2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Synthesizing Document Database Queries using Collection Abstractions

Qikang Liu, Yang He, Yanwen Cai, Byeongguk Kwak, Yuepeng Wang
Simon Fraser University, Burnaby, BC, Canada
{qla116, yha244, yca452, bka47, yuepeng}@sfu.ca

***Abstract*—Document databases are increasingly popular in various applications, but their queries are challenging to write due to the flexible and complex data model underlying document databases. This paper presents a synthesis technique that aims to generate document database queries from input-output examples automatically. A new domain-specific language is designed to express a representative set of document database queries in an algebraic style. Furthermore, the synthesis technique leverages a novel abstraction of collections for deduction to efficiently prune the search space and quickly generate the target query. An evaluation of 110 benchmarks from various sources shows that the proposed technique can synthesize 108 benchmarks successfully. On average, the synthesizer can generate document database queries from a small number of input-output examples within tens of seconds.**

I. Introduction

Document databases like MongoDB [30] and CouchDB [10] have become increasingly popular in various real-world scenarios, such as online commercial platforms, financial services, gaming, and social media applications [31]. Different from traditional relational databases that primarily use structured data like tables, document databases persist data in a semi-structured format such as JSON and BSON. While the semi-structured data format provides developers with great flexibility in storing and querying complex data structures directly, it also raises significant challenges for users to write queries for document databases.

To help users write document database queries in an easy and convenient fashion, we develop a synthesis technique to generate queries automatically. Inspired by prior work on automated synthesis of SQL queries for relational databases [14], [47], [54], our technique aims to generate document database queries from input-output examples. Specifically, the user only needs to provide a small number of examples to demonstrate the query, where the input example is a small document database consisting of a few documents, and the output example is the desired query result over the input. The goal of our synthesis technique is to generate a document database query such that executing the query over the input example produces the output example.

However, unlike synthesizing SQL queries, there are several key challenges to synthesizing queries for document databases.

- *Hierarchical and nested data structures*. Document databases support hierarchical and nested data structures, such as arrays, documents, and their combinations. Since queries for document databases constantly operate over these complex data structures, it is crucial for synthesizers to reason about complex data structures efficiently for better performance.
- *Specialized query language*. Query languages for document databases may use specialized operators over complex data structures that relational databases cannot handle. For instance, MongoDB uses a lookup operator in aggregation pipelines to query data over multiple collections. Synthesizers need to support an expressive query language for document databases while maintaining the efficiency of exploring a large search space of the target query.

To address these challenges, we have designed *a new domain-specific language* based on the aggregation pipeline in MongoDB that can express a representative set of queries with core operators of document databases. The queries of this language are in an algebraic style similar to relational algebra but tailored towards document databases.

Furthermore, prior work on program synthesis proposed an approach to speed up the synthesis process by deduction [13], [14]. For fast synthesis of document database queries, we have adapted this approach to our setting and developed *a novel abstraction for collections containing hierarchical and nested data structures* to prune the search space efficiently. The key insight is that the "shape" and size of collections can help the synthesizer quickly prune incorrect queries, even if the query is partial. Thus, our abstraction consists of two pieces of information about the collection: First, it includes the *type* of documents inside the collection. Second, it includes a logical formula describing constraints over the *size* of the collection.

More specifically, our synthesis technique is presented schematically in Figure 1. At a high level, the synthesis technique takes an iterative approach and has two phases in each iteration. In the first phase, the synthesizer aims to find a query sketch, which is a partial query with some unknown constructs. In the second phase, it tries to complete the sketch into a full query that can satisfy all provided input-output examples. In general, it is not efficient to check if a sketch is feasible to be completed into a correct full query by checking all possible completions against the examples, because a sketch may have a large number of completions. To avoid such inefficiency, the key part of our synthesis technique is a deduction engine, which can check if a sketch is feasible to get a correct query without checking its completions. In particular, the deduction engine can directly evaluate the sketch over *abstractions of collections* and obtain abstract collections. If the expected output example is a valid concretization of one of

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00152

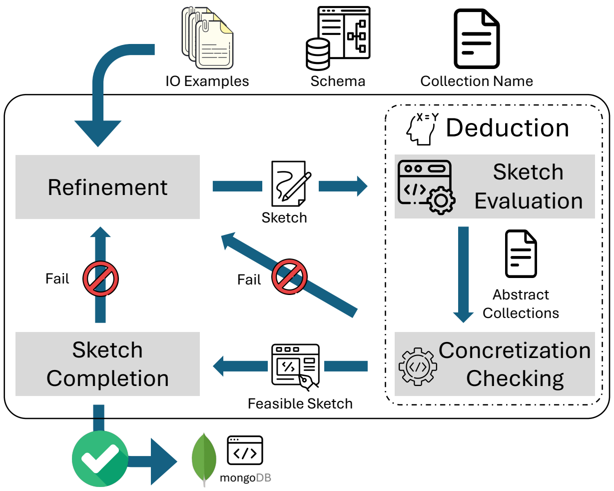

Fig. 1: Schematic workflow.

the resulting abstract collections, the synthesizer concludes the sketch is feasible to complete and proceeds to find a correct completion. Otherwise, the synthesizer can safely conclude the sketch is infeasible to complete, prune the search space accordingly, and propose a different sketch to the next iteration by refining the infeasible sketch.

Based on this technique, we have developed a tool called NOSDAQ that can synthesize document database queries from input-output examples. To evaluate the synthesis technique, we have collected 110 benchmarks from various application scenarios, including StackOverflow, Kaggle, MongoDB official documents, and Twitter API documents. The evaluation result shows that NOSDAQ can successfully synthesize 108 document database queries within the 5-minute time limit. Furthermore, NOSDAQ only uses 1 – 3 input-output examples and finishes query synthesis in an average of 14.2 seconds, which demonstrates the effectiveness and efficiency of our synthesis technique.

Contributions. To summarize, the main contributions of this paper are as follows.

1) We develop a technique for synthesizing document database queries from input-output examples.
2) We design a new domain-specific language to express document database queries in algebraic style.
3) We design a novel abstraction for collections containing hierarchical and nested data structures and use this abstraction to speedup the synthesis of document database queries based on deduction.
4) We define the abstract semantics of document database queries based on our abstraction of collections.
5) We develop a tool called NOSDAQ and evaluate it over 110 benchmarks from various sources. The evaluation result shows that NOSDAQ is effective and efficient in synthesizing document database queries.

Organization. The remainder of this paper is structured as follows. Section II provides a motivating example to illustrate our technique. Section III formalizes the synthesis problem, and Section IV introduces collection abstractions. Sections V and VI present the synthesis algorithm and its implementation details, respectively. Section VII presents the experimental setup and evaluation results. Section VIII discusses the related work, followed by a conclusion in Section IX.

```
{posts: [{
    _id: "1", title: "Title-1",
    replies: [{depth: 0}, {depth: 0}, {depth: 1}]
 }, {
    _id: "2", title: "Title-2",
    replies: [{depth: 0}, {depth: 1}, {depth: 2}]
 }, {
    _id: "3", title: "Title-3",
    replies: [{depth: 0}, {depth: 1}, {depth: 2},
        {depth: 3}]
 }]}
```

Fig. 2: Input example.

II. MOTIVATING EXAMPLE

To explain our synthesis technique, let us consider a concrete motivating example. Given a document database collected from the Kaggle website that stores a list of Reddit posts. The database only has one collection called `posts` with the following schema [1]

$$Arr\langle\{\texttt{\_id} : String, \texttt{title} : String, \texttt{replies} : Arr\langle\{\texttt{depth} : Num\}\rangle\}\rangle$$

where $Arr\langle\tau\rangle$ denotes the array type of τ. Specifically, the `posts` collection contains an array of documents, where each document has three attributes: `_id`, `title`, and `replies`. The `replies` attribute is also an array of documents and the document has one attribute `depth` denoting the nesting level of the reply from the root post.

Now suppose the user wants to query the title of posts which have more than one non-zero-depth replies and the count of these replies. NOSDAQ can help the user synthesize this query automatically. The user needs to provide small input-output examples to demonstrate their intention. For instance, Figure 2 is an input example, and the corresponding output example is

```
[{reply_count: 3, title: "Title-3"},
 {reply_count: 2, title: "Title-2"}]
```

The goal of NOSDAQ is to synthesize a query such that executing the query on the input example produces the output example. NOSDAQ takes an iterative approach to solve the synthesis problem. In each iteration, it first proposes a query sketch that may contain unknowns and then checks if the sketch is feasible to complete. If feasible, NOSDAQ completes the sketch into a full query by enumerative search and checks if any query satisfies the input-output example. If the sketch is infeasible to complete, then NOSDAQ refines the sketch and starts the next iteration.

First iteration. NOSDAQ starts with the simplest sketch in its domain-specific language – the `posts` collection and checks its feasibility. To do so, the deduction engine of NOSDAQ uses the collection abstractions and evaluates the sketch based on its

[1] The database schema is simplified in this section for illustration.

abstract semantics. Specifically, the abstraction for the `posts` collection is $\tilde{\mathcal{C}} = (\mathcal{T}, \phi)$ where

$$\mathcal{T} : \{\texttt{\_id} : \textit{String}, \texttt{title} : \textit{String}, \texttt{replies} : \textit{Arr}\langle\{\texttt{depth} : \textit{Num}\}\rangle\}$$

is the type of inside documents and $\phi : l_0 = 3$ is the formula describing the size of the collection is 3. NOSDAQ takes the abstract collection as input and evaluates the sketch `posts` based on the abstract semantics. The evaluation result is $\{\tilde{\mathcal{C}}_1\}$ where $\tilde{\mathcal{C}}_1 = (\mathcal{T}, \phi)$, which is exactly the same as $\tilde{\mathcal{C}}$. An important observation here is that the output example is not a concretization of $\tilde{\mathcal{C}}_1$ because its document has type $\{\texttt{title} : \textit{String}, \texttt{reply\_count} : \textit{Num}\}$ and its size is 2. Thus, NOSDAQ concludes the sketch `posts` is not feasible to complete to a correct query and starts to refine the sketch for the next iteration. In particular, NOSDAQ generates candidate sketches based on the grammar of its query language, such as $\textit{Project}(\texttt{posts}, \vec{h})$ and $\textit{Match}(\texttt{posts}, \phi)$.

Deduction with collection abstractions. Several iterations later, NOSDAQ encounters the following sketch Ω_2

$$\textit{Project}(\textit{Match}(\textit{Unwind}(\texttt{posts}, h_1), \phi), \vec{h_2})$$

This time, the evaluation result is $\{\tilde{\mathcal{C}}_2\}$ where $\tilde{\mathcal{C}}_2 = (\mathcal{T}_2, \phi_2)$ where $\mathcal{T}_2$ is $\{\texttt{title} : \textit{String}\}$ and ϕ_2 is $l_0 = 3 \wedge l_1 \geq l_0 \wedge l_2 \leq l_1 \wedge l_3 = l_2$, where l_3 corresponds to the size of $\tilde{\mathcal{C}}_2$. The sketch Ω_2 is still infeasible to complete, because the output document has an additional attribute `reply_count` that does not match the type $\mathcal{T}_2$. NOSDAQ prunes this sketch Ω_2 and continues to search for a feasible sketch.

Feasible sketch. After a few more iterations, NOSDAQ finds another sketch Ω_3

$$\textit{Project}(\textit{Match}(\textit{AddFields}(\textit{Group}(\textit{Match}(\textit{Unwind}(\texttt{posts}, h_1), \phi), \vec{h_2}, \vec{a}, \vec{A}), \vec{h_3}, \vec{E}), \phi'), \vec{h_4})$$

The evaluation result of this sketch over the abstract semantics is a set of abstract collections Λ, meaning the result can be some one inside Λ. Among this set, there is an abstract collection $\tilde{\mathcal{C}}_3 = (\mathcal{T}_3, \phi_3) \in \Lambda$ where

$$\mathcal{T}_3 : \{?_0^+ : \textit{Any}, ?_3^+ : \textit{Num}\}$$
$$\phi_3 : l_0 = 3 \wedge l_1 \geq l_0 \wedge l_2 \leq l_1 \wedge l_3 < l_2 \wedge l_4 = l_3 \wedge l_5 \leq l_4 \wedge l_6 = l_5$$

Here, $?_0^+$ and $?_3^+$ denote placeholders that can match one or more attributes. *Any* denotes any value type. l_6 is the variable that corresponds to the size of $\tilde{\mathcal{C}}_3$. Observe that the output example is a concretization of abstract collection $\tilde{\mathcal{C}}_3$, because the `title` matches $?_0^+$ and `reply_count` matches $?_3^+$. In addition, the size of the output collection is consistent with the size of $\tilde{\mathcal{C}}_3$, because $l_6 = 2 \wedge \phi_3$ is satisfiable. Therefore, NOSDAQ finds a feasible sketch Ω_3.

Sketch completion. Given a feasible sketch Ω_3, NOSDAQ aims to complete Ω_3 by finding instantiations of all unknown operators in the sketch, such as $h_1, \vec{h_2}, \vec{a}$, etc. Towards this goal, NOSDAQ performs enumerative search and finds the following query finally

$$\textit{Project}(\textit{Match}(\textit{AddFields}(\textit{Group}(\textit{Match}(\textit{Unwind}(\texttt{posts}, \texttt{replies}), \texttt{replies.depth} > 0), [\texttt{\_id}, \texttt{title}], [\texttt{reply\_count}], [\textit{Count}()]), [\texttt{title}], [\texttt{\_id.title}]), \texttt{reply\_count} > 1), [\texttt{reply\_count}, \texttt{title}])$$

$$
\begin{array}{rcl}
\text{Schema } \mathcal{S} & ::= & \{N_1 \mapsto T_{C_1}, \ldots, N_m \mapsto T_{C_m}\} \\
\text{Collection Type } T_C & ::= & \textit{Arr}\langle T_D \rangle \\
\text{Document Type } T_D & ::= & \{a_1 : T_{V_1} \ldots a_n : T_{V_n}\} \\
\text{Value Type } T_V & ::= & T_D \mid \textit{Arr}\langle T_V \rangle \mid T_P \\
\text{Primitive Type } T_P & ::= & \textit{Num} \mid \textit{String} \mid \textit{Bool} \\
& \mid & \textit{Datetime} \mid \textit{ObjectId}
\end{array}
$$

$N \in$ **Collection Names** $\quad a \in$ **Attributes**

Fig. 3: Schema of document databases.

$$
\begin{array}{rcl}
\text{Database } \mathcal{D} & ::= & \{N_1 \mapsto \mathcal{C}_1, \ldots, N_m \mapsto \mathcal{C}_m\} \\
\text{Collection } \mathcal{C} & ::= & [D] \\
\text{Document } D & ::= & \{a_1 : v_1, \ldots, a_n : v_n\} \\
\text{Value } v & ::= & D \mid [v_1, \ldots, v_n] \mid c
\end{array}
$$

$N \in$ **Collection Names** $\quad a \in$ **Attributes** $\quad c \in$ **Constants**

Fig. 4: Definition of document databases.

Executing this query on the input example produces exactly the output example, so the synthesis process is finished. The query corresponds to the following MongoDB query

```
db.posts.aggregate([
  {$unwind: "$replies"},
  {$match: {"replies.depth": {$gt: 0}}},
  {$group:
    {_id: { _id: "$_id", title: "$title" },
     reply_count: { $count: {}}}},
  {$addFields: {title: "$_id.title"}},
  {$match: {reply_count: {$gt: 1}}},
  {$project: {_id: 0, reply_count: 1, title: 1}}])
```

III. PROBLEM FORMULATION

In this section, we present formulations that are necessary for the rest of the paper and formally define our problem.

A. Document Schema and Database

We first precisely define the document schema and document database considered in this paper.

Document schema. As shown in Figure 3, a document schema $\mathcal{S}$ is a map from collection names to collection types, where a collection type is an array of document types. A document type is a map from attributes to different value types, including document types, arrays, and primitive types such as *Num*, *String*, and *Bool*.

Document database. As shown in Figure 4, a document database is a map from collection names to their corresponding collections. A collection is an array of documents. A document is a map from attributes to values, where the value is a document, an array of values, or a constant of primitive types.

Typing and conformance. Figure 5 presents a set of typing rules for conformance checking between document databases and schemas, where judgments of the form $\vdash \mathcal{D} : \mathcal{S}$ mean the database $\mathcal{D}$ conforms to schema $\mathcal{S}$. [2] Specifically, according to the T-Primitive rule, the type of a constant v is simply $\textit{Type}(v)$.

[2] We view *Null* as a special value of any primitive type. If an attribute has both null values and non-null values in some collection, then its type will be the same as that of the non-null value.

$$\frac{v \in \textit{Constants} \quad \textit{Type}(v) = \tau}{\vdash v : \tau} \text{ (T-Primitive)}$$

$$\frac{\vdash v_i : \tau \quad i = 1, \ldots, n}{\vdash [v_1, \ldots, v_n] : \textit{Arr}\langle\tau\rangle} \text{ (T-Array)}$$

$$\frac{\begin{array}{c} D = \{a_1 : v_1, \ldots, a_n : v_n\} \\ \vdash v_i : \tau_i \quad i = 1, \ldots, n \end{array}}{\vdash D : \{a_1 : \tau_1, \ldots, a_n : \tau_n\}} \text{ (T-Doc)}$$

$$\frac{\begin{array}{c} \mathcal{D} = \{N_1 \mapsto \mathcal{C}_1, \ldots, N_m \mapsto \mathcal{C}_m\} \\ \vdash \mathcal{C}_i : \tau_i \quad i = 1, \ldots, m \end{array}}{\vdash \mathcal{D} : \{N_1 \mapsto \tau_1, \ldots, N_m \mapsto \tau_m\}} \text{ (T-DB)}$$

Fig. 5: Rules for conformance between databases and schemas.

$$\begin{array}{rcl}
\text{Query } \mathcal{Q} & ::= & N \mid \textit{Project}(\mathcal{Q}, \vec{h}) \mid \textit{Match}(\mathcal{Q}, \phi) \\
& \mid & \textit{AddFields}(\mathcal{Q}, \vec{h}, \vec{E}) \mid \textit{Unwind}(\mathcal{Q}, h) \\
& \mid & \textit{Group}(\mathcal{Q}, \vec{h}, \vec{a}, \vec{A}) \mid \textit{Lookup}(\mathcal{Q}, h, h, N, a) \\
\text{Pred } \phi & ::= & \top \mid \bot \mid h \odot c \mid \textit{SizeEq}(h, c) \mid \textit{Exists}(h) \\
& \mid & \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\text{Expr } E & ::= & h \mid h \oplus h \mid f(h) \\
\text{Agg } A & ::= & \textit{Sum}(h) \mid \textit{Avg}(h) \mid \textit{Min}(h) \mid \textit{Max}(h) \mid \textit{Count}() \\
\text{LogicOp } \odot & ::= & \leq \mid < \mid = \mid \neq \mid > \mid \geq \\
\text{ArithOp } \oplus & ::= & + \mid - \mid \times \mid / \mid \%
\end{array}$$

$N \in$ **Collection Names** $\quad f \in$ **Math Functions**
$c \in$ **Constants** $\quad a \in$ **Attributes** $\quad h \in$ **Access Paths**

Fig. 6: Syntax of MongoDB Query. The two array parameters of AddFields must have the same length. The last two parameters of Group also must have the same length.

The T-Array rule describes that all elements v_i in an array must have the same type. If the element type is τ, then the array is of type $\textit{Arr}\langle\tau\rangle$. The T-Doc rule states that the type of a document $D = \{a_1 : v_1, \ldots, a_n : v_n\}$ is $\{a_1 : \tau_1, \ldots, a_n : \tau_n\}$ where τ_i is the type of v_i. Finally, based on the T-DB rule, the schema (or the type) of a database is basically a map from collection names to types of their corresponding collections.

B. Query Language

Next, we describe the syntax and semantics [3] of our query language for document databases. The query language has a straightforward correspondence to a core query language of the MongoDB aggregation pipelines.

The syntax of the query language is shown in Figure 6. At a high level, a query is a sequence of operations including *Project*, *Match*, *AddFields*, *Unwind*, *Group*, and *Lookup*, where different operators take different arguments such as a predicate ϕ or an expression E. Each operator corresponds to a stage of the MongoDB aggregation pipeline. More specifically, the name N simply retrieves collection N from the database. $\textit{Project}(\mathcal{Q}, \vec{h})$ projects fields with access paths $\vec{h}$ from each document in the collection of $\mathcal{Q}$. $\textit{Match}(\mathcal{Q}, \phi)$ filters the documents in $\mathcal{Q}$'s collection, retaining only those satisfy the predicate ϕ. $\textit{AddFields}(\mathcal{Q}, \vec{h}, \vec{E})$ introduces new fields $\vec{h}$ with associated values of $\vec{E}$ to each document in $\mathcal{Q}$. $\textit{Unwind}(\mathcal{Q}, h)$ deconstructs an array field h in the documents of $\mathcal{Q}$, mapping each document to a series of documents where the value of h is replaced by individual elements of the original array. $\textit{Group}(\mathcal{Q}, \vec{h}, \vec{a}, \vec{A})$ groups documents of $\mathcal{Q}$ based on grouping keys h, transforming each group into a single document with new attributes $\vec{a}$ and aggregated values $\vec{A}$. Finally, $\textit{Lookup}(\mathcal{Q}, h_1, h_2, N, a)$ adds a new attribute a to each document of $\mathcal{Q}$, where the attribute's value is a list of documents from a foreign collection N. This list only includes documents whose specified field h_2 in the foreign collection is the same as field h_1 in the original collection.

The predicate ϕ can be true $\top$, false $\bot$, logical comparison $h \odot c$, size equality $\textit{SizeEq}(h, c)$, existence of an access path $\textit{Exists}(h)$, and boolean connectives. The expression E can be an access path h, arithmetics $h \oplus h$, and mathematical functions $f(h)$. The access path is a sequence of attributes separated by dots such as $a_1.a_2.a_3$, denoting the path to access the data from the root document.

Example 1. *Let us consider a document* {`_id: 1, name: "John", class: "SE", info:` {`score: 90`}}. *The access path for the* `score` *attribute in* `info` *is* `info.score`.

Example 2. *Consider a MongoDB query*

```
db.coll.aggregate([{$group:
    {_id: {name: "$name", class: "$class"},
     total: {$sum: "$info.score"}}}])
```

It can be represented by the following query in our language

$$\textit{Group}(\texttt{coll}, [\texttt{name}, \texttt{class}], [\texttt{total}], [\textit{Sum}(\texttt{info.score})])$$

Example 3. *Consider a collection*

$$N = [\{a : 1, b : [2, 3]\}, \{a : 4, b : [5, 6]\}]$$

The evaluation result of $\textit{Unwind}(N, b)$ *is*

$$[\{a : 1, b : 2\}, \{a : 1, b : 3\}, \{a : 4, b : 5\}, \{a : 4, b : 6\}]$$

C. Problem Statement

Before we state the problem to solve in this paper, let us first define input-output examples.

Definition 1 (Input-output example)**.** *An example $\mathcal{E}$ over schema $\mathcal{S}$ is a pair (I, O) where I is the document database over schema $\mathcal{S}$ (i.e., $\vdash I : \mathcal{S}$) and O is the output collection.*

Synthesis problem. Given a database schema $\mathcal{S}$, a collection name $N \in \textit{dom}(\mathcal{S})$, and input-output examples $\vec{\mathcal{E}}$ over $\mathcal{S}$, the goal of our synthesis problem is to find a query $\mathcal{Q}$ over collection N in the language shown in Figure 6 such that for each example $(I, O) \in \vec{\mathcal{E}}$, it holds that $[\![\mathcal{Q}]\!]_I = O$. Here, $[\![\mathcal{Q}]\!]_I$ represents the evaluation result of $\mathcal{Q}$ given input database I.

IV. Abstraction for Collections

In this section, we will introduce the abstraction for collections in document databases and how to compute abstractions for queries and sketches.

Intuitively, since collections in document databases contain an array of documents, the abstraction for collections should contain two pieces of information: (1) the *type* of documents

[3]The formal semantics is described in the technical report [25].

inside the collection and (2) the *size* of the collection. Based on this idea, we can define abstract collections and databases.

Definition 2 (Abstract collection)**.** *An abstract collection* $\tilde{\mathcal{C}} = (\tau, \phi)$ *is a pair that consists of the type* τ *of inside documents and the formula* ϕ *about the collection size.*

Definition 3 (Abstract database)**.** *An abstract database* $\tilde{\mathcal{D}} = \{N_1 \mapsto \tilde{\mathcal{C}}_1, \ldots, N_m \mapsto \tilde{\mathcal{C}}_m\}$ *is a map from collection names to abstract collections.*

Since the synthesis process also involves partial programs that may yield unknown attributes, values, or types in the documents, we now augment documents with a notion of placeholders.

Definition 4 (Placeholder)**.** *A placeholder* $?^m$ *denotes a top-level attribute that can match any concrete attribute and* $m \in \{1, +\}$ *denotes how many attributes it can match.* $?^1$ *means the placeholder matches exactly one attribute and* $?^+$ *means it can match one or more attributes.*

Accordingly, we update the type of documents with placeholders and augment attributes with a special type called *Any* that represents any possible value type.

Definition 5 (Augmented type)**.** *An augmented type* $\mathcal{T}$ *is an extension of the document type* T_D *in Figure 3 where the attribute can be a named attribute or a placeholder and its type can be a value type* T_V *or* Any *denoting any value type.*

Example 4. *Let us consider an augmented type*

$$\{a : \mathit{String},\ ?^+_1 : \mathit{Any},\ ?^+_2 : \mathit{Num},\ ?^1_3 : \mathit{Arr}\langle\{c : \mathit{Num}, d : \mathit{String}\}\rangle\}$$

Here, $?^+_1$ *is a placeholder that matches one or more attributes of any type.* $?^+_2$ *is a placeholder that matches one or more attributes of* Num *type.* $?^1_3$ *is a placeholder that matches exactly one attribute corresponding to a collection where the document is of type* $\{c : \mathit{Num}, d : \mathit{String}\}$*.*

Next, we can lift the notion of abstract collections to cases where placeholders are involved in the documents.

Definition 6 (Abstract collection with placeholders)**.** *An abstract collection* $\tilde{\mathcal{C}} = (\mathcal{T}, \phi)$ *is a pair consisting of (1) the augmented type* $\mathcal{T}$ *of inside documents with potential placeholders and (2) the formula* ϕ *about the collection size.*

In the rest of the paper, we simply refer to abstract collections with placeholders as abstract collections, if the meaning is clear in the context.

Definition 7 (Match)**.** *Let* τ *be a document type and* $\mathcal{T}$ *be an augmented type. We say* τ *matches* $\mathcal{T}$*, denoted* $\tau \triangleleft \mathcal{T}$*, if (1) replacing* $?^1$ *and* $?^+$ *with exactly one and at least one attributes respectively and (2) replacing each occurrence of* Any *with a value type in* $\mathcal{T}$ *yield a type equal to* τ*.*

Having defined the match relation between document types and augmented types, we can define the relation between concrete collections and abstract collections.

Algorithm 1 Synthesis Algorithm

procedure SYNTHESIZE($\mathcal{S}, N, \vec{\mathcal{E}}$)
    Input: Database schema $\mathcal{S}$, collection name N, input-output examples $\vec{\mathcal{E}}$
    Output: A query $\mathcal{Q}$ or $\perp$ indicating failure
    $\mathcal{W} \leftarrow \{N\}$
    while $\neg$*IsEmpty*($\mathcal{W}$) **do**
        $\Omega \leftarrow \mathcal{W}$.*Dequeue*()
        if DEDUCE($\mathcal{S}, \Omega, \vec{\mathcal{E}}$) **then**
            $\mathcal{Q} \leftarrow$ COMPLETESKETCH($\mathcal{S}, \Omega, \vec{\mathcal{E}}$)
            if $\mathcal{Q} \neq \perp$ **then return** $\mathcal{Q}$
        $\mathcal{W}$.*EnqueueAll*(REFINE(Ω))
    return $\perp$

Definition 8 (Collection concretization)**.** *A collection* $\mathcal{C}$ *concretizes an abstract collection* $\tilde{\mathcal{C}} = (\mathcal{T}, \phi)$*, denoted* $\mathcal{C} \sqsubseteq \tilde{\mathcal{C}}$*, if (1)* $\tau \triangleleft \mathcal{T}$ *where* $\vdash \mathcal{C} : \mathit{Arr}\langle\tau\rangle$ *and (2)* $\mathit{SAT}(\phi \wedge l_n = |\mathcal{C}|)$ *where* $n = \mathit{MaxLabel}(\phi)$*.*

Intuitively, if collection $\mathcal{C}$ concretizes abstract collection $\tilde{\mathcal{C}} = (\mathcal{T}, \phi)$, then (1) the type of documents in $\mathcal{C}$ matches the augmented type $\mathcal{T}$ of documents in $\tilde{\mathcal{C}}$; and (2) the size of $\mathcal{C}$ is consistent with the size of $\tilde{\mathcal{C}}$ described by formula ϕ.

Example 5. *Consider the output collection* $\mathcal{C}$ *in Section II*

```
[{reply_count: 3, title: "Title-3"},
 {reply_count: 2, title: "Title-2"}]
```

Suppose $\tilde{\mathcal{C}} = (\mathcal{T}, \phi)$ *is an abstract collection where*

$$\mathcal{T} : \{?^+_0 : \mathit{Any},\ ?^+_2 : \mathit{Num}\}$$
$$\phi : l_0 = 3 \wedge l_1 \geq l_0 \wedge l_2 \leq l_1 \wedge l_3 < l_2 \wedge l_4 = l_3 \wedge l_5 \leq l_4 \wedge l_6 = l_5$$

Here, l_6 *is the variable for the size of* $\tilde{\mathcal{C}}$*. First, the type* $\{\texttt{reply\_count} : \mathit{Num}, \texttt{title} : \mathit{String}\}$ *matches the augmented type* $\mathcal{T}$*. Second, the size predicate* $l_6 = 2$ *is consistent with formula* ϕ*. Therefore,* $\mathcal{C}$ *concretizes* $\tilde{\mathcal{C}}$*.*

We can also lift the concretization relation to databases and abstract databases.

Definition 9 (DB concretization)**.** *A database* $\mathcal{D}$ *over schema* $\mathcal{S}$ *concretizes an abstract database* $\tilde{\mathcal{D}} = \{N_1 \mapsto \tilde{\mathcal{C}}_1, \ldots, N_m \mapsto \tilde{\mathcal{C}}_m\}$*, denoted* $\mathcal{D} \sqsubseteq \tilde{\mathcal{D}}$*, if for all* $1 \leq i \leq m$

$$\mathcal{S}[N_i] = \mathit{Arr}\langle\tau_i\rangle \Leftrightarrow \tilde{\mathcal{C}}_i = (\tau_i, l_0 = |\mathcal{D}[N_i]|)$$

V. SYNTHESIS USING COLLECTION ABSTRACTIONS

In this section, we present our synthesis technique based on the abstraction of collections.

A. High-Level Algorithm

As shown in Algorithm 1, our synthesis algorithm adapts the standard iterative approach based on worklists and sketches [13], [14] to the setting of document database queries. Given a database schema $\mathcal{S}$, a collection name N, and input-output examples $\vec{\mathcal{E}}$, the SYNTHESIZE procedure aims to find a query $\mathcal{Q}$ over schema $\mathcal{S}$ such that it satisfies the examples $\vec{\mathcal{E}}$. Specifically, the worklist $\mathcal{W}$ is initialized to be a singleton queue with the simplest sketch N (Line 2). While the worklist is not empty, the synthesis procedure enters a loop (Lines 3 – 8) that dequeues the current sketch Ω (Line 4) and checks if it is feasible to complete (Line 5). If yes, the procedure invokes the COMPLETESKETCH procedure and tries to obtain a correct query (Lines 6–7). If the sketch is infeasible to complete or all of its completions are incorrect, the procedure also invokes the REFINE procedure to transform the current sketch Ω to a set of sketches based on the grammar in Figure 6 (Line 8). This synthesis procedure is repeated until a correct query $\mathcal{Q}$ is found (Line 7) or returns $\perp$ if the worklist is empty.

B. Sketch Enumeration and Refinement

Definition 10 (Sketch)**.** *A sketch Ω is a query $\mathcal{Q}$ where only the collection name is known and other arguments are unknown.*

Example 6. *Let us consider again the following sketch from the motivating example.*

$$\mathit{Project}(\mathit{Match}(\mathit{Unwind}(\texttt{posts}, h_1), \phi), \vec{h_2})$$

Here, the collection name `posts` *is known, but access path h_1, predicate ϕ, and access paths $\vec{h_2}$ are unknown.*

Given a sketch Ω over collection N, the REFINE procedure substitutes the collection N with all possible query operators shown in Figure 6 and obtains a set $S_\Omega = \{\mathit{Project}(N, \vec{h}), \mathit{Match}(N, \phi), \mathit{AddFields}(N, \vec{h}, \vec{E}),$ $\mathit{Unwind}(N, h), \mathit{Group}(N, \vec{h}, \vec{a}, \vec{A}), \mathit{Lookup}(N, h, h, N, a)\}$ and produces six new sketches. The refined sketches are

$$\{\Omega[\Omega_i/N] \mid \Omega_i \in S_\Omega\}$$

C. Abstract Semantics

Since the key novelty of our synthesis technique is performing deduction on collection abstractions to prune infeasible sketches, we first introduce the abstract semantics of executing sketches over abstract collections.

At a high level, the abstract semantics is consistent with the concrete semantics in describing how an operator modifies the collection size and the type of its documents, but it applies to the abstract database. Formally, the abstract semantics is defined in Figure 7, where judgments of the form $\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda$ mean that a sketch Ω evaluates to a set of abstract collections Λ given an abstract database $\tilde{\mathcal{D}}$ and the output document type τ_O. Specifically, by the A-Collection rule, the only abstract collection for query N can be obtained by looking up the abstract database $\tilde{\mathcal{D}}$. By A-Match, *Match* reduces the collection size without changing the type of its inside documents. By A-Project, *Project* preserves the collection size but modifies the document type. In particular, the output document only retains a subset of the original attributes, and the remaining attributes can be inferred from the output. According to A-AddFields, *AddFields* adds one or more attributes of *Any* type without changing the size of the collection. By the A-Unwind rule, $\mathit{Unwind}(\Omega, h)$ potentially

$$\frac{\tilde{\mathcal{C}} = \tilde{\mathcal{D}}[N]}{\tilde{\mathcal{D}}, \tau_O \vdash N \Downarrow \{\tilde{\mathcal{C}}\}} \text{ (A-Collection)}$$

$$\frac{\begin{array}{c}\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda \quad (\mathcal{T}, \phi) \in \Lambda \\ (\mathcal{T}, \phi \wedge l_j \leq l_i) \in \Lambda' \\ \mathit{Id}(\Omega) = i \quad \mathit{Id}(\mathit{Match}(\Omega, P)) = j\end{array}}{\tilde{\mathcal{D}}, \tau_O \vdash \mathit{Match}(\Omega, P) \Downarrow \Lambda'} \text{ (A-Match)}$$

$$\frac{\begin{array}{c}\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda \quad (\mathcal{T}, \phi) \in \Lambda \\ \tau_k = \mathit{ToDocType}(\mathcal{T}) \\ ((\mathcal{T} - \tau_k) \cup (\tau_k \cap \tau_O), \phi \wedge l_j = l_i) \in \Lambda' \\ \mathit{Id}(\Omega) = i \quad \mathit{Id}(\mathit{Project}(\Omega, \vec{h})) = j\end{array}}{\tilde{\mathcal{D}}, \tau_O \vdash \mathit{Project}(\Omega, \vec{h}) \Downarrow \Lambda'} \text{ (A-Project)}$$

$$\frac{\begin{array}{c}\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda \quad (\mathcal{T}, \phi) \in \Lambda \\ (\mathcal{T} \cup \{?_0^+ : \mathit{Any}\ \}, \phi \wedge l_j = l_i) \in \Lambda' \\ \mathit{Id}(\Omega) = i \quad \mathit{Id}(\mathit{AddFields}(\Omega, \vec{h}, \vec{E})) = j\end{array}}{\tilde{\mathcal{D}}, \tau_O \vdash \mathit{AddFields}(\Omega, \vec{h}, \vec{E}) \Downarrow \Lambda'} \text{ (A-AddFields)}$$

$$\frac{\begin{array}{c}\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda \quad (\mathcal{T}, \phi) \in \Lambda \\ \mathit{Type}(a_A) = \mathit{Arr}\langle\tau\rangle \wedge \mathit{NotInArr}(a_A) \\ \{(\mathcal{T}[\tau/a_A], \phi \wedge l_j \geq l_i) | a_A \in \mathcal{T} \wedge \forall p. \forall q. a_A \neq ?_p^q\} \subseteq \Lambda' \\ \mathit{Id}(\Omega) = i \quad \mathit{Id}(\mathit{Unwind}(\Omega, h)) = j\end{array}}{\tilde{\mathcal{D}}, \tau_O \vdash \mathit{Unwind}(\Omega, h) \Downarrow \Lambda'} \text{ (A-Unwind)}$$

$$\frac{\begin{array}{c}\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda \quad (\mathcal{T}, \phi) \in \Lambda \\ F = \{\mathit{ToDocType}(\tilde{\mathcal{D}}[N]_{\mathcal{T}}) | N \in \mathit{dom}(\tilde{\mathcal{D}})\} \\ \{(\mathcal{T} \cup \{?_j^1 : \mathit{Arr}\langle\tau_F\rangle\}, \phi \wedge l_j = l_i) | \tau_F \in F\} \subseteq \Lambda' \\ \mathit{Id}(\Omega) = i \quad \mathit{Id}(\mathit{Lookup}(\Omega, h, h, N, a)) = j\end{array}}{\tilde{\mathcal{D}}, \tau_O \vdash \mathit{Lookup}(\Omega, h, h, N, a) \Downarrow \Lambda'} \text{ (A-Lookup)}$$

$$\frac{\begin{array}{c}\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda \quad (\mathcal{T}, \phi) \in \Lambda \\ G = \{\{?_j^+ : \mathit{Num}\ \}, \{\}\} \\ \{(\{\texttt{\_id} : \tau_K\} \cup \tau_g, \phi \wedge l_j < l_i) \\ | \tau_K \subseteq \mathit{ToDocType}(\mathcal{T}) \wedge \tau_g \in G\} \subseteq \Lambda' \\ \mathit{Id}(\Omega) = i \quad \mathit{Id}(\mathit{Group}(\Omega, \vec{h}, \vec{a}, \vec{A}) = j\end{array}}{\tilde{\mathcal{D}}, \tau_O \vdash \mathit{Group}(\Omega, \vec{h}, \vec{a}, \vec{A}) \Downarrow \Lambda'} \text{ (A-Group)}$$

Fig. 7: Abstract Semantics. The *ToDocType* function transforms an augmented type to a document type by deleting all placeholder attributes and the attributes with *Any* type. The *NotInArr* checks whether an attribute is not nested in an array type otherwise it is unable to be unwinded.

increases the collection size, deconstructs the array h of sketch Ω, and updates the type accordingly. By the A-Lookup rule, *Lookup* preserves the collection size but introduces a new attribute to the $\mathcal{T}$, where the type of the new attribute is the same as that of the foreign collection. Finally, as shown in the A-Group rule, *Group* reduces the collection size and constructs a new type. In particular, it introduces a new attribute `_id` as the key and uses a new τ_g to represent a series of numeric attributes for aggregation results. τ_g can also be empty, indicating the absence of aggregation attributes.

Example 7. *Consider again the following sketch in Section II*

$$\mathit{Project}(\mathit{Match}(\mathit{Unwind}(\texttt{posts}, h_1), \phi), \vec{h_2})$$

Based on the rules in Figure 8, we recursively evaluate the sketch. The evaluation result of `posts` *is*

$$\{(\{\texttt{\_id} : \mathit{String}, \texttt{title} : \mathit{String}, \texttt{replies} : \\ \mathit{Arr}\langle\{\texttt{depth} : \mathit{Num}\}\rangle\}, l_0 = 3)\}$$

The result of Unwind($\texttt{posts}, h_1$) *is*

$$\{(\{\_\texttt{id} : \textit{String}, \texttt{title} : \textit{String}, \texttt{replies} : \{\texttt{depth} : \textit{Num}\}\}, l_0 = 3 \wedge l_1 \geq l_0)\}$$

The result of Match(Unwind($\texttt{posts}, h_1$), ϕ) *is*

$$\{(\{\_\texttt{id} : \textit{String}, \texttt{title} : \textit{String}, \texttt{replies} : \{\texttt{depth} : \textit{Num}\}\}, l_0 = 3 \wedge l_1 \geq l_0 \wedge l_2 \leq l_1)\}$$

The result of Project(Match(Unwind($\texttt{posts}, h_1$), ϕ), $\vec{h_2}$) *is*

$$\{(\{\texttt{title} : \textit{String}\}, l_0 = 3 \wedge l_1 \geq l_0 \wedge l_2 \leq l_1 \wedge l_3 = l_2)\}$$

Next, we establish the relationship among queries, sketches, concrete semantics, and abstract semantics with a theorem.

Theorem 1. [4] *Let $\tilde{\mathcal{D}}$ be an abstract database over schema $\mathcal{S}$, Ω be a sketch, $\mathcal{Q}$ be a query that is a completion of Ω, and (I, O) be an input-output example, where $\vdash I : \mathcal{S}$ and $\vdash O : \textit{Arr}\langle\tau_O\rangle$. If $[\![\mathcal{Q}]\!]_I = O$, $I \sqsubseteq \tilde{\mathcal{D}}$, and $\tilde{\mathcal{D}}, \tau_O \vdash \Omega \Downarrow \Lambda$, then there exists an abstract collection $\tilde{\mathcal{C}} \in \Lambda$ such that $O \sqsubseteq \tilde{\mathcal{C}}$.*

Intuitively, the theorem states that the abstract semantics is correct with respect to the concrete semantics. In particular, if the input is a concretization of the abstract database and the query is a completion of the sketch, then the evaluation result of the sketch on the abstract database is an over-approximation of the output produced by executing the query on the input.

D. Deduction by Collection Abstractions

Next, let us present how to perform deduction based on the collection abstractions.

Deduction algorithm. Our deduction algorithm is shown in Algorithm 2. For each example $\mathcal{E}_j = (I_j, O_j)$, we compute the document type in O_j. The COMPUTETYPE computes the type of O_j by the typing rules in Figure 5 (Line 4). Then the *In* function extracts the document type from the type of O_j, namely $\textit{In}(\textit{Arr}\langle\tau\rangle) = \tau$. We also compute the abstract input database $\tilde{\mathcal{D}}_j$ by computing all the abstractions of collections in the database (Line 5). Each collection name N_i is mapped to an abstract collection whose augmented type is the document's type inside the collection and the predicate is l_0 equals the collection size. For all pairs of abstract input database $\tilde{\mathcal{D}}_j$ and output document type τ_{Oj}, we evaluate the sketch Ω based on the abstract semantics in Figure 7 and get a set of abstract collections for each example (Line 6). If for each example (I_j, O_j), there is an abstract collection $\tilde{\mathcal{C}} \in \Lambda_j$ such that O_j is a concretization of $\tilde{\mathcal{C}}$, then the sketch is feasible to complete (Line 7). Otherwise, the sketch is infeasible.

Concretization check. Recall from Definition 8 that to check a collection $\mathcal{C}$ is a concretization of abstract collection $\tilde{\mathcal{C}} = (\mathcal{T}, \phi)$, we need to check (1) the type τ of documents inside $\mathcal{C}$ matches $\mathcal{T}$, i.e. $\tau \triangleleft \mathcal{T}$, and (2) the size of $\mathcal{C}$ is consistent with the formula ϕ. We use an off-the-shelf SMT solver to check condition (2) by checking the satisfiability of formula $\phi \wedge l_n = |\mathcal{C}|$ where $n = \textit{MaxLabel}(\phi)$. We also develop a procedure for type match based on Definition 7, which can be best explained with the following example.

[4]Proofs of all the theorems are available in the technical report [25].

Algorithm 2 Deduction by Abstract Collections

1: **procedure** DEDUCE($\mathcal{S}, \Omega, \vec{\mathcal{E}}$)
  Input: The database schema $\mathcal{S}$, a sketch Ω and input-output examples $\vec{\mathcal{E}}$
  Output: $\top$ if Ω is feasible otherwise $\bot$
2:   **for** $j \leftarrow 1$ to $|\vec{\mathcal{E}}|$ **do**
3:     $(I_j, O_j) \leftarrow \mathcal{E}_j$
4:     $\tau_{Oj} \leftarrow \textit{In}(\text{COMPUTETYPE}(O_j))$
5:     $\tilde{\mathcal{D}}_j \leftarrow \{N_i \mapsto (\textit{In}(\mathcal{S}[N_i]), l_0 = |I_j[N_i]|) | N_i \in \textit{dom}(\mathcal{S})\}$
6:     $\Lambda_j \leftarrow \text{EVAL}(\tilde{\mathcal{D}}_j, \tau_{Oj}, \Omega)$
7:   **if** $\forall j. \exists \tilde{\mathcal{C}}. \tilde{\mathcal{C}} \in \Lambda_j \wedge O_j \sqsubseteq \tilde{\mathcal{C}}$ **then return** $\top$
8:   **else return** $\bot$

Example 8. *Suppose we have an augmented type*

$$\mathcal{T} = \{\texttt{name} : \textit{String}, \texttt{id} : \textit{String}, \texttt{info} : \{\texttt{tel} : \textit{String}\}, ?_1^+ : \textit{Num}, ?_2^+ : \textit{Any}, ?_3^1 : \textit{Arr}\langle\{\texttt{profId} : \textit{String}, \texttt{profName} : \textit{String}\}\rangle\}$$

and document type τ

$$\{\texttt{id} : \textit{String}, \texttt{name} : \textit{String}, \texttt{info} : \{\textit{tel} : \textit{String}\}, \texttt{newField} : \textit{Bool}, \texttt{sum} : \textit{Num}, \texttt{profs} : \textit{Arr}\langle\{\texttt{profId} : \textit{String}, \texttt{profName} : \textit{String}\}\rangle\}$$

Here, $\{\texttt{name} : \textit{String}, \texttt{id} : \textit{String}, \texttt{info} : \{\texttt{tel} : \textit{String}\}\}$ in $\mathcal{T}$ is matched by $\{\texttt{name} : \textit{String}, \texttt{id} : \textit{String}, \texttt{info} : \{\texttt{tel} : \textit{String}\}\}$ in τ, because the corresponding attributes have the same names and types. $\{?_3^1 : \texttt{Arr}\langle\{\texttt{profId} : \textit{String}, \texttt{profName} : \textit{String}\}\rangle\}$ is matched by $\{\texttt{profs} : \texttt{Arr}\langle\{\texttt{profId} : \textit{String}, \texttt{profName} : \textit{String}\}\rangle\}$, because $\texttt{profs}$ has the same type as placeholder $?_3^1$ and $?_3^1$ matches exactly one attribute. Finally, $\{?_1^+ : \textit{Num}\}$ is matched by $\{\texttt{sum} : \textit{Num}\}$ because they have the same type, and $\{?_2^+ : \textit{Any}\}$ is matched by $\{\texttt{newField} : \textit{Bool}\}$ because Any can match any value type.

To understand why our deduction algorithm is correct, let us consider the following theorem.

Theorem 2. *Given a database schema $\mathcal{S}$, a sketch Ω, and input-output examples $\vec{\mathcal{E}}$, if* DEDUCE$(\mathcal{S}, \Omega, \vec{\mathcal{E}})$ *returns $\bot$, then there is no completion $\mathcal{Q}$ of Ω such that for all $(I, O) \in \vec{\mathcal{E}}$, $[\![\mathcal{Q}]\!]_I = O$.*

Intuitively, the theorem states that our deduction-based pruning is sound. In other words, if the deduction algorithm returns $\bot$ for a sketch, then no completions of the sketch satisfy all the input-output examples.

E. Sketch Completion

The COMPLETESKETCH takes as input a schema $\mathcal{S}$, a sketch Ω, and input-output examples $\vec{\mathcal{E}}$ and returns a query $\mathcal{Q}$ satisfying all examples or $\bot$ if such a query does not exist.

We use an enumerative search algorithm to fill unknowns in the sketch according to the query operators.

1) *Project*. We compute the common attributes in the input and output and use these common attributes as arguments.
2) *Match*. We enumerate all predicates obtained from a combination of access paths, constants, comparisons, and logic connectives. Also, the observational equivalent class is used to avoid duplicate predicates.
3) *AddFields*. We enumerate all possible expressions for newly generated attributes.
4) *Unwind*. We enumerate all array attributes in the top level of the document and unwind them.
5) *Group*. We enumerate all group keys and accumulators and use value-based analysis to prune impossible accumulators.
6) *Lookup*. We enumerate all foreign collections and their attributes as arguments.

In addition, we also perform type checking to prune impossible arguments. For instance, if the value for a newly generated attribute has a different type than it should be in the output, we prune this completion from the search space.

We now conclude this section with two theorems about the overall synthesis algorithm.

Theorem 3 (Soundness)**.** *Let $\mathcal{S}$ be a database schema, $\vec{\mathcal{E}}$ be input-output examples, and N be a collection name. Suppose* COMPLETESKETCH *is sound, if* SYNTHESIZE$(\mathcal{S}, \vec{\mathcal{E}}, N)$ *returns a query $\mathcal{Q}$, then $\mathcal{Q}$ satisfies examples $\vec{\mathcal{E}}$.*

Theorem 4 (Completeness)**.** *Let $\mathcal{S}$ be a database schema, $\vec{\mathcal{E}}$ be input-output examples, and N be a collection name. Suppose* COMPLETESKETCH *is complete, if there exists a query accepted by the grammar in Figure 6 that is over collection N and satisfies examples $\vec{\mathcal{E}}$, then* SYNTHESIZE$(\mathcal{S}, \vec{\mathcal{E}}, N)$ *does not return $\perp$.*

Intuitively, the soundness theorem states that if the synthesis algorithm returns a query, then the query satisfies all input-output examples. The completeness theorem ensures that if there exists a query in our language satisfying all input-output examples, then the synthesis algorithm can find a query.

VI. IMPLEMENTATION

We have implemented the proposed synthesis technique in a tool called NOSDAQ and use Z3 [11] as the SMT solver.

Heuristics for sketch completion. Based on the observation that most *Group* operators do not have more than two group keys, we limit the number of group keys to two during sketch completion. In addition, although NOSDAQ supports simple constants (e.g., null) in sketch completion, it expects the user to provide more complicated constants such as string literals.

Translation to MongoDB queries. NOSDAQ performs syntax-directed translation to transform the document database query in its domain-specific language to the MongoDB query language. Furthermore, it also performs optimizations to improve the conciseness and efficiency of translated queries, such as merging continuous *AddFields* and *Project* operators.

TABLE I: Statistics of datasets. **#n** is the number of benchmarks. **#a**, **#d**, **#e**, **#i**, **#o**, **#c** denote the average number of document attributes, document depths, examples, collection sizes in input and output examples, and constants, respectively.

dataset	#n	#a	#d	#e	#i	#o	#c
StackOverflow	33	4.9	1.5	2.5	2.4	1.4	0.9
MongoDB Document	26	5.4	1.4	1.1	4.7	2.6	0.6
Twitter API	5	18.4	2.6	1.0	2.0	2.6	0.0
Kaggle	46	19.8	4.1	1.0	1.8	3.6	0.5
Total	110	11.9	2.6	1.5	2.6	2.3	0.6

TABLE II: Statistics of ground truth queries. **#s**, **#op**, **#P**, **#M**, **#L**, **#U**, **#G**, **#A** denote the number of AST nodes, query operators, *Project*, *Match*, *Lookup*, *Unwind*, *Group*, *AddFields*, respectively.

dataset		#s	#op	#P	#M	#L	#U	#G	#A
	avg	12	1.88	0.42	0.79	0.03	0.27	0.36	0
Stack-Overflow	med	10	1	0	1	0	0	0	0
	min	4	1	0	0	0	0	0	0
	max	33	5	1	2	1	2	2	0
	avg	8	1.15	0.31	0.42	0.04	0.08	0.27	0.04
Official Document	med	7	1	0	0	0	0	0	0
	min	4	1	0	0	0	0	0	0
	max	17	3	1	1	1	2	1	1
	avg	16	2.6	0.8	0	0	1	0.6	0.2
Twitter API	med	14	2	1	0	0	1	1	0
	min	9	2	0	0	0	0	0	0
	max	26	4	1	0	0	2	1	1
	avg	13	3.2	0.7	0.52	0	1.54	0.43	0
Kaggle	med	12	3	1	0	0	2	0	0
	min	8	2	0	0	0	0	0	0
	max	27	6	1	2	0	3	2	0
	avg	12	2.29	0.53	0.55	0.02	0.79	0.38	0.02
Total	med	11	2	1	1	0	1	0	0
	min	4	1	0	0	0	0	0	0
	max	33	6	1	2	1	3	2	1

VII. EVALUATION

In this section, we present several experiments that are designed to answer the following research questions.

RQ1. Is NOSDAQ effective and efficient to synthesize document database queries from input-output examples?
RQ2. How does each component of the collection abstraction affect synthesis time?
RQ3. How does NOSDAQ compare against other baseline synthesizers?
RQ4. How does the collection size of input-output examples impact the performance of NOSDAQ?

Experimental setup. All experiments are conducted on a machine with an Intel i9-13905H CPU and 32 GB of physical memory, running the Ubuntu 22.04 WSL2 operating system.

A. Benchmarks

We have collected 110 benchmarks from 4 representative sources, i.e., StackOverflow, MongoDB official document, Twitter API documents, and Kaggle competitions, which cover a wide spectrum of realistic scenarios.

- StackOverflow. The StackOverflow dataset is adapted from StackOverflow posts where developers ask about real-world problems. Each post in our dataset has 453K visits, 4 answers, and 127 votes on average, which demonstrates these queries attract lots of attention from the community. Most of the the examples and constants are extracted from the post content. If some post does not provide enough examples, we add the examples.

- MongoDB Document. The MongoDB official documents cover a representative set of queries that the MongoDB community believes are commonly used in practice. The examples and constants are all collected from the example section of official documents.
- Twitter API. The Twitter dataset consists of tweets and user replies which mainly focus on calculating tweet statistics, such as the count of replies. The benchmarks represent typical scenarios for data analysts to get information from social networks and online forums. The examples are collected from the response data of APIs.
- Kaggle. The Kaggle dataset contains information about satellite images, where benchmarks reflect scenarios for scientific research, such as extracting different labels for training machine learning models and collecting statistics. The examples are sampled from the provided JSON file.

Table I summarizes the statistics of these datasets. Among these datasets, Twitter API and Kaggle benchmarks are more complex than StackOverflow and MongoDB Document in terms of the number of attributes, depths, etc.

To further understand the complexity of benchmarks, we have also collected the statistics on the ground truth queries in Table II. The maximum AST size of ground truth queries is 33 among all benchmarks, and the average is 12. Over half of the ground truth queries have an AST size larger than 10. This indicates a high level of complexity, as longer queries typically require synthesizers to explore a larger search space. Furthermore, the number of operators (or pipeline stages) in a single query ranges from 1 to 6. Frequently occurring operators include *Project*, *Match*, *Unwind*, and *Group*. Notably, *Unwind* and *Group* pose significant challenges for synthesis, as they can substantially change the structure of collections and documents. In contrast, the *Lookup* operator appears infrequently in ground truth queries. This is consistent with the typical usage of document databases where users try to avoid "join" operations between multiple collections. Similarly, the *AddFields* operator is also not used frequently in our datasets.

B. Effectiveness and Efficiency

The evaluation results and the statistics of synthesized programs are presented in Table III. Given a time limit of 5 minutes, NOSDAQ can solve 108 out of 110 benchmarks and only gets timeout on two challenging benchmarks (both in Kaggle). Note that the ground-truths of these two benchmarks are more complex than the others from our manual inspection. This serves as evidence of the effectiveness of NOSDAQ in synthesizing document database queries from examples. Further, NOSDAQ can solve most benchmarks in an average of 14.2 seconds as shown in Table III. Furthermore, observing the number of sketches $\#\Omega$ and complete programs $\#\mathcal{Q}$, NOSDAQ iterates over 175 sketches but only completes 57 full programs on average. It demonstrates that our synthesis technique based on collection abstractions is efficient in pruning infeasible sketches and thus speeds up the synthesis process.

TABLE III: Evaluation results for NOSDAQ. **#n** and ✓ denote the number of benchmarks and solved benchmarks, and **time** indicates the time (in seconds) to solve benchmarks. $\#\Omega$, $\#\mathcal{Q}$, **#size** refer to the number of sketches, complete programs, and AST nodes of synthesized programs, respectively.

dataset	#n	✓		time (s)	#Ω	#Q	#size
Stack-Overflow	33	33	avg	9.2	86	31	12
			med	2.6	15	6	11
			min	0.5	2	1	4
			max	184.5	854	308	32
MongoDB Document	26	26	avg	5.7	11	53	9
			med	1.1	6	14	10
			min	0.5	2	1	4
			max	78.6	124	576	19
Twitter API	5	5	avg	10.5	81	61	15
			med	10.1	36	81	15
			min	1.8	15	1	9
			max	19.9	165	131	22
Kaggle	46	44	avg	23.4	350	79	16
			med	6.8	160	4	13
			min	1.0	8	1	8
			max	201.6	3975	1235	38
Total	110	108	avg	14.2	175	57	13
			med	3.2	31	7	11
			min	0.5	2	1	4
			max	201.6	3975	1235	38

Qualitative analysis. We observe that the number of attributes in the document, the depth of the document, the number of constants, and the query complexity affect the synthesis time. For instance, the Kaggle dataset needs longer synthesis time than others because the benchmark has a large number of attributes and the documents are deeply nested. In general, more complex queries need the synthesizer to iterate more sketches. More attributes, deeper nesting, and more constants require enumerating more queries while completing the sketch.

Non-desired programs. To understand if NOSDAQ can synthesize desired queries, we have manually inspected all 108 synthesized queries and found 107 of them are equivalent to the desired ones. There is only one benchmark (from StackOverflow) where NOSDAQ synthesized a plausible query in terms of the example but the query is not desired. The reason is that this benchmark involves a complex predicate that requires numerous unseen examples to eliminate mismatch cases. However, only a few examples are provided on the StackOverflow post, so NOSDAQ cannot find the desired predicate but synthesize an alternative satisfying the examples.

Answer to RQ1: NOSDAQ successfully synthesizes 108 out of 110 benchmarks from examples and the average synthesis time is 14.2 seconds.

C. Ablation Study

To understand how the type and size information in collection abstractions may affect the efficiency, we perform an ablation study. Specifically, we have created three variants of NOSDAQ that disable (1) the size information, (2) the type

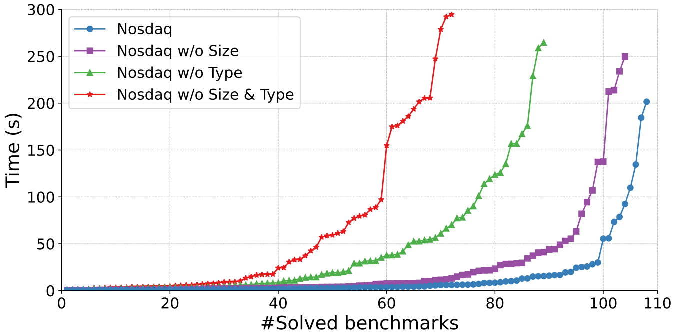

Fig. 8: Ablation study.

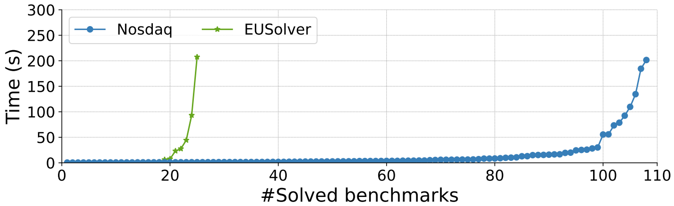

Fig. 9: Comparison between NOSDAQ and EUSOLVER.

information, and (3) both size and type information in the abstraction. We run all these variants on the 110 benchmarks and obtain the result shown in Figure 8, where a point (x, y) means a variant can synthesize x benchmarks and the time for each benchmark is within y seconds. As shown in the figure, without size in the abstraction, the variant times out on 4 more benchmarks and requires approximately 10 seconds longer on average. Without type, the variant triggers timeout on 19 more benchmarks and requires around 27 seconds longer on average to complete the synthesis process. This indicates that the document type in the collection abstraction significantly improves the synthesis time.

Answer to RQ2: Both type and size information can make NOSDAQ more efficient but the former is more significant.

D. Comparison with Baselines

To compare NOSDAQ with a baseline, we have instantiated the EUSOLVER framework [1] to synthesize document database queries from examples. As a generic solver, EUSOLVER can be easily extended to support documents and collections in the specification, since it provides necessary support for lists and maps. Secondly, EUSOLVER remains a competitive baseline in program synthesis, as evidenced from recent work [4], [22], [32]. As shown in Figure 9, as opposed to 108 benchmarks solved by NOSDAQ, EUSOLVER can only solve 25 benchmarks within the 5-minute time limit due to the large search space of document database queries in general.

To compare NOSDAQ with the LLM-based approach, we have used ChatGPT (version gpt-4o-2024-08-06) to synthesize all of our 110 benchmarks. Specifically, we have used the same set of input-output examples and constants in each benchmark and asked ChatGPT to generate MongoDB queries. To make fair comparisons, we did not provide additional natural language descriptions about what the query should do. The evaluation shows that GPT can only generate the desired query for 53 out of 110 benchmarks. For 24 benchmarks, the generated query is plausible but undesired, i.e., it is consistent with the examples but not equivalent to the desired one. For the remaining 33 benchmarks, the generated query is inconsistent with the input-output examples. The errors made by GPT include misunderstanding the semantics of operators, missing predicates, etc. Recall that NOSDAQ can synthesize desired queries for 107 benchmarks and plausible but undesired query for 1 benchmark. We believe our synthesis technique is more effective and generalizable than GPT to synthesize document database queries from examples.

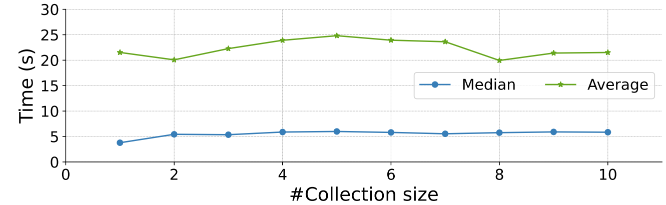

Fig. 10: Impact of collection size on synthesis time.

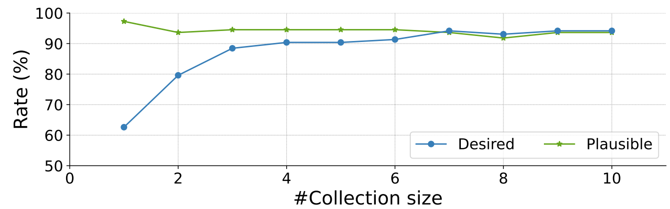

Fig. 11: Impact of collection size on rates of plausible and desired queries.

Answer to RQ3: NOSDAQ can solve 108 out of 110 benchmarks, whereas EUSOLVER can only solve 25 benchmarks, and ChatGPT-4o can solve 77 benchmarks.

E. Impact of Collection Size

To analyze the impact of collection size on the performance of NOSDAQ, we have conducted experiments across all 110 benchmarks to evaluate how different collection sizes influence NOSDAQ's behavior. Specifically, we sampled 10 documents for each collection and ran NOSDAQ on variants with collection sizes ranging from 1 to 10 documents. The impact on synthesis time is presented in Figure 10, while the impact on rates of plausible and desired queries are shown in Figure 11.

The plausible rate is defined as the ratio of benchmarks synthesized within a 5-minute time limit to the total number of benchmarks. The desired rate represents the ratio of benchmarks for which the synthesized query is equivalent to the desired one to the total number of synthesized benchmarks.

As shown in the figures, the synthesis time of NOSDAQ remains relatively insensitive to changes in collection size within the range of 1 to 10 documents in each collection. Similarly, the plausible rate also remains stable. In contrast, the desired rate shows a significant increase when the collection size grows from 1 to 3, after which it stabilizes. This can be attributed to the fact that smaller collection sizes provide insufficient examples to synthesize the desired query, leading to simpler queries that are plausible but not desired.

Answer to RQ4: The synthesis time of NOSDAQ demonstrates minimal sensitivity to changes in collection size. The rate of synthesizing a desired query increases rapidly as the collection size grows from 1 to 3 and stabilizes thereafter.

F. Threats to Validity

First, although we believe our datasets are representative, which are obtained from various real-world scenarios, our evaluation results are limited to the collected datasets. The NOSDAQ tool might perform differently on other datasets. Second, our domain-specific language only corresponds to a core subset of the MongoDB aggregation pipeline. While it is convenient to extend the abstract semantics to other query operators, the performance of the tool might be different due to the change in SMT formulas for symbolic reasoning. Third, all the experiments are conducted on a machine as specified in Section VII. Running the experiments on a different machine may yield different results.

VIII. RELATED WORK

Program synthesis for software engineering. Program synthesis techniques have been applied to address various software engineering problems, such as program refactoring [9], [33], [37], [39], program repair [26], [28], [34], [50], code completion [16], [40], software testing [43], [55], and so on. This paper focuses on the topic of generating document database queries from input-output examples.

Synthesizing database queries. Among related papers, the most related is a body of work on synthesizing database queries. SQLSYNTHESIZER [54], SCYTHE [47] and PATSQL [44] synthesize SQL queries for relational databases from examples, while SICKLE [56] synthesizes analytical SQL queries given computation demonstrations. SQLIZER [53] considers nature language description as the specification for SQL query synthesis. However, none of the prior work can synthesize queries of document databases such as MongoDB.

Synthesis with deduction. A line of work performs deduction to prune infeasible programs in program synthesis [7], [8], [13]–[15], [21], [23], [24], [36], [38]. For example, MORPHEUS [14] and NEO [13] utilize SMT-based deduction that generates formulas based on semantics and input-output examples to prune infeasible programs. NGDS [23] and CONCORD [8] combine deduction and machine learning techniques to prune the search space. NOSDAQ adapts the high-level approach of MORPHEUS [14] and NEO [13] to the setting of document database queries. However, MORPHEUS mainly focuses on tabular data, whereas NOSDAQ focuses on hierarchical data. The abstraction used by MORPHEUS is related to the number of rows and columns of tables. This abstraction cannot be directly used for deduction in a synthesizer that aims to generate document database queries, because these queries operate over more involved hierarchical data. Therefore, NOSDAQ uses the novel abstraction consisting of hierarchically nested types for its documents and the collection size, which is one of the main contributions of this paper.

Synthesis with abstraction. Another line of related work is to synthesize programs using abstractions [19], [29], [42], [46], [48]. For example, SIMPL [42] uses abstract interpretation to guide the synthesis of imperative programs from examples. Mell et al. [29] also use abstract interpretation for optimal program synthesis. BLAZE [48] constructs and iteratively refines the abstract finite tree automata that represent a set of programs. This approach iteratively prunes and refines automata when the corresponding programs do not satisfy examples, until a correct program is found. Unlike prior techniques, NOSDAQ employs a novel collection abstraction to represent complex hierarchical data (e.g., BSON) and uses abstract semantics to rule out infeasible sketches representing a large set of programs.

Wrangling semi-structured data. Various techniques have been proposed to wrangle semi-structured data, such as JSON, XML documents, spreadsheets, and log files. For example, there is a line of work [2], [3], [41], [45] that aims to map XML documents to relational data for query processing. DATAMARAN [17] converts the semi-structured log into a structured relational format. FLASHEXTRACT [24] extracts relevant data from text files, websites, and spreadsheets. FLASHRELATE [5] extracts relational data from semi-structured spreadsheets by examples. TREEX [35] synthesizes extractors for real-world large-scale websites. Since document databases store semi-structured data by nature, NOSDAQ can also be viewed as a query synthesizer over semi-structured data. However, different from prior work, NOSDAQ focuses on core query language of document databases and aims to address significant challenges raised by specialized operators such as *Group*, *Unwind*, and *Lookup*.

Synthesizing data transformation scripts. Many synthesizers aim to automatically generate data transformation scripts from high-level specifications [6], [12], [14], [18], [20], [27], [51], [52]. For example, HADES [51] synthesizes scripts to handle hierarchically structured data such as file systems, XML, and HDF files. MITRA [52] aims to synthesize scripts to convert hierarchical data into relational tables. DYNAMITE [49] transforms data between various types of databases by synthesizing Datalog programs. In contrast, NOSDAQ is designed to handle complex data structures in document databases and leverage collection abstractions to efficiently synthesize queries from examples, which is beyond the capability of prior work. For instance, HADES focuses on structure changes in the transformation but does not support aggregations, but NOSDAQ can synthesize aggregate queries with *Group* operations.

IX. CONCLUSION

This paper presents a technique that automatically synthesizes document database queries from input-output examples. To achieve better performance, we develop a novel abstraction for collections containing hierarchical and nested data structures and leverage this abstraction for deduction to prune the search space of target queries. An evaluation of 110 benchmarks from various sources demonstrates our technique is effective and efficient in solving 108 benchmarks.

References

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 319–336, 2017.

[2] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the xml-to-relational mapping problem. In *Proceedings of the ACM International Workshop on Web Information and Data Management (WIDM)*, page 31–38, 2004.

[3] Mustafa Atay, Artem Chebotko, Dapeng Liu, Shiyong Lu, and Farshad Fotouhi. Efficient schema-based xml-to-relational data mapping. *Information Systems*, 32(3):458–476, 2007.

[4] Celeste Barnaby, Qiaochu Chen, Roopsha Samanta, and Isil Dillig. Imageeye: Batch image processing using program synthesis. *Proc. ACM Program. Lang.*, 7(PLDI):686–711, 2023.

[5] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 218–228. ACM, 2015.

[6] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[7] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multimodal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 487–502. ACM, 2020.

[8] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pages 587–610. Springer, 2020.

[9] Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Isil Dillig. Synthesis-powered optimization of smart contracts via data type refactoring. *Proc. ACM Program. Lang.*, 6(OOPSLA2):560–588, 2022.

[10] CouchDB. https://couchdb.apache.org, 2024.

[11] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.

[12] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI, page 1–12. ACM, 2020.

[13] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 420–435. ACM, 2018.

[14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 422–436. ACM, 2017.

[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 229–239. ACM, 2015.

[16] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. Codehint: dynamic and interactive synthesis of code snippets. In *International Conference on Software Engineering (ICSE)*, pages 653–663. ACM, 2014.

[17] Yihan Gao, Silu Huang, and Aditya Parameswaran. Navigating the data lake with datamaran: Automatically extracting structure from log datasets. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD, page 943–958. ACM, 2018.

[18] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, page 317–330. ACM, 2011.

[19] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[20] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 317–328. ACM, 2011.

[21] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 1159–1174. ACM, 2020.

[22] Ruyi Ji, Chaozhe Kong, Yingfei Xiong, and Zhenjiang Hu. Improving oracle-guided inductive synthesis by efficient question selection. *Proc. ACM Program. Lang.*, 7(OOPSLA1):819–847, 2023.

[23] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018.

[24] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 542–553. ACM, 2014.

[25] Qikang Liu, Yang He, Yanwen Cai, Byeongguk Kwak, and Yuepeng Wang. Synthesizing document database queries using collection abstractions, 2024.

[26] Fan Long and Martin C. Rinard. Staged program repair with condition synthesis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 166–178. ACM, 2015.

[27] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: an extensible synthesis framework for data science. *Proc. VLDB Endow.*, 12(12):1914–1917, aug 2019.

[28] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 691–701. ACM, 2016.

[29] Stephen Mell, Steve Zdancewic, and Osbert Bastani. Optimal program synthesis via abstract interpretation. *Proc. ACM Program. Lang.*, 8(POPL):457–481, 2024.

[30] MongoDB. https://www.mongodb.com, 2024.

[31] MongoDB. Why use mongodb and when to use it? https://www.mongodb.com/resources/products/fundamentals/why-use-mongodb, 2024.

[32] Amirmohammad Nazari, Yifei Huang, Roopsha Samanta, Arjun Radhakrishna, and Mukund Raghothaman. Explainable program synthesis by localizing specifications. *Proc. ACM Program. Lang.*, 7(OOPSLA2):2171–2195, 2023.

[33] Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Inês Lynce, Vasco M. Manquinho, Ruben Martins, and Claire Le Goues. SOAR: A synthesis approach for data science API refactoring. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 112–124. IEEE, 2021.

[34] Wonseok Oh and Hakjoo Oh. Pyter: effective program repair for python type errors. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 922–934. ACM, 2022.

[35] Adi Omari, Sharon Shoham, and Eran Yahav. Synthesis of forgiving data extractors. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, pages 385–394. ACM, 2017.

[36] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 619–630. ACM, 2015.

[37] Shankara Pailoor, Yuepeng Wang, and Isil Dillig. Semantic code refactoring for abstract data types. *Proc. ACM Program. Lang.*, 8(POPL):816–847, 2024.

[38] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, page 107–126. ACM, 2015.

[39] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin T. Vechev. Refactoring with synthesis. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, (OOPSLA)*, pages 339–354. ACM, 2013.

[40] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–428. ACM, 2014.

[41] Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton, and Igor Tatarinov. A general technique for querying xml documents using a relational database system. *SIGMOD Rec.*, 30(3):20–26, sep 2001.

[42] Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *International Symposium on Static Analysis (SAS)*, pages 364–381. Springer, 2017.

[43] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Pasareanu. Syrust: automatic testing of rust libraries with semantic-aware program synthesis. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 899–913. ACM, 2021.

[44] Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. Patsql: efficient synthesis of sql queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.*, 14(11):1937–1949, jul 2021.

[45] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD, page 204–215. ACM, 2002.

[46] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 327–338. ACM, 2010.

[47] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 452–466. ACM, 2017.

[48] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[49] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. Data migration using datalog program synthesis. *Proc. VLDB Endow.*, 13(7):1006–1019, mar 2020.

[50] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE / ACM, 2017.

[51] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 508–521. ACM, 2016.

[52] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *Proc. VLDB Endow.*, 11(5):580–593, oct 2018.

[53] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[54] Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE, page 224–234. IEEE Press, 2013.

[55] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. History-driven test program synthesis for JVM testing. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1133–1144. ACM, 2022.

[56] Xiangyu Zhou, Rastislav Bodik, Alvin Cheung, and Chenglong Wang. Synthesizing analytical sql queries from computation demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI, page 168–182. ACM, 2022.