f_{t_1t_2}$ indicate whether there is at least one mention of the fact connecting tokens t_1 and t_2.

Example 5.1. Table 2 illustrates the use of the aforementioned variables by an example. The example uses the first part of the schema description from Figure 1c. The first column represents the slot ID. The second column represents tokens selected in each slot (i.e., tokens for which variables x_{it} are set to one). The third column represents tokens added to the context in the corresponding slot (i.e., tokens for which variables a_{it} are set to one). The fourth column represents tokens that appear in the first layer of the context (i.e., tokens for which c_{i1t} is set to one). The fifth column represents tokens that appear in the second layer of the context (i.e., tokens for which $c_{i2t} = 1$). The next column describes the representation used for each selected identifier (i.e., "non-bracket") token. There can be at most one identifier token per slot. The column contains a token representation iff r_{itg} is set to one for the corresponding function g). Finally, the last column describes fact mentions associated with slots. It contains combinations of tokens if the associated variable $m_{it_1t_2}$ is set to one). For instance, the first slot contains an opening bracket in combination with token `Table Students`. This token therefore appears in the context of each of the following slots (since the associated closing bracket is not part of the example anymore). As the context is initially empty, the first context layer contains the table name. The second slot also contains an opening bracket, together with the column name `UniStu_ID`. This token is added to the second context layer in the following slot (since this layer is the first unused layer). In the second slot, as the context contains the table name, column name `UniStu_ID` is implicitly associated with that table (as indicated by the corresponding fact).

5.2 Constraints

Linear constraints ensure that each solution represents a valid schema description (with associated token mapping). The following constraints connect different variables of the same variable group (e.g., different variables x_{it}). Other constraints connect variables from different groups. Figure 3 illustrates dependencies between variables of different groups. Circles in Figure 3 represent variable groups and lines represent constraints connecting variables of different groups. Variable groups e_i, a_{it}, and c_{ilt} are auxiliary variables, set as a function of variables x_{it}. This is represented by corresponding connections in Figure 3. At the same time, variables a_{it} (representing the addition of new tokens to the context) are connected to variables c_{ilt} (representing context). Variables

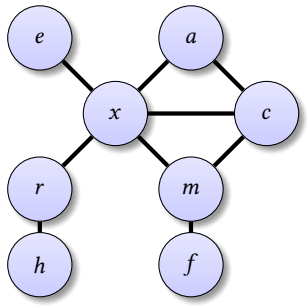

Figure 3: Constraints between variable groups.

$m_{it_1t_2}$ (representing mentions of facts) depend on tokens selected in the current slot (variables x_{it}) and tokens in the current context (variables c_{ilt}). Variables $f_{t_1t_2}$ (representing the statement of a fact anywhere in the description) aggregate the values of variables $m_{it_1t_2}$ (representing a fact mention at a specific slot). Finally, values of variables r_{itg} (capturing the representation of selected tokens) depend on the variables representing tokens selected in specific slots (x_{it}). Admissible values for variables h_g (indicating whether function g is used to represent at least one token) depend only on variables r_{itg} (representing functions used to represent specific tokens).

Table 3 contains all constraints needed to guarantee admissible solutions. Constraint groups are named with IDs from C1 to C28. Implicitly, whenever i appears in a sum or universal quantifier, it runs over all available slot positions. Similarly, t, t_1, and t_2 run over all tokens, except for the notation $t \in ID$ (in which case t is restricted to identifier tokens, excluding brackets). The value domain of l are all context layers whereas g runs over all representation functions. Section 7 contains a formal proof, showing that, together, those constraints imply valid solutions. The remainder of this subsection describes those constraints and gives an intuitive explanation for why they are necessary.

Constraint groups C1 to C5 ensure that value assignments for variables x_{it} and e_i are valid. Specifically, C1 and C5 ensure that each slot contains at most one identifier token and at most one bracket. Constraints C2 and C3 imply that variables e_i (indicating an empty slot) are assigned consistently with the values of x_{it}. C4 ensures that empty slots appear only at the end (thereby avoiding redundant representations of the same schema description). Constraints C6 to C8 ensure correct bracketing. C6 ensures that opening brackets are combined with an identifier token in the same slot (such that all tokens appearing within brackets are implicitly associated with that

Table 3: Constraints of integer linear program representing schema compression problem.

ID	Constraint	Semantics
C1	$\forall i : x_{i"("} + x_{i")"} + e_i \leq 1$	Only one bracket or empty slot
C2	$\forall i : e_i \geq 1 - \sum_t x_{it}$	Slot is empty if no tokens are selected
C3	$\forall i, t : e_i \leq 1 - x_{it}$	No empty slot if any token is selected
C4	$\forall i : e_i \leq e_{i+1}$	All empty slots at the end of prompt
C5	$\forall i : \sum_{t \in ID} : x_{it} \leq 1$	At most one identifier per position
C6	$\forall i : x_{i"("} \leq \sum_{t \in ID} x_{it}$	Must connect opening bracket with identifier
C7	$(\sum_i x_{i"("}) - (\sum_i x_{i")"}) = 0$	Same number of opening and closing brackets
C8	$\forall i : \sum_{j \leq i} (x_{i"("} - x_{i")"}) \geq 0$	Never more closing than opening brackets
C9	$\forall i, t : x_{it} + \sum_l c_{ilt} \leq 1$	Do not select tokens already in context
C10	$\forall i, l : \sum_t c_{ilt} \leq 1$	At most one token per context layer
C11	$\forall i, l : \sum_t c_{ilt} \geq \sum_t c_{(i+1)lt}$	Use context layer consecutively
C12	$\sum_{l,t} c_{0lt} = 0$	Initial context is empty
C13	$\forall i : \sum_{l,t} c_{ilt} + x_{i"("} - x_{i")"} = \sum_{l,t} c_{(i+1)lt}$	Correct number of tokens in context
C14	$\forall i, t : a_{it} \leq x_{i"("}$	No context addition without opening bracket
C15	$\forall i, t : a_{it} \leq x_{it}$	No context addition without selecting token
C16	$\forall i, t : a_{it} \geq x_{i"("} + x_{it} - 1$	Opening bracket and token imply context addition
C17	$\forall i, t : \sum_l c_{(i+1)lt} \geq a_{it}$	Have token in context after adding it
C18	$\forall i, l, t : c_{(i+1)lt} \geq c_{ilt} - x_{i")"}$	Cannot drop context without closing bracket
C19	$\forall i, l, t : c_{(i+1)lt} \leq c_{ilt} + x_{i"("}$	Cannot add context without opening bracket
C20	$\forall i, t_1, t_2 : m_{it_1t_2} \leq \sum_l c_{ilt_1}$	Fact mention requires first token in context
C21	$\forall i, t_1, t_2 : m_{it_1t_2} \leq x_{it_2}$	Fact mention requires second token selected
C22	$\forall i, t_1, t_2 : m_{it_1t_2} \geq (\sum_l c_{ilt_1}) + x_{it_2} - 1$	Fact mention if first token in context and second selected
C23	$\forall t_1 < t_2 : f_{t_1t_2} \leq \sum_i (m_{it_1t_2} + m_{it_2t_1})$	Fact is not stated unless it is mentioned
C24	$\forall i, t_1 < t_2 : f_{t_1t_2} \geq m_{it_1t_2} + m_{it_2t_1}$	Fact is stated if it is mentioned at least once
C25	$\forall \{t_1, t_2\} \in True : f_{t_1t_2} = 1$	True facts need to be stated
C26	$\forall \{t_1, t_2\} \in False : f_{t_1t_2} = 0$	False facts cannot be stated
C27	$\forall i, t \in ID : x_{it} = \sum_g r_{itg}$	Choose one representation for each selected token
C28	$\forall i, t, g : r_{itg} \leq h_g$	Must add explanation of function in prompt to use it

token). C7 ensures that the number of opening and closing brackets is equal whereas C8 ensures admissible ordering of those brackets (ensuring that the number of closing brackets never exceeds the number of opening brackets encountered previously).

Constraints C9 to C19 ensure that variables c_{ilt} accurately represent context assigned by the sequence of opening (and closing) brackets and associated tokens, represented by variables x_{it}. Constraint C9 ensures that tokens already in the context cannot be selected in the associated slot. Constraint C10 ensures that each context layer only represents a single selected token. Furthermore, constraint C11 ensures that context layers are used consecutively. Constraints C12 and C13 ensure that the total number of selected tokens in the context is accurate. C12 ensures no selected tokens in the first context whereas C13 bounds the change, compared to the context of the prior slot, as a function of the number of opening and closing brackets in the previous slot. Constraints C14 to C16 ensure that variables a_{it} (representing the addition of token t to the context after slot i) are set consistently with variables x_{it}. C17 ensures that tokens added in slot i appear in context of the following slot. Constraints C18 and C19 ensure no changes to the context in the absence of opening and closing brackets.

Constraints C20 to C26 refer to the semantics of the selected schema description, ensuring that all relevant facts are mentioned (and no incorrect statements are included). Constraints C20 to C22 ensure that variables $m_{it_1t_2}$ (indicating that tokens t_1 and t_2 are connected in slot i) are set consistently with variables x_{it} and c_{ilt}. C23 and C24 ensure that variables $f_{t_1t_2}$ (indicating whether the connection between tokens t_1 and t_2 is mentioned at least once) are set consistently with variables $m_{it_1t_2}$. C25 and C26 make sure that all true facts are mentioned whereas no incorrect statements are made. The set $True$ refers to all token pairs that are connected by the database schema, namely associations between tables and their columns and between columns and their associations. On the other hand, $False$ contains all token pairs that should not be connected in the schema description, namely connections between columns and tables in which they do not appear as well as connections between columns and types or constraints that do not apply.

Constraints C27 and C28 focus on the representation of tokens. Specifically, C27 ensures that each selected token is mapped to exactly one representation. This representation is either the identity function (i.e., the token is represented by its name) or a function that abbreviates a common prefix by a shorter symbol. In principle,

Table 4: Terms that appear in the objective function.

Term	Semantics
$\sum_{i,t,g}(r_{itg} \cdot Size(g(t)))$	Sum over selected token representations, weighted by length
$\sum_{g}(h_p \cdot Size(FText(g)))$	Sum over selected functions, weighted by description length

the constraints are slightly more permissive, compared to token mappings introduced in Section 2. They allow different occurrences of the same token to map to different representations. However, due to the objective function discussed next, an optimal solution assigns all occurrences to the shortest representation (and uses a single representation if several of them have the same length since introducing more functions increases the text length). Constraint C28 makes sure that all representation functions used at least once are also introduced (represented by variable h_g).

5.3 Objective Function

The goal of optimization is to minimize the length of the schema description, measuring length as the number of tokens required on the target LLM. This is equivalent to minimizing monetary processing fees if using LLMs hosted by providers such as OpenAI. Table 4 summarizes the terms that appear in the objective function to minimize. The schema description contains two types of text: text describing representation functions and text describing the schema itself (and possibly referring to the previously introduced representation functions). Hence, the objective function sums over all selected tokens and used functions, weighted by the length of the associated text description. Note that, in principle, the number of tokens used by the LLM may be slightly lower than the objective function above. This could happen if the LLM introduces single tokens representing multiple tokens used in the schema description. This could be taken into account by a more complex objective, detecting consecutive tokens that are merged by the LLM and reducing the text size accordingly. However, in practice, having single LLM tokens to represent, e.g., combinations of column names is unlikely. The experiments show that the objective above leads to significant cost improvements.

5.4 Extracting Solution

Extracting the schema description from the ILP solution is straightforward. We start by iterating over representation functions and add descriptions at the start of the prompt. Currently, Schemonic supports functions that abbreviate common prefixes. Their description is of the form "[Symbol] means [Prefix]" where [Symbol] is a symbol that can be represented by a single LLM token (and is not otherwise used as part of schema identifiers). [Prefix] represents a prefix that appears frequently in the schema.

Next, we iterate over slots (in ascending order of slot ID), and add for each selected token the selected representation. If slots contain an identifier token and an opening bracket (this is the only permissible case in which a slot contains more than one token), we add the identifier token first and then the opening bracket. Furthermore, we add a whitespace after each slot (except for empty slots and the last used slot). Finally, we add table-level annotations from the input schema (currently, Schemonic only considers annotations on single columns for optimization). The result is a text containing a full schema description.

Algorithm 3 Merge columns with the same annotations.

```
1:  Input: Original schema s.
2:  Output: Schema with merged columns.
3:  function MergeColumns(s)
4:      // Iterate over schema tables
5:      for t ∈ s.tables do
6:          // Retrieve all columns
7:          C ← t.columns
8:          // Get all annotation sets
9:          A ← {c.annotations | c ∈ C}
10:         // Associate annotations with column groups
11:         G ← {⟨C_a, a⟩ | a ∈ A, ∀g ∈ C_a ⊆ C : g.annotations = a}
12:         // Create merged columns
13:         M ← ∅
14:         for ⟨C_a, a⟩ ∈ G do
15:             // Create group name
16:             if |C_a| > 1 then
17:                 n ← "[" + C_a[0].name + ", " + ... + "]"
18:             else
19:                 n ← C_a[0].name
20:             end if
21:             // Add to merged columns
22:             M ← M ∪ {n}
23:         end for
24:         // Replace original columns
25:         t.columns ← M
26:     end for
27:     return s
28: end function
```

6 OPTIMIZATIONS

This section introduces several optimizations, enabling Schemonic to find optimal solutions faster.

6.1 Merging Columns

To reduce the size of the search space, Schemonic merges columns that have the same annotations and are associated with the same table. Intuitively, whenever one of those columns appears in a given context, replacing that column with the entire group does not add any incorrect facts. Also, adding the other columns conveys correct facts about these columns without requiring additional context.

Algorithm 3 shows how Schemonic merges columns into column groups. Given a schema s as input, the algorithm iterates over all schema tables. For each table, it collects the set of annotation sets, considering all table columns. It groups columns by their annotations (Line 11) and creates a set of merged columns (Variable M). Column groups may be singletons, in which case the column remains unchanged. If a group contains multiple columns, its name is derived from the column names in the group, surrounded by square brackets. The list of merged columns is assigned to the table.

6.2 Greedy Algorithm

ILP solvers such as Gurobi allow users to provide initial solutions as a starting point. This can speed up optimization significantly.

Schemonic uses a simple greedy algorithm to generate solutions as a starting point. It iterates over all tables and, for each table, merges columns with equal annotations into column groups, as described in the previous subsection. It generates a description according to the following grammar (represented in Extended Backus-Naur Form with * representing an unlimited number of repetitions of the previous symbol):

<SchemaDef> → <TableDef>*
<TableDef> → Table <TableName>(<ColumnDef>*)
<ColumnDef> → <ColumnGroup>(<Annotation>*)

This solution associates each table with its column groups and each column group with the corresponding annotations. The greedy solution is used to set start values for variables associated with tokens (x_{it} and c_{ilt}). No start values are set for representation variables (r_{itg}).

Example 6.1. For the schema from Figure 1a, the greedy algorithm merges the `UniStu_Street_Name` and `UniStu_City` columns as they have the same annotations. It generates the following description (tabs added for readability):

```
Table Students(
    [UniStu_Street_Name UniStu_City](
        varchar(255) NOT NULL)
    UniStu_ID(int primary key)
    UniStu_Name(varchar(120) NOT NULL)
    UniStu_Street_Nr(int NOT NULL))
```

6.3 Value Hints

Finally, ILP solvers often enable users to specify hints on likely variable values. Such values are prioritized during search (while alternative values are eventually explored as well).

Intuitively, tokens that appear more frequently in the original schema description tend to be more useful for creating context. E.g., creating context for common column annotations (within which all relevant columns can be enumerated) is preferable over creating context for each single column (in which its annotations can be included). Hence, Schemonic sorts tokens by their occurrence frequency. It provides the ILP solver with hints related to infrequent tokens (i.e., tokens that are not within the top ten in terms of occurrence frequency). For those tokens, all related context variables are assigned to zero as default value.

7 FORMAL ANALYSIS

Section 7.1 proves that the transformation from schema compression to ILP, described in Section 5, is correct. Section 7.2 analyzes the complexity of the problem and approach.

7.1 Correctness

The following theorems prove correctness of different aspects of the ILP transformation (using constraints from Table 3).

Theorem 7.1. *An integer linear program solution represents a valid (i.e., syntactically correct) schema description.*

Proof. The proof uses induction over the number of slots. Trivially, if no slots are used then the description is empty and therefore valid. Now, assume that any solution using up to N slots is valid. This implies that solutions with $N+1$ slots are valid as well, as demonstrated next. We distinguish different cases, based on the value assignment in the first slot. If $e_1 = 1$ then no tokens are selected in the first slot (C2, C3) and all the following slots are empty as well (C4). Hence, the description is empty and therefore valid. Next, assume $e_1 = 0$ and $x_{1t} = 1$ for some identifier token t while $x_{1"("} = x_{1")"} = 0$ (i.e., we have no brackets in the first slot). Due to C5, no more than one identifier token can be selected in the first slot. We can therefore decompose the associated description d into $d = td'$ where d' uses N slots. Due to the inductional assumption, d' and therefore td' is valid. Now, assume the first slot contains a bracket. Due to C1, it can only contain a single bracket. Due to C8, the first bracket must be an opening bracket (i.e., $x_{1"("} = 1$). Due to C6, the first slot must select an identifier token t as well (i.e., $x_{1t} = 1$). C7 and C8 imply correct bracketing within the description. Hence, we can find a closing bracket associated with the opening bracket of the first slot. This means we can decompose the description d into $t(d')d''$ where d' and d'' use less than N slots. Also, since we selected matching brackets and since the bracketing of d is correct, the bracketing in d' and d'' is correct, too. □

Theorem 7.2. *Each selected token is mapped to one representation and each representation function is introduced.*

Proof. Due to C27, exactly one representation is used for each selected identifier token. Due to C28, all relevant functions for the selected representations must be introduced. □

Theorem 7.3. *An integer linear program solution assigns context consistently with selected tokens.*

Proof. The proof uses induction over the slot count. For the first slot, the context is empty (C12). This is trivially consistent since there are no preceding slots with opening brackets. Now, assume context variables are consistent until slot number i. We prove that they are consistent for slot $i+1$ as well. We distinguish different cases, based on the content of slot i. If i contains a single token without brackets, the context does not change between slots i and $i+1$. Since $x_{i"("} = x_{i")"} = 0$, all context variables remain unchanged due to C18 and C19 (which is consistent). Now, assume that slot i contains a closing bracket (i.e., $x_{i")"} = 1$). Furthermore, assume that the context at slot i uses l layers (i.e., the first l layers contain a token). Due to C19, no tokens are added in the context (comparing the context for slot i to the one for slot $i+1$). Due to C13, only one single token is deleted in the context. Due to C11, this token must be deleted in the last used layer l (which is consistent). Finally, assume that slot i contains an opening bracket, together with one token t (this is the only remaining possibility due to C1, C5, C6). Assume that l context layers are used at slot i. Due to C13, only one single token is added in context $i+1$ (compared to context i). None of the currently selected tokens in the first l layers can be removed due to C18. Also, due to C10, no token can be added in any of the l used layers. Due to C11, a token can only be added in layer $l+1$. Variable a_{it}, indicating the addition of token t to the context at position i, must be set to one (due to C14, C15, and C16).

Due to C17, this implies that at least one context layer at position $i+1$ must contain this token. Due to C9, this token was not selected in any layer up to layer l. Instead, the only remaining possibility is that the token is added in layer $l+1$. □

Theorem 7.4. *An integer linear program solution describes the target schema accurately.*

Proof. A fact connects two identifier tokens t_1 and t_2. A fact is mentioned if the description contains a sub-expression of the form $t_1(\ldots t_2 \ldots)$ or $t_2(\ldots t_1 \ldots)$. Solutions represent syntactically valid schema descriptions (Theorem 7.1) and context is assigned consistently (Theorem 7.3). Hence, there must be a slot i such that $x_{it_1} = 1$ and $\exists l : c_{ilt_2} = 1$ or, vice versa, a slot with $x_{it_2} = 1$ and $\exists l : c_{ilt_1} = 1$. In that case, $m_{t_1t_2} = 1$ or $m_{t_2t_1} = 1$ due to C20, C21, and C22. Due to C23 and C24, having at least one mention of a fact is equivalent to $f_{t_1t_2} = 1$ (for $t_1 < t_2$). Also, due to C25, all true facts are stated while, due to C26, no incorrect facts are stated. □

In summary, the prior theorems show that each ILP solution represents a syntactically and semantically correct schema description.

7.2 Complexity

We analyze complexity of the schema compression problem.

Theorem 7.5. *Schema compression is NP-hard.*

Proof. The proof uses a polynomial-time reduction from uncapacitated facility location (UFL) [16]. An instance of UFL is defined by cost values f_i for opening a facility at location i, as well as cost factors c_{ij} such that the cost of servicing location j from facility location i is c_{ij}. The goal is to minimize the sum of both cost terms. We transform such an instance to an instance of schema compression as follows. First, introduce a single table with one unannotated column for each client location j. Using an expression of the form $TableName(Column_1, Column_2, \ldots)$ expresses all relevant facts (i.e., the association between the table and its columns) as concisely as possible. However, there are choices regarding the column representation. Introduce one representation function g_i for each eligible facility location i such that $size(Text(g_i)) = f_i$. Also, choose representation functions and column names such that $Size(g_i(Column_i)) = c_{ij}$. The solution to the schema compression problem introduces a subset of representation functions. Those functions correspond to the optimal facilities to open. □

The following theorems analyze the size of the ILP as a function of the dimensions of the schema compression problem. Often, the ILP size correlates with the time it takes to find optimal solutions. We denote by n_t the number of tokens, by n_i the number of slots, by n_l the number of context layers, and by n_g the number of representation functions considered.

Theorem 7.6. *The number of integer linear program variables is in* $O(n_i \cdot n_t \cdot (n_g + n_l + n_t))$.

Proof. Variable group r_{itg} has a number of variables in $O(n_i \cdot n_t \cdot n_g)$, dominating the number of decision variables (x_{it}), function selection variables (h_g), empty-slot variables (e_i), and context addition variables (a_{it}). The number of context variables c_{ilt} is in $O(n_i \cdot n_l \cdot n_t)$ while the number of variables representing fact mentions ($m_{it_1t_2}$) is in $O(n_i \cdot n_t^2)$ (thereby dominating the group of variables $f_{t_1t_2}$ representing fact statements). □

Theorem 7.7. *The number of integer linear program constraints is in* $O(n_i \cdot n_t \cdot (n_l + n_t + n_g))$.

Theorem 7.8. *Comparing constraint groups C1 to C19, groups C18 and C19 integrate a dominant number of constraints* ($O(n_i \cdot n_l \cdot n_t)$). *Comparing groups C20 to C26, groups C20 to C22 have a dominant number of constraints* ($O(n_i \cdot n_t^2)$). *Comparing C27 to C28, group C28 has a dominant number of constraints* ($O(n_i \cdot n_t \cdot n_g)$).

8 EXPERIMENTAL RESULTS

Section 8.1 describes the experimental setup. Section 8.2 compares Schemonic to different compression baselines. Section 8.3 validates that LLMs are able to understand compressed schema representations. Finally, Section 8.4 analyzes the impact of various tuning parameters on optimization performance.

8.1 Experimental Setup

Schemonic, as well as the baselines, are implemented in Python 3.10. Schemonic uses Gurobi 10 as ILP solver. All of the following experiments are executed on an EC2 c5.4xlarge instance with 32 GB of main memory and 16 virtual CPUs, running Ubuntu 22.04. We compare Schemonic to several baseline methods for database schema representation. First, we compare against the associated SQL DDL commands, as formatted by the "sqlglot" Python library. This baseline is referred to as "SQL" in the following plots. Second, we compare against the schema representation used in a prompt template for text-to-SQL translation, available for sale on a popular prompt distribution platform[2]. This baseline is referred to as "PB". Third, we compare to the output of the greedy algorithm discussed in Section 6.2 ("Greedy").

The experiments use schemata from three different benchmarks. First, we use schemata of the PublicBI benchmark [14]. This benchmark is derived from Tableau workbooks and represents real user data. Second, we evaluate baselines on the schema of the TPC-H benchmark. Third, we use the schemata of the SPIDER benchmark [38], a popular benchmark for text-to-SQL translation featuring 166 databases. In all cases, we measure the number of tokens according to the GPT tokenizer. Unless noted otherwise, we configure Schemonic to use all optimizations discussed in Section 6, up to three context layers, nine prefixes, and a timeout of 20 minutes per instance.

8.2 Comparing Compression Methods

Figure 4 compares different schema compression methods in terms of their size and compression overheads. The figure contains boxplots for each benchmark and baseline. The PublicBI and SPIDER benchmarks contain multiple database schemata and each data point is associated with one schema. The TPC-H benchmark only features a single database (containing eight tables). This is why the boxplots for TPC-H condense into a single line. For PublicBI and SPIDER, boxes cover the range between the 25th and 75th percentile, the line inside of each box represents the median, and diamond symbols mark the arithmetic average. As usual, lower

[2] https://promptbase.com/prompt/generate-sql-based-on-your-schema

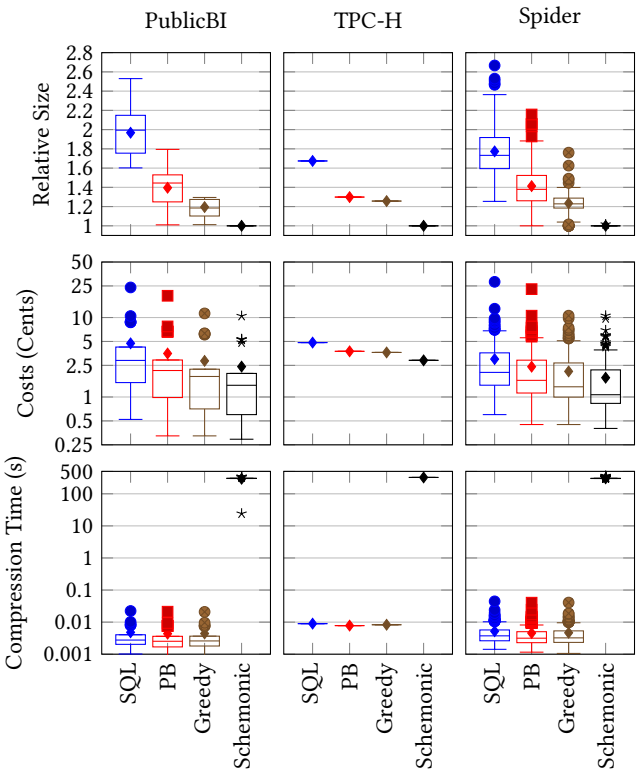

Figure 4: Comparing compression ratio, fees per invocation using GPT-4, and compression time of Schemonic to baselines (note the logarithmic y-axis for the lower two rows).

whiskers denote the smallest data values larger than the lower box bound minus 150% of the box height. Upper whiskers are defined analogously. Single marks represent outliers outside of that range.

The columns in Figure 4 are associated with the three benchmarks. The first row reports the size of the schema description, measured in tokens and scaled to the size of the smallest description for each benchmark. On average, Schemonic reduces the size of schema descriptions by a factor between 1.7 (TPC-H) and 2 (PublicBI). In some cases, Schemonic achieves a compression factor of close to three (on the SPIDER benchmark). Compared to the simpler greedy approach, Schemonic reduces the length of the schema description by at least 20% on average for each of the three benchmarks. E.g., for the SPIDER benchmark, Schemonic reduces description length by over 23% on average and up to 76% for some schemata. For TPC-H, Schemonic reduces the length of the schema description by 26%, compared to the greedy approach.

The second row reports the fees for reading the schema description via GPT-4. At the time of writing, reading 1,000 tokens with GPT-4 (gpt-4-32k) costs 6 cents of processing fees[3]. For instance, when using GPT-4 for text-to-SQL translation, the cost of reading the schema description has to be paid for each invocation of the model, i.e., for each new query. As processing fees are proportional to the schema size, the tendencies in the second row are similar to the ones shown in the first row of plots (note, however, that the y-axis of the second row is logarithmic whereas the y-axis of the first row is linear). The costs of reading the database schema once reach up to 28 cents for a traditional schema representation but are always below 11 cents for the descriptions generated by Schemonic (those maxima occur for the "baseball_1" database of the SPIDER benchmark, featuring 26 tables). Clearly, fees for processing schema descriptions via GPT can be comparable to or can even dominate the processing costs of typical SQL queries (e.g., costs for reading schema descriptions are comparable to the costs of querying hundreds of gigabytes of data on BigQuery[4]). This makes it worthwhile to optimize this component of total processing fees. While prices for models such as GPT-4 tend to decrease over time, new models appear regularly and come with higher costs.

The third row (note the logarithmic y-axis) reports compression time for the different baselines. All baselines except for Schemonic achieve compression times of less than 100 milliseconds. Those baselines exceed compression times of 10 milliseconds only for benchmarks containing large schemata with a reading cost of 10 cents and more. Schemonic consumes up to five minutes of compression time (i.e., it reaches the timeout). However, ILP solvers continuously generate solutions. This means that even if Schemonic reaches the timeout, it produces solutions that come with near-optimality guarantees (generated by the ILP solver). Investing time into schema compression pays off in scenarios such as text-to-SQL translation where schema descriptions are read frequently whereas schema changes are comparatively rare. In such scenarios, the additional time Schemonic spends in compression is quickly offset by savings when processing schema descriptions via language models.

8.3 Compression versus LLM Accuracy

We analyze whether compression impacts result quality for text-to-SQL translation [12]. TPC-H and the PublicBI benchmark only feature SQL queries, no associated natural language questions. Hence, we cannot evaluate the precision of text-to-SQL translation using those benchmarks. SPIDER, on the other hand, features SQL queries with corresponding natural language questions. We use the first 200 questions from the training set of the original SPIDER benchmark ("SPIDER" in the following plots), the 508 questions of SPIDER-Realistic [9] ("SPIDER-Real"), a benchmark variant referring to the same schemata as SPIDER, as well as the 1034 questions of SPIDER Synthetic [11] ("SPIDER-Syn"), another variant referring to the same schemata. We use the following prompt template for text-to-SQL translation:

```
Schema:[SD]
Question:[NLQ]
SQL:
```

`[SD]` is a placeholder, representing the (original or compressed) database schema description, while `[NLQ]` represents the question to translate into an SQL query. We instantiate the prompt template by substituting both placeholders and extracting the translated SQL query from the completion generated by the language model.

Figure 5 shows the number of successfully translated queries (as prior work on text-to-SQL translation, this number is calculated automatically by comparing the results of executing generated

[3]https://openai.com/pricing

[4]https://cloud.google.com/bigquery/pricing

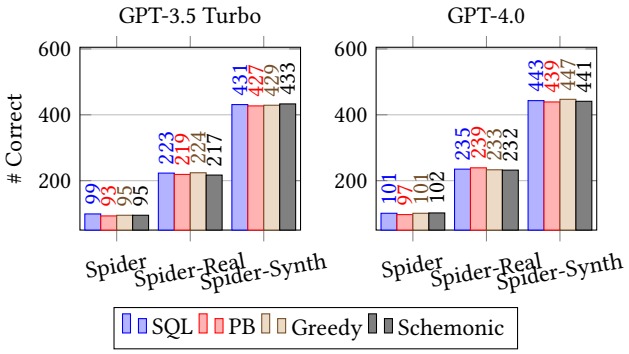

Figure 5: Number of correctly translated queries for different benchmarks, models, and schema descriptions.

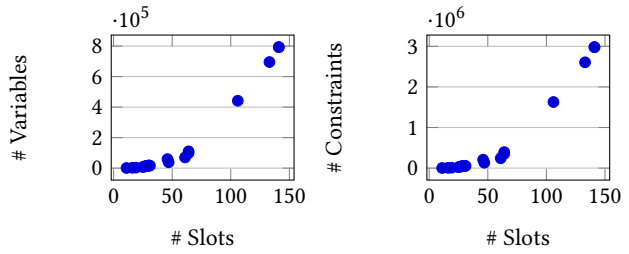

Figure 6: Size of integer linear program as a function of input problem size.

SQL queries to the ground truth result) for all three benchmark variants. The figure reports results for GPT-3.5 Turbo as well as for the (significantly larger) GPT-4 model, using the default variants of each model as of June 1, 2024. Results are shown for the schema descriptions generated by all baselines introduced before. In all scenarios, the success ratio is similar across all baselines. For both GPT models, Schemonic solves most test cases for one benchmark whereas the greedy algorithm solves most test cases in another one of the three benchmarks. Even for the smaller GPT-3.5 Turbo model, the results are inconsistent with a significant reduction in result quality due to compressed schema descriptions.

8.4 Further Analysis

Figure 6 illustrates dependencies between schema dimensions and the size of the associated ILP. Each point corresponds to one test case, reporting the schema length on the x-axis and the number of ILP variables or constraints on the y-axis. According to the formal analysis in Section 7, the number of variables and constraints grows linearly in the number of slots and quadratically in the number of distinct tokens. Assuming that column names are the token subset with dominant size, the number of tokens is highly correlated with schema length. The results in Figure 6 show superlinear growth and are therefore consistent with the predictions from Section 7.

Finally, Figure 7 reports on results of an ablation study, successively removing the optimizations discussed in Section 6. On the y-axis, the figure measures the percent of test cases for which a

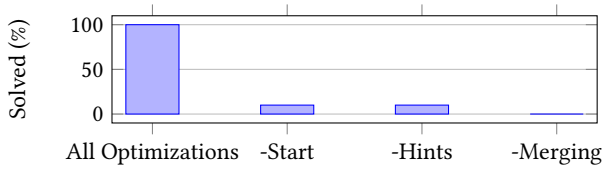

Figure 7: Impact on optimizations on ratio of test cases solved.

valid solution was found during an optimization time of five minutes. From left to right, the figure first removes the insertion of a greedy solution as start values, the addition of hints on variable values, and finally column merging. While 100% of test cases are solved with all optimizations, the number reduces to zero with all optimizations deactivated.

9 RELATED WORK

Prior work has shown that changes to the prompt can significantly influence the performance of language models in certain scenarios [28, 31, 39], motivating work aimed at maximizing output quality by automatically tuning prompts [8, 30, 34]. On the other hand, a recent line of work on prompt compression shows that eliminating redundant information in the prompt has negligible impact on result quality in many scenarios[1, 6, 15, 20, 21, 25]. Schemonic belongs into this category. It differs from prior approaches as the method is specialized to database schema compression, enabling Schemonic to guarantee that all generated descriptions are semantically correct. E.g., using small language models for prompt compression [20] is a more generic approach but lacks formal guarantees that compression preserves all relevant information. Prompt compression is complementary to other approaches, aimed at reducing cost for large language models, such as compressing models themselves [37] or batching multiple tasks for the same context [24].

The proposed approach connects to prior work on relational data compression [3, 13, 17–19, 29] as well as workload compression [5, 22, 35]. However, it differs by its target (schema compression) and its context (prompt compression to reduce LLM cost). Broadly, this work connects to prior work using ILP to solve database optimization problems [2, 10, 27]. However, the problem solved in this work differs from the problems addressed in prior work. Finally, Schemonic connects to various applications in the database area that require describing database schemata to LLMs. Among others, such applications include text-to-SQL translation [23, 32, 38, 40], an area where prompting is popular [12], data wrangling tasks [26, 36], or information extraction [7].

10 CONCLUSION

Schemonic optimizes prompts that contain a description of a relational database, minimizing the number of tokens used on an LLM (and therefore invocation overheads). The experiments show that this approach reduces costs significantly while compressed representations can be well understood by LLMs.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

REFERENCES

[1] Md Adnan Arefeen, Biplob Debnath, and Srimat Chakradhar. 2023. LeanContext: Cost-Efficient Domain-Specific Question Answering Using LLMs. *CoRR* abs/2309.0 (2023), 1–8. arXiv:2309.00841 http://arxiv.org/abs/2309.00841

[2] Zohreh Asgharzadeh Talebi, Rada Chirkova, and Yahya Fathi. 2013. An integer programming approach for the view and index selection problem. *DKE* 83 (2013), 111–125. https://doi.org/10.1016/j.datak.2012.11.001

[3] Shivnath Babu, Minos Garofalakis, and Rajeev Rastogi. 2001. SPARTAN: A Model-Based Semantic Compression System for Massive Data Tables. *SIGMOD Record* 30, 2 (2001), 283–294. https://doi.org/10.1145/376284.375693

[4] Philip a Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic Schema Matching, Ten Years Later. *VLDB* 4, 11 (2011), 695–701. https://doi.org/10.1007/s007780100057

[5] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. 2002. Compressing SQL workloads. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD 2002* (2002), 488–499. https://doi.org/10.1145/564691.564747

[6] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. (2023). arXiv:2305.05176 http://arxiv.org/abs/2305.05176

[7] Jim Cowie and Wendy Lehnert. 1996. Information Extraction. *Commun. ACM* 39, 1 (1996), 80–91.

[8] Mingkai Deng, Jianyu Wang, Cheng Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P. Xing, and Zhiting Hu. 2022. RLPROMPT: Optimizing Discrete Text Prompts with Reinforcement Learning. In *EMNLP*. 3369–3391. https://doi.org/10.18653/v1/2022.emnlp-main.222 arXiv:2205.12548

[9] Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-Grounded Pretraining for Text-to-SQL. In *NAACL-HLT*. 1337–1350. https://doi.org/10.18653/v1/2021.naacl-main.105 arXiv:2010.12773

[10] Tansel Dokeroglu, Murat Ali Bayir, and Ahmet Cosar. 2014. Integer linear programming solution for the multiple query optimization problem. In *Information Sciences and Systems*. 51–60. https://doi.org/10.1007/978-3-319-09465-6

[11] Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021. Towards robustness of text-to-SQL models against synonym substitution. In *ACL-IJCNLP*. 2505–2515. https://doi.org/10.18653/v1/2021.acl-long.195 arXiv:2106.01065

[12] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *PVLDB* 17, 5 (2024), 1132–1145. https://doi.org/10.14778/3641204.3641221 arXiv:2308.15363

[13] Yihan Gao and Aditya Parameswaran. 2016. SQUISH: Near-optimal compression for archival of relational datasets. In *SIGKDD*, Vol. 13-17-Augu. 1575–1584. https://doi.org/10.1145/2939672.2939867 arXiv:1602.04256

[14] Bogdan Ghit, Cwi Amsterdam, NL Diego Tomé, and Peter Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR*. 1–7.

[15] Henry Gilbert, Michael Sandborn, Douglas C. Schmidt, Jesse Spencer-Smith, and Jules White. 2023. Semantic Compression With Large Language Models. (2023). arXiv:2304.12512 http://arxiv.org/abs/2304.12512

[16] Sudipto Guha. 1999. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms* 31 (1999), 228–248.

[17] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *SIGMOD*. 1733–1746. https://doi.org/10.1145/3318464.3389734

[18] Balakrishna R Iyer and David Wilhite. 1994. Data Compression Support in Databases. In *VLDB*. 695–704.

[19] H. V. Jagadish, Raymond T. Ng, Beng Chin Ooi, and Anthony K.H. Tung. 2004. ItCompress: An iterative semantic compression algorithm. In *ICDE*, Vol. 20. 646–657. https://doi.org/10.1109/ICDE.2004.1320034

[20] Huiqiang Jiang, Qianhui Wu, Chin Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. In *EMNLP*. 13358–13376. https://doi.org/10.18653/v1/2023.emnlp-main.825 arXiv:2310.05736

[21] Hoyoun Jung and Kyung-Joong Kim. 2023. Discrete Prompt Compression with Reinforcement Learning. *CoRR* abs/2308.0 (2023), 1–12. arXiv:2308.08758 http://arxiv.org/abs/2308.08758

[22] Piotr Kołaczkowski. 2008. Compressing Very Large Database Workloads for Continuous Online Index Selection. *Lecture Notes in Computer Science* 5181 LNCS, 3 (2008), 791–799. https://doi.org/10.1007/978-3-540-85654-2_71

[23] Fei Li and HV Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *SIGMOD*. 709–712.

[24] Jianzhe Lin, Maurice Diesendruck, Liang Du, and Robin Abraham. 2023. BatchPrompt: Accomplish more with less. *CoRR* abs/2309.0 (2023), 1–20. arXiv:2309.00384 http://arxiv.org/abs/2309.00384

[25] Jesse Mu, Xiang Lisa Li, and Noah Goodman. 2023. Learning to Compress Prompts with Gist Tokens. *CoRR* abs/2304.0 (2023), 1–26. arXiv:2304.08467 http://arxiv.org/abs/2304.08467

[26] Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *PVLDB* 16, 4 (2022), 738–746. arXiv:2205.09911 http://arxiv.org/abs/2205.09911

[27] Stratos Papadomanolakis and Anastassia Ailamaki. 2007. An integer linear programming approach to database design. In *ICDEW*. 442–449. https://doi.org/10.1109/ICDEW.2007.4401027

[28] Pouya Pezeshkpour and Estevam Hruschka. 2023. Large Language Models Sensitivity to The Order of Options in Multiple-Choice Questions. (2023). arXiv:2308.11483 http://arxiv.org/abs/2308.11483

[29] Meikel Poess and Dmitry Potapov. 2003. Data Compression in Oracle. In *VLDB*. 937–947. https://doi.org/10.1016/B978-012722442-8/50087-2

[30] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. 2023. GRIPS: Gradient-free, Edit-based Instruction Search for Prompting Large Language Models. In *EACL*. 3827–3846. https://doi.org/10.18653/v1/2023.eacl-main.277 arXiv:2203.07281

[31] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2023. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I learned to start worrying about prompt formatting. *CoRR* (2023). arXiv:2310.11324 http://arxiv.org/abs/2310.11324

[32] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. 2020. ATHENA++: natural language querying for complex nested SQL queries. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2747–2759. https://doi.org/10.14778/3407790.3407858

[33] Richard Shin and Benjamin Van Durme. 2021. Evaluating the Text-to-SQL Capabilities of Large Language Models. *CoRR* abs/2204.0, 1 (2021), 1–12. https://arxiv.org/abs/2204.00498

[34] Taylor Shin, Yasaman Razeghi, Robert L. Logan, Eric Wallace, and Sameer Singh. 2020. AUTOPROMPT: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *EMNLP*. 4222–4235. https://doi.org/10.18653/v1/2020.emnlp-main.346 arXiv:2010.15980

[35] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2022), 660–673. https://doi.org/10.1145/3514221.3526152

[36] Nan Tang, Ju Fan, Fangyi Li, Jianhong Tu, Xiaoyong Du, Guoliang Li, Sam Madden, and Mourad Ouzzani. 2021. Rpt: Relational pre-trained transformer is almost all you need towards democratizing data preparation. *PVLDB* 14, 8 (2021), 1254–1261. https://doi.org/10.14778/3457390.3457391 arXiv:2012.02469

[37] Zhaozhuo Xu, Zirui Liu, Beidi Chen, Yuxin Tang, Jue Wang, Kaixiong Zhou, Xia Hu, and Anshumali Shrivastava. 2023. Compress, Then Prompt: Improving Accuracy-Efficiency Trade-off of LLM Inference with Transferable Prompt. (2023), 1–21. arXiv:2305.11186 http://arxiv.org/abs/2305.11186

[38] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2020. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018*. 3911–3921. https://doi.org/10.18653/v1/d18-1425 arXiv:1809.08887

[39] Chujie Zheng, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. 2024. Large Language Models are not Robust Multiple Choice Selectors. In *ICLR*. 1–14. arXiv:2209.15093 http://arxiv.org/abs/2209.15093

[40] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.0, 1 (2017), 1–12. arXiv:1709.00103 http://arxiv.org/abs/1709.00103