Recursive Question Understanding for Complex Question Answering over Heterogeneous Personal Data

Philipp Christmann
Max Planck Institute for Informatics
Saarland Informatics Campus
pchristm@mpi-inf.mpg.de

Gerhard Weikum
Max Planck Institute for Informatics
Saarland Informatics Campus
weikum@mpi-inf.mpg.de

Abstract

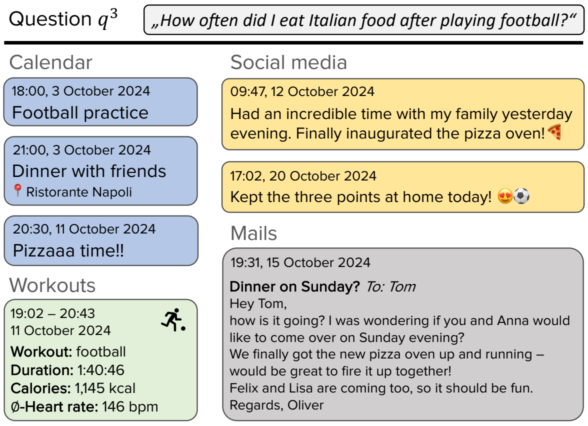
Question answering over mixed sources, like text and tables, has been advanced by verbalizing all contents and encoding it with a language model. A prominent case of such heterogeneous data is personal information: user devices log vast amounts of data every day, such as calendar entries, workout statistics, shopping records, streaming history, and more. Information needs range from simple look-ups to queries of analytical nature. The challenge is to provide humans with convenient access with small footprint, so that all personal data stays on the user devices. We present REQAP, a novel method that creates an executable operator tree for a given question, via recursive decomposition. Operators are designed to enable seamless integration of structured and unstructured sources, and the execution of the operator tree yields a traceable answer. We further release the PERQA benchmark, with persona-based data and questions, covering a diverse spectrum of realistic user needs.

1 Introduction

1.1 Motivation and Problem Statement

Question answering (QA) over combinations of structured and textual data has been substantially advanced by verbalizing all contents and encoding it with a large language model (LLM) (Yin et al., 2020; Thorne et al., 2021; Oğuz et al., 2022).

A prominent case of such heterogeneous content, of growing importance, is *personal information management* (Halevy et al., 2024). User devices collect vast amounts of personal data on a daily basis. Each interaction with digital services, such as Amazon Marketplace, Netflix or Spotify, is stored for the specific user, and services running on smartphones or smart watches track data without explicit user actions, such as travel routes or workouts. Users want convenient tools to query their data, as easy as with an LLM-powered AI co-pilot. At the same time, *data protection* is a crucial requirement, as full control over sensitive content is a top priority for users (Mansour et al., 2016; Kalokyri et al., 2018). Ideally, all data and its processing resides locally on the owner's devices (mobile, tablet or PC), tightly limiting the available computational resources. Reconciling these goals is a challenge (Halevy et al., 2024).


Question q^3 *„How often did I eat Italian food after playing football?"*

Calendar

18:00, 3 October 2024
Football practice

21:00, 3 October 2024
Dinner with friends
📍 Ristorante Napoli

20:30, 11 October 2024
Pizzaaa time!!

Workouts

19:02 – 20:43
11 October 2024
Workout: football
Duration: 1:40:46
Calories: 1,145 kcal
Ø-Heart rate: 146 bpm

Social media

09:47, 12 October 2024
Had an incredible time with my family yesterday evening. Finally inaugurated the pizza oven! 🍕

17:02, 20 October 2024
Kept the three points at home today! 😍⚽

Mails

19:31, 15 October 2024
Dinner on Sunday? *To: Tom*
Hey Tom,
how is it going? I was wondering if you and Anna would like to come over on Sunday evening?
We finally got the new pizza oven up and running – would be great to fire it up together!
Felix and Lisa are coming too, so it should be fun.
Regards, Oliver

Figure 1: Excerpt of relevant user data for the question q^3, stored in *structured* (workouts), *semi-structured* (calendar) and *unstructured* (social media, mails) sources.

Users' needs range from look-ups of events to queries of analytical nature (Tan et al., 2023):

q^1: *Which books did I buy for my friends last Christmas?*
q^2: *When did I last have pizza with both my parents?*
q^3: *How often did I eat Italian food after playing football?*
q^4: *Daily kilometers I ran during business trips?*

Such questions require fine-grained understanding of the user intent, and entail complex operations like *grouping*, *aggregations*, *joins* or matching *temporal conditions*, often with hundreds of relevant information pieces from different sources.

The goal of this work is to devise a QA system for personal data that (i) supports such complex information needs, (ii) is user-friendly and intuitive for lay people, and (iii) has low footprint regarding computation, memory and energy, suitable for running directly on user devices.

Findings of the Association for Computational Linguistics: ACL 2025, pages 18269–18288
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

1.2 State of the Art and its Limitations

Verbalization: State-of-the-art methods for QA over such heterogeneous data would mostly follow the *verbalization* theme. Relevant data is retrieved, linearized into textual form and cast into token sequences (Oğuz et al., 2022; Christmann et al., 2022). These are fed into an LLM (Minaee et al., 2024) for generating answers, in the spirit of **retrieval-augmented generation (RAG)** (Thorne et al., 2021; Liu et al., 2022; Badaro et al., 2023).

Verbalization has been successfully applied for heterogeneous QA when few evidence pieces are sufficient for answering. However, verbalization falls short as relevant user data often exceeds the context window of LLMs, going well beyond 100K tokens (e.g., for q^3 and q^4). Even if their context size was unlimited, LLMs struggle with more complex operations over hundreds of events, such as aggregation or grouping. This limitation is amplified for numerical data, which is encoded as simple tokens, hindering precise aggregation.

Translation: The predominant theme for such complex questions, with large-scale aggregation and joins, is *translation*. An LLM is provided with the natural language question and the data schema (i.e., headers of tables), in order to generate code in the form of a logical query, typically for languages like SQL or SPARQL (e.g., Fan et al. (2024); Li et al. (2024); Liu et al. (2024b)), which is then directly executed. This **code generation** (**CODEGEN**) is often referred to as NL2SQL or Text2SQL (Li et al., 2024; Liu et al., 2024b).

Translation works well when all data appears in a structured manner, with a well-designed data schema, such that a single query can access all relevant information jointly. In our setting, this is not the case, as exemplified in Figure 1. Relevant user data naturally appears in text, which is not easily accessible by queries. Processing such unstructured text is a key requirement in the context of QA over personal data (e.g., for q^2, q^3, and q^4).

Consider question q^3 as our running example, for which Figure 1 shows a small excerpt of relevant user data. Relevant information, such as playing football, could be stored in different forms, including *structured sources* like workout tables, *semi-structured sources* such as calendar entries, or *unstructured sources* such as social media posts. Moreover, the very same event is often captured by multiple sources (e.g., calendar entry on 11 October and social media post on 12 October), which could lead to double-counting. Finally, cues are often subtle (e.g., the football icon in the second social media post), which means that simple matching of surface text is not sufficient.

1.3 Approach and Contributions

To overcome these limitations, the key idea of this work is to bridge the gap between verbalization and translation, and combine the strengths of both paradigms. Our approach creates an *operator tree*, similar to a query in the translation paradigm, for which we propose judiciously designed operators that enable access to unstructured text, based on ideas of the verbalization paradigm.

Specifically, we present a novel methodology for the task of QA over personal data, REQAP (Recursive Question Understanding for Complex Question Answering over Personal Data). All personal data sources are treated as lists of events described by key-value pairs. For a given user question, REQAP derives a database-style operator tree. Notably, in order to generate these operator trees, we propose a new mechanism for recursively *decomposing* the complexity of questions, via in-context learning and model distillation of small-scale LMs. This enables deployment of the entire QA system on end user devices. Apart from standard SQL-like operators, we introduce two novel operators, `RETRIEVE` and `EXTRACT`, that enable efficient retrieval from both structured and unstructured data sources, and enhancing unstructured text with question-related key-value pairs, via on-the-fly information extraction. These operator trees are executed for deriving the traceable answer.

For evaluation, we construct a new benchmark, PERQA, which synthesizes realistic user data and questions based on personas, and has 3,500 complex questions and more than 40,000 events per persona. Experiments on PERQA demonstrate substantial improvements of REQAP compared to verbalization or translation baselines.

Salient contributions of this work are[1]:

- a novel approach, REQAP, for QA over heterogeneous personal data, based on a recursive question understanding mechanism;
- a complete QA system that can run on end user devices, trained via LLMs and their in-context learning capabilities;
- a new large-scale benchmark for personal QA, PERQA, with realistic data and questions.

---

[1] Code and data available at https://reqap.mpi-inf.mpg.de.

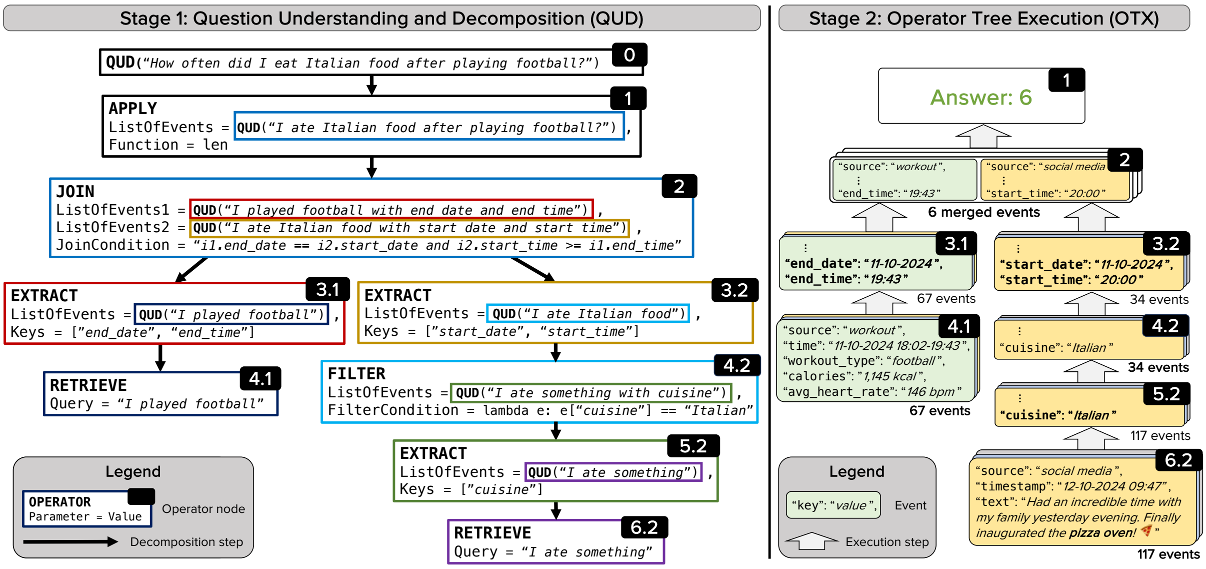

Stage 1: Question Understanding and Decomposition (QUD)
QUD("How often did I eat Italian food after playing football?")
0
APPLY
ListOfEvents = QUD("I ate Italian food after playing football?"),
Function = len
1
JOIN
ListOfEvents1 = QUD("I played football with end date and end time"),
ListOfEvents2 = QUD("I ate Italian food with start date and start time"),
JoinCondition = "i1.end_date == i2.start_date and i2.start_time >= i1.end_time"
2
EXTRACT
ListOfEvents = QUD("I played football"),
Keys = ["end_date", "end_time"]
3.1
EXTRACT
ListOfEvents = QUD("I ate Italian food"),
Keys = ["start_date", "start_time"]
3.2
RETRIEVE
Query = "I played football"
4.1
FILTER
ListOfEvents = QUD("I ate something with cuisine"),
FilterCondition = lambda e: e["cuisine"] == "Italian"
4.2
EXTRACT
ListOfEvents = QUD("I ate something"),
Keys = ["cuisine"]
5.2
RETRIEVE
Query = "I ate something"
6.2
Legend
OPERATOR
Parameter = Value
Operator node
Decomposition step
Stage 2: Operator Tree Execution (OTX)
Answer: 6
1
"source": "workout",
"end_time": "19:43"
"source": "social media
"start_time": "20:00"
2
6 merged events
"end_date": "11-10-2024",
"end_time": "19:43"
3.1
67 events
"start_date": "11-10-2024",
"start_time": "20:00"
3.2
34 events
"source": "workout",
"time": "11-10-2024 18:02-19:43",
"workout_type": "football",
"calories": "1,145 kcal",
"avg_heart_rate": "146 bpm"
4.1
67 events
"cuisine": "Italian"
4.2
34 events
"cuisine": "Italian"
5.2
117 events
"source": "social media",
"timestamp": "12-10-2024 09:47",
"text": "Had an incredible time with my family yesterday evening. Finally inaugurated the pizza oven! 🍕"
6.2
117 events
Legend
"key": "value",
Event
Execution step

Figure 2: Example inference of REQAP. The left side illustrates the output of the QUD stage, an operator tree, for the example question q^3. The right side shows how this tree is executed within the OTX stage to derive the answer.

2 The REQAP Data Model

We treat all data sources as temporally ordered lists of **events**, where each event is a dictionary with a set of **key-value pairs**. For example, each mail is an event with keys `date`, `time`, `sender`, `recipients`, `subject` and `body`; for music streams, each event would have keys like `date`, `start_time`, `end_time`, `title`, `artists`. Example events are shown in the bottom right of Figure 2 (workout in light green and social media in yellow).

Our current implementation covers the following event types, accessed via data exports: *calendar entries*, *notes*, *mails* and *social media posts* with semi-structured and unstructured data, and *workouts*, streams (*movies*, *TV series*, *music*), and *online purchases* with structured data. Note that event types can be added easily with this agile data model, which treats all contents as key-value pairs.

3 The REQAP Methodology

REQAP processes questions in two stages:
(i) the **question understanding and decomposition (QUD)** for constructing the operator tree, and (ii) the **operator tree execution (OTX)** stage for deriving the answer with the corresponding events. Figure 2 shows a characteristic example that illustrates the two stages of REQAP.

3.1 Question Understanding&Decomposition

Inference via ICL. Due to the absence of appropriate training data, we leverage *in-context learning* (ICL) (Brown et al., 2020) to generate our operator trees. We construct handcrafted operator trees for a diverse set of 40 questions, and then *dynamically* retrieve 8 ICL examples at each step via BM25.

In principle, we could prompt the LLM to generate a complete operator tree in a single shot, following the code generation line of works (Li et al., 2024). However, this approach is often too complex, even for an LLM, with many output trees being incomplete or wrong. Instead, we devise a novel method for *recursive decomposition*, where we call the LLM repeatedly to generate *partial operator trees* that have the freedom to resolve sub-questions of the user's input in *recursive* QUD calls. These sub-questions are then the input to subsequent LLM calls, in order to refine the partial operator tree. The recursion stops when all leaf nodes are properly mapped to operators without any open QUD calls.

Example: Consider the operator tree in Figure 2.
(i) The first LLM call generates node 1, with an *aggregate function* (list length, i.e., counting) applied to the question results.
(ii) This is refined into node 2, with the *join* over two *sub-questions*: *"I played football"* and *"I ate Italian food"*.
(iii) Each of two sub-questions is translated into an `EXTRACT` operator, 3.1 and 3.2, and the left branch invokes the leaf-node `RETRIEVE` operator 4.1.
(iv) The right branch 3.2 is further refined, leading to a cascade of `FILTER` (4.2), `EXTRACT` (5.2) and `RETRIEVE` (6.2) operators.
(v) This operator tree is then executed bottom-up, as visualized on the right side of Figure 2.

`RETRIEVE` (Query, Optional: ListOfEvents)
`EXTRACT` (ListOfEvents, Keys)

`JOIN` (ListOfEvents1, ListOfEvents2, JoinCondition)
`GROUP_BY` (ListOfEvents, ListOfKeys)

`FILTER` (ListOfEvents, FilterCondition)
`MAP` (ListOfEvents, Function)
`APPLY` (ListOfEvents, Function)
`UNNEST` (ListOfEvents, KeyToUnnest)

`ARGMIN`, **`ARGMAX`**, **`MIN`**, **`MAX`**, **`SUM`**, **`AVG`**

Table 1: The REQAP operators with their input types.

To facilitate this method, our ICL examples for the LLM follow the same approach, showing how a complex question can be split into an operator part and a simpler sub-question. Thus, each few-shot sample is a sequence of decomposition steps.

Training. Recall that we aim to deploy REQAP on the users' personal devices directly, including mobiles. This would be impossible with the large models ($\geq$70B parameters) used above for ICL.

The solution is to harness the (question, operator-tree) pairs obtained via ICL to *distill* (Hsieh et al., 2023) the LLM into a smaller model (e.g., LLaMA with 1B parameters). To ensure high-quality training samples, we generate multiple operator trees for each training question, evaluate them on the data, and keep only those that return correct results.

Inference via SFT. These *supervised fine-tuned* (SFT) models are deployed for generating operator trees from user questions, and follow the same decomposition procedure as with ICL. In our experiments, we report on different QUD model sizes.

3.2 Operator Tree Execution

The OTX stage executes the operator tree obtained in the QUD stage to derive the answer. Table 1 provides the full list of operators. The following describes the two most important ones: `RETRIEVE` and `EXTRACT`. Other operators and support functions (e.g., `APPLY` for invoking arbitrary Python functions, or `UNNEST` for enumerating elements in a list), are defined in Appendix A.2.

3.2.1 `RETRIEVE` Operator

Functionality. The `RETRIEVE` operator takes as input a *query* generated by the QUD stage. The output consists of *all* events from the personal data sources that soft-match the query, via matching against all *verbalized* key-value pairs.

Challenges. Unlike standard RAG settings, the `RETRIEVE` operator faces specific challenges for analytic queries. Many of these aggregate over hundreds of relevant events, and missing a fraction of these would already distort the final answer. Therefore, we need to ensure that this operator has *high recall*, ideally near 100%, without adding too many false positives. Moreover, the qualifying events identified by `RETRIEVE` may have redundancy, as the same event can be mentioned in calendar, workout, social media, email etc. Therefore, de-duplication of redundant entries is crucial.

Execution. The `RETRIEVE` operator conceptually scans *all* personal data sources and can find matches among all their key-value pairs. However, our method aims to reduce the computational costs by early pruning of entire sources that do not contribute. This goal is achieved by executing the operator in a pipeline with five steps:

Step 1: We use the efficient *sparse retriever SPLADE* (Formal et al., 2022) to obtain an initial pool of candidate events, aiming at high recall (i.e., minimizing the risk of missing a good result). All events with a non-zero score >0.1 are retained.

Step 2: The resulting pool of events is analyzed to identify *patterns*: key-value pairs that occur frequently (i.e., above a specified frequency threshold) in the candidate pool; for example, `subject=`*"Team meeting"* in calendar or `type=`*"movie"*. In addition, for each source with matching events, we add a specific pattern (e.g., `source=`*"mail"*), to score the entire source at once.

Step 3: Finally, the patterns, each representing a set of events, are classified into three categories: (a) relevant (all events are kept), or (b) completely irrelevant (all events are dropped), or (c) partially relevant (events further filtered by key-value pairs).

Step 4: A second classifier runs on all remaining events in (c). It labels key-value pairs into two bins: (a) keep (for downstream operators) or (b) drop.

Step 5: Finally, to tame the potential redundancy of qualifying events (e.g., a calendar entry and a mail on the same meeting), we perform de-duplication. Whenever two events have overlapping time intervals, we combine them into a single event with all key-value pairs and the union of the timespans.

The pruning of entire sources in step 3 (e.g., music streams for queries on food or friends) yields a massive speed-up, and early elimination of frequent key-value pairs also contributes to the efficiency of REQAP (e.g., disregarding all workouts with `type=`*"gym"* for queries on football). At the same time, to ensure high recall, our pipeline is designed to avoid unduly dismissing relevant events.

Training. For steps 3 and 4, the two classifiers are implemented as *cross-encoders* (CE) (Déjean et al., 2024; Lin et al., 2021) for high precision. Our experiments use a standard CE[2], fine-tuned with examples for pattern and event classification.

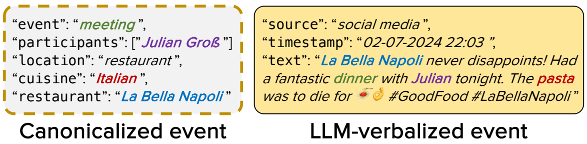
We derive training data from our dataset: for each question, we identify relevant events as the ones used by the ground-truth query (as present in the PERQA dataset). For each `RETRIEVE` call in the operator tree, we would then add these events as positive training instances, and uniformly sample hard negatives from events retrieved by SPLADE. Patterns are labeled based on all matching events, as either (a) fully relevant, (b) fully irrelevant, or (c) partially relevant.

3.2.2 `EXTRACT` Operator

Functionality. The `EXTRACT` operator takes as input a set of *keys*, like attributes/field names, and a list of *events*, typically returned by an upstream `RETRIEVE` operator (i.e., child node in the operator tree). The operator generates values for all combinations of events and keys in the input, and augments events with these new key-value pairs.

Challenges. The `EXTRACT` operator faces some very difficult cases when the input key does not semantically match any of the data keys (beyond simple cases such as matching *"day"* to `date` in calendar or workout data). The challenges arise when the operator's input key requires *extracting* information from a longer text field, such as the body of a mail or the content of a social media post. Example keys would be *"participants"*, *"trip destination"*, or *"cuisine"* – none of which equals any existing data key. Note that the number of input events can often be in the order of hundreds or thousands, necessitating an efficient implementation.

Execution. Hence, we utilize a small-scale *sequence-to-sequence model*, which is given (i) the input key, (ii) the *verbalized* key-value pairs, and (iii) notable user information (e.g., a list of friends) as input, and generates a short phrase as output. For the running example about Italian food after football (question q^3 in Figure 1), the input key is *"cuisine"* and the value *"Italian"* is generated based on the phrase *"...pizza oven..."* in the mail body. For efficiency, `EXTRACT` creates a frozen mapping for simpler input keys (e.g., *"date"*→`start_date`), covering cases where at least 70% of the first 50 inputs lead to the very same event-key (`start_date`). For the remaining input events, this mapping is then applied directly, avoiding further generation.



```
"event": "meeting",
"participants": ["Julian Groß"]
"location": "restaurant",
"cuisine": "Italian",
"restaurant": "La Bella Napoli"
```

Canonicalized event

```
"source": "social media",
"timestamp": "02-07-2024 22:03 ",
"text": "La Bella Napoli never disappoints! Had
a fantastic dinner with Julian tonight. The pasta
was to die for 🍝👌 #GoodFood #LaBellaNapoli"
```

LLM-verbalized event

Figure 3: Example canonicalized event (not accessible at inference), with the corresponding verbalized version.

Training. To enable this rich behavior, we fine-tune a small LM, like BART (Lewis et al., 2020).

We derive high-quality training data from our dataset, which provides parallel data of canonicalized events and their verbalized counter-parts. Figure 3 shows an example of a canonicalized entry representing a meeting, and a corresponding social media post that is generated via an LLM.

4 The PERQA Dataset

4.1 Prior Benchmarks

There is a wealth of established benchmarks for all kinds of QA (Rogers et al., 2023). However, almost all of these are centered on objective contents about real-world facts, and thus unsuitable for our setting. QA over personal data needs to consider subjective events including personally sensitive details. Such data is subject to privacy laws, and this explains the lack of an appropriate benchmark.

To the best of our knowledge, the only existing data source that is suitable for our task is TimelineQA (Tan et al., 2023), which provides code for generating persona-based contents and questions. It allows generating personal data at large scale. However, the specified attributes for synthetic personas are basic, and user content is generated based on a limited number of simple templates. Further, complex questions are derived from solely 42 templates (vs. ≈2,000 in PERQA). Thus, TimelineQA exhibits very limited diversity, not suitable to capture real-world challenges in personal QA.

4.2 Benchmark Creation

To bridge this gap in the existing literature, we devise a principled mechanism to create the PERQA dataset. The construction proceeds in three steps: (i) *persona collection*, in which we let humans (co-workers at our research group) craft realistic and detailed biographical information on **fictional persons**, (ii) *user data creation*, which makes use of the persona information (such as their music or

---

[2] cross-encoder/ms-marco-MiniLM-L-12-v2

Category ↓ / Split →	**Train**	**Dev**	**Test**
Personas	12	2	6
Question templates	1,200	170	600
Unique questions	2,571	170	826
Movie streams	1,257	424	1,245
TV series streams	238	148	230
Music streams	33,754	31,616	38,257
Workouts	635	622	822
Online purchases	531	1,074	850
Calendar entries	1,552	1,367	2,106
Mails	506	490	636
Social media posts	670	579	819
Total events (per persona)	39,142	36,322	44,965

Table 2: Statistics for the PerQA dataset.

travel preferences) and creates large amounts of realistic user data, and (iii) *question generation*, that generates complex questions, and their ground-truth answers, for the personas and their user data.

Persona collection. The personas are created by having humans (from our research group) complete questionnaires with 30 fields about age, gender, family, education, work history, traveling, hobbies, fitness activities, interests in music and movies, food tastes and other preferences. We collect 20 such human-crafted personas.

Canonicalized events. The next step is to create large-scale event repositories for each persona. In this stage, events are *canonicalized*: for each event type, events have a specific set of key-value pairs. This allows us to derive ground-truth answers later.

For streaming data (movies, TV series, music), we utilize Wikidata (Vrandečić and Krötzsch, 2014) to derive real-life entities matching the persona preferences (e.g., the favorite genre and songs). Similarly, for online shopping, we utilize a public data dump (Ni et al., 2019a). For realistic travel itineraries in-line with the persona information, we prompt GPT4o. For workouts, we utilize public data from Endomondo (Ni et al., 2019b).

We further add special events, like personal milestones (marriage, birth of a child, new job,...) based on the personas. Finally, we add regular events like doctor appointments, meetings with friends, or New Year's parties. This is done based on pre-defined probability values, inspired by the procedure in Tan et al. (2023).

On average, for each persona, we create 40K such canonicalized events. The left side of Figure 3 provides an example canonicalized event. A list of all event types is provided in Appendix B.

Observable events. In reality, events such as restaurant visits, birthday parties, or a child birth, are usually not captured in a canonicalized form, but in unstructured text as in calendar entries, mails and social media posts. We utilize LLaMA3.1[3] to verbalize such events, and create realistic and diverse user data. Figure 3 illustrates the connection between canonicalized and verbalized events. Streams, workouts and online purchases are kept in a canonicalized form with high probability, but are also verbalized with a relatively small probability.

Generation of complex questions. For generating questions, we handcraft a diverse set of 40 complex questions to start with – the same questions that are used for ICL in the QUD stage. We then iteratively sample up to 20 questions from this set, and prompt an LLM (with a temperature of 0.6) to generate 25 new questions. In addition to the questions, the ICL examples (and the model outputs) also contain SQL queries, run on a database with the canonicalized events, to derive the ground-truth answers. Figure 5 shows the instruction used for creating these question-query pairs. We utilize GPT4o and LLaMA3.1[3] to construct 1,000 questions each, for enhanced diversity. Generated question-query pairs are de-duplicated, and checked for correctness semi-automatically.

Deriving PerQA. We release PerQA with 20 personas, 40K events per persona, and 3,567 unique questions. This data is split along two dimensions, personas and question templates, to derive the train (12 personas, 1,200 templates), dev (2 personas, 170 templates), and test (6 personas, 600 templates) sets. Question templates are initiated based on the personas, to ensure natural and meaningful questions. Table 2 shows statistics of PerQA, and further details are provided in Appendix B, including example questions and data in Tables 8 and 13.

5 Experimental Setup

Configurations. We experiment with three different options for QUD: using **GPT4o** ("gpt-4o") or **LLaMA3.3**[4] with 8 ICL examples, or a 1B model[5] fine-tuned as described in Section 3.1 (**SFT**). To initialize the `EXTRACT` operator, we use BART-base[6], and the `RETRIEVE` operator uses SPLADE[7] and a cross-encoder pre-trained on MSMARCO[8]. Further details for model training are in Appendix C.

---

[3] meta-llama/Llama-3.1-70B-Instruct

[4] meta-llama/Llama-3.3-70B-Instruct

[5] meta-llama/Llama-3.2-1B-Instruct

[6] facebook/bart-base

[7] naver/splade-cocondenser-ensembledistil

[8] cross-encoder/ms-marco-MiniLM-L-12-v2

Baselines. We compare ReQAP against Rag and CodeGen:

- **Rag:** This retrieval-augmented generation follows the *verbalization* paradigm. For retrieving relevant events, a two-stage technique is employed, with SPLADE retrieval followed by a fine-tuned cross-encoder (similar as in 3.2.1).
- **CodeGen:** We also compare against a code generation baseline following the *translation* paradigm. The input is the user question and the SQL schema of the personal data, and the output is a SQL query. For fine-tuning, we utilize SQL queries that lead to the correct answer (similar as in 3.1).

For fair comparison, these baselines use the same model variants as the QUD stage of ReQAP: ICL with 8 examples using GPT4o or LLaMA3.3, or a fine-tuned decoder-only LM.

Metrics. We measure **Hit@1** as our main metric for personal QA, where users expect a single concise answer. As some questions aggregate over hundreds of events, we also measure a relaxed metric **Rlx-Hit@1** which allows a ±10% slack for numeric answers. Both metrics are micro-averaged over all questions in the test set. Statistical significance of improvements by ReQAP compared to the best baseline (for each model) is measured via McNemar's test with $p<0.05$, and is indicated by an asterisk(*).

6 Experimental Results

6.1 Key Findings

ReQAP clearly wins over baselines. Table 3 shows the main results on PerQA. First of all, ReQAP substantially outperforms baselines for each variant. The difference is most pronounced with LLaMA3.3, where ReQAP improves Hit@1 by 8.3 percentage points compared to CodeGen, the best baseline. The same observation holds for Rlx-Hit@1, with similar improvements over baselines.

Rag fails on PerQA. While the baselines based on CodeGen remain competitive, Rag performs poorly. It cannot cope with questions that need to aggregate (sum, avg, ordering, etc.) over hundreds of events. In some cases, the input data exceeds the maximum context size of the LM (128K tokens).

SFT is a viable solution for on-device inference. The variants based on GPT4o and LLaMA3.3 are useful for comparison purposes, but cannot be run directly on user devices. Remarkably, our SFT variant, which is suitable for on-device inference, exhibits competitive performance even with a 100x smaller model size. Its Hit@1 of 0.380 is almost as good as the best observed score of 0.386 (by ReQAP with GPT-4o), and its relaxed Hit@1 of 0.53 is the best across the board.

Model → Method ↓	GPT4o ($\gg$100B)	LLaMA3.3 (70B)	SFT (1B)
Rag	0.149 (0.20)	0.123 (0.18)	0.029 (0.06)
CodeGen	0.319 (0.44)	0.239 (0.33)	0.315 (0.47)
ReQAP (ours)	**0.386*** (**0.52**)*	**0.322*** (**0.46**)*	**0.380*** (**0.53**)*

Table 3: Main results on PerQA ***test*** set. Metric is Hit@1, and Rlx-Hit@1 in parentheses.

Complexity	No. of questions	Rag (GPT4o)	CodeGen (GPT4o)	ReQAP (GPT4o)
Ordering	1,236	0.167	0.440	**0.529**
Grouping	931	0.172	0.444	**0.537**
Temporal	1,945	0.129	0.290	**0.417**
Aggregation	1,790	0.130	0.228	**0.296**
Join	369	0.073	0.176	**0.236**
Multi-source	1,710	0.196	0.237	**0.365**

Table 4: Performance by complexity types of questions.

Approximate answers are good. The questions in PerQA are difficult for all methods, with Hit@1 being below 40%. Nevertheless, ReQAP often yields decent order-of-magnitude answers. An indicator is the Rlx-Hit@1 metric, for which ReQAP achieves 53% with the small SFT variant. Recall that this allows ±10% slack, which is on the conservative side. Even more relaxed approximations could still be tolerable by users.

6.2 Analysis

ReQAP performs well across question types. We categorized questions in PerQA into different complexity types: ordering (e.g., *"Which song did I listen to the most during my last trip?"*), grouping (*"which four restaurants do I go to the most?"*), temporal (*"How many times did I visit the doctor in the last 3 years?"*), aggregation (*"What was the total amount spent on online purchases last year?"*), join (*"How many times did I watch a movie after a run?"*), and multi-source (*"which of my trips was the shortest?"*). Note that each question can be assigned multiple such complexity types.

Results for this analysis are shown in Table 4. ReQAP consistently outperforms baselines, and performs especially well on questions requiring ordering or grouping. Join questions are the most challenging for all methods.

Method ↓ / Metrics →	Hit@1	Rlx-Hit@1
ReQAP (SFT)	**0.396**	**0.54**
w/ QUD – one-shot	0.356	0.50
w/ `RETRIEVE` – SPLADE-only	0.269	0.36
w/ `EXTRACT` – keys-only	0.138	0.23

Table 5: Ablation studies on the PerQA ***dev*** set.

	QUD → **Operators ↓**	**XS** (135M)	**S** (360M)	**M** (1B)	**L** (3B)
XS	`RETRIEVE` (4M) & `EXTRACT` (16M)	0.167 (0.28)	0.185 (0.28)	0.193 (0.31)	0.193 (0.30)
S	`RETRIEVE` (16M) & `EXTRACT` (31M)	0.240 (0.37)	0.287 (0.41)	0.302 (0.42)	0.302 (0.42)
M	`RETRIEVE` (23M) & `EXTRACT` (70M)	0.331 (0.48)	0.353 (0.51)	0.378 (0.50)	0.389 (0.52)
L	`RETRIEVE` (33M) & `EXTRACT` (139M)	0.356 (0.51)	0.364 (0.52)	0.396 (0.54)	0.400 (0.55)

Table 6: Effect of model sizes on performance. Metric is Hit@1, and Rlx-Hit@1 in parentheses.

`RETRIEVE` and `EXTRACT` operators are decisive. Table 5 shows ablation studies for ReQAP, when (i) the operator tree is generated in one shot, *without recursive decomposition*, (ii) all events retrieved by SPLADE are retained, *without cross-encoders*, (iii) values are extracted from events only by exact matches of keys, *without LM generation*.

Performance degrades dramatically for each of these variants. Most notably, the devised implementation of the `RETRIEVE` and `EXTRACT` operators substantially contribute to ReQAP's performance. Also, the recursive decomposition of the question intent strikes as a key asset as generating the operator trees in a single step is prone to errors.

ReQAP adapts to low-resource computation. Our goal is a method that can run on user devices. However, there is high variance in the available compute-power and memory on client hardware, and QUD alone may use cloud services without violating data protection. Hence, we trained models used in ReQAP of varying sizes: from 135M-3B parameters for the QUD model, 4M-33M for the cross-encoders of `RETRIEVE`, and 16M-139M for the `EXTRACT` model. Table 6 shows the results.

While there is a substantial performance gap between the smallest variant (QUD: XS, Operators: XS) and the largest variant (L, L), we observe interesting trade-offs and compromises. When the memory footprint has to be low, but computational costs and run-time are secondary, the (XS, L) variant is a good choice, achieving a Rlx-Hit@1 of 51%. Recall that QUD runs on the question only, without access to user data. When questions alone are not sensitive, a larger cloud model can be used.

Model → **Method ↓**	**GPT4o** ($\gg$100B)	**LLaMA3.3** (70B)	**SFT** (1B)
Rag	0.321 (0.36)	**0.369** (**0.41**)	0.214 (0.22)
CodeGen	0.262 (0.33)	0.187 (0.22)	0.135 (0.17)
ReQAP (ours)	**0.357** (**0.39**)	0.286 (0.31)	**0.313** (**0.35**)

Table 7: Hit@1 and Rlx-Hit@1 on TimelineQA.

Error analysis. We analyzed 50 incorrect answers. ReQAP failed because of (i) an incorrect operator tree (50%), (ii) errors by the `RETRIEVE` operator (28%), or (iii) errors by the `EXTRACT` operator (22%). Note that sometimes just a few false negatives or false positives can lead to an incorrect answer, underlining the difficulty of PerQA.

Additional results, including anecdotal examples and run-time analysis, are provided in Appendix D.

6.3 Experiment with TimelineQA

Setup. Although TimelineQA is deemed insufficient for our setting, we obtained insight on how ReQAP performs on these questions. For this purpose, we used their 42 templates for complex questions, and adapted them to our realistic user data. We ran the questions for each of the 6 test personas in PerQA. SFT variants utilize the models trained on PerQA, as the limited number of questions in TimelineQA does not allow for training.

Results. Table 7 shows the results on TimelineQA. Rag shows stronger performance than on the more challenging questions in PerQA. ReQAP substantially outperforms both Rag and CodeGen, with the small model (SFT) and with GPT4o.

7 Evaluation via User Study

The PerQA benchmark is designed with synthetic personas to avoid privacy issues. The goal of this user study is two-fold: (i) analyze whether the questions in PerQA are comparable with real user questions, and (ii) investigate whether a small-scale variant of ReQAP, run on end-user devices, can deal with such realistic information needs.

7.1 Setup

We conducted our user study with 20 undergraduate students. First, we instructed participants to export their personal data from various services like Apple, Netflix and Spotify. Then, users were provided with Docker images for running ReQAP on their own laptops and exports, but without giving us access to their personal data. The Docker images had disabled Internet connectivity, to guarantee the impossibility of data leaks.

We initially instructed each participant to generate 100 questions, without restricting them to any question type or sources. Users then selected a subset of 20 questions that are answerable based on their personal data, assuming a hypothetical perfect QA system. More details are given in Appendix E.

7.2 Results

User and PERQA questions are similar. We ran our QUD module on user and benchmark questions, and compared the structure of the resulting operator trees, abstracting away the individual constants. For 376 out of 400 (=20 users×20 questions) user questions (94%), we were able to map them to an isomorphic operator tree in PERQA. This indicates that the questions in PERQA indeed capture the complexities of real-life information needs.

REQAP performs well in the wild. For each of their 20 selected questions, users ran REQAP on their own device and data. We asked them whether the provided answer is *correct*, *almost correct* (with minor drifts tolerated by the user), or *incorrect*. By design, REQAP allows users to identify which events were used for answering, aiding them to understand the correctness of answers. Users judged ca. 28% of the answers for their own questions as fully correct, and 41% as almost correct.

Users also evaluated 20 PERQA questions against their personal data, with allowance for minor adjustments to adapt them to their data. In this setting, REQAP achieves even better performance: 45% for Hit@1 (and 60% for Rlx-Hit@).

8 Related Work

Personal data management. This theme was initially motivated by the tasks of desktop search (Dong and Halevy, 2005) and mail/document search (Dumais et al., 2003). With the increasing abundance of personal data on mobile phones and cloud storage, the topic has been revived (Halevy et al., 2024). Recent works target simple retrieval over personal data (Ribeiro et al., 2022; Gurrin et al., 2023). The closest work to ours is TimelineQA (Tan et al., 2023), which conducts experiments with standard RAG and CODEGEN approaches – we show that REQAP substantially outperforms these state-of-the-art baselines.

Orthogonally, there are concerns about platform providers "knowing everything" about users, and the need for humans to stay in full control of their personal data (Mansour et al., 2016; Kalokyri et al., 2018). Our approach follows this rationale.

Data verbalization. A popular way of using LLMs for multi-source QA is to verbalize all data and treat it uniformly as if were a collection of natural language sentences (Oğuz et al., 2022). This approach has been pursued for the case of text+tables (Yin et al., 2020; Badaro et al., 2023), and for combining knowledge graphs with other sources (Christmann et al., 2022, 2023; Pramanik et al., 2024; Zhang et al., 2024).

Question-to-query translation. State-of-the-art methods, often referred to as Text2SQL or NL2SQL (Katsogiannis-Meimarakis and Koutrika, 2023; Liu et al., 2024b), leverage LLM pre-training with code repositories (e.g., GitHub) and fine-tune the model with question-query pairs (Pourreza and Rafiei, 2023; Fan et al., 2024; Li et al., 2024; Gao et al., 2024; Gong and Sun, 2024). Importantly, these methods require *schema* information about the tables, column headers and column types of the underlying sources. For complex schemas or when only partial information is provided, the performance drops sharply.

Question decomposition. By decomposing a user question, REQAP aims to understand the user's intent and derive a suitable representation. This has been pursued for a variety of specialized settings, including Jia et al. (2024) for temporal questions, and Saeed et al. (2024) and Liu et al. (2024a) for delegating sub-tasks to an LLM. Also, enhanced prompting techniques like chain-of-thought (Wei et al., 2022) and tree-of-thoughts (Yao et al., 2023) have been proposed. While these approaches enable LLMs to decompose tasks of certain kinds, they exhibit similar drawbacks as RAG, and do not generalize to QA over heterogeneous data sources.

9 Conclusion

This paper introduced a new approach, REQAP, for QA over heterogeneous personal data, spanning structured tables and unstructured text. REQAP recursively decomposes a user question into an operator tree, which can be efficiently executed on user devices. In our experiments, REQAP outperforms methods based on verbalization (RAG) or translation (CODEGEN), and our small-model configurations achieve competitive performance, thus reconciling the goals of expressive QA and light-weight footprint.

10 Limitations

Our experimental evaluation is based on two benchmarks and a small-scale user study. Obviously, it would be desirable to consider additional datasets and scale up the user study. Neither of these is easily feasible. Collecting data and questions from real users is highly sensitive. Even with anonymization techniques, adversaries that have background knowledge (e.g., co-workers of participating users) may obtain privacy-critical information. This is why we did not pursue this approach.

On the technical side, the REQAP system supports a limited set of data repositories (calendar, mail, music streaming etc.). Extending this would be straightforward as long as a service can cast its data into our data model of events with key-value pairs. A more important restriction is the system's assumption that the data is materialized, via data exports from the user's accounts. This is non-critical with synthetic personas or in a controlled user study, but would be a concern on behalf of real users. Ideally, REQAP would tap into the user's data only through APIs of the underlying services, giving rise to QA over a federation of autonomous and encapsulated services. While this is unrealistic with the current setup of mobile phones, pursuing such a federated architecture would be important for future research on reconciling user convenience and data protection.

11 Ethical Considerations

Our benchmark construction intentionally avoided gathering data from real users, as this would be extremely difficult to properly anonymize. For adversaries with background knowledge about a user, even anonymized data would not perfectly prevent privacy breaches.

The user study that we carried out with a small cohort had all participants sign an explicit consent form, and has been approved by the Ethics Review Board. Moreover and most importantly, it was carried out in a way where the users had full control over each and every step. All their data was exported from their service without leaving their personal devices. The software for running REQAP was packaged into Docker images, with Internet connection switched off. Only the user-created questions and relevance labels for answers were given to us, and the users had the option of removing questions, or removing names by placeholders, if deemed too sensitive.

All code and data of this work is publicly accessible at https://reqap.mpi-inf.mpg.de for reproducibility and support of follow-up research.

Acknowledgements

We would like to thank Rishiraj Saha Roy, Sneha Singhania, and Magdalena Kaiser for their helpful feedback on this paper. We also thank the anonymous reviewers for their constructive comments.

References

Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. 2023. Transformers for Tabular Data Representation: A Survey of Models and Applications. *Transactions of the Association for Computational Linguistics*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS 2020)*.

Philipp Christmann, Rishiraj Saha Roy, and Gerhard Weikum. 2022. Conversational Question Answering on Heterogeneous Sources. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2022)*.

Philipp Christmann, Rishiraj Saha Roy, and Gerhard Weikum. 2023. Explainable Conversational Question Answering over Heterogeneous Sources via Iterative Graph Neural Networks. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2023)*.

Hervé Déjean, Stéphane Clinchant, and Thibault Formal. 2024. A Thorough Comparison of Cross-Encoders and LLMs for Reranking SPLADE. *arXiv*.

Xin Luna Dong and Alon Y. Halevy. 2005. A Platform for Personal Information Management and Integration. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*.

Susan T. Dumais, Edward Cutrell, Jonathan J. Cadiz, Gavin Jancke, Raman Sarin, and Daniel C. Robbins. 2003. Stuff I've Seen: A System for Personal Information Retrieval and Re-Use. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2003)*.

Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining Small Language Models and Large Language Models for Zero-Shot NL2SQL. *Proceedings of the VLDB Endowment (VLDB 2024)*.

Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2022. From Distillation to Hard Negative Sampling: Making Sparse Neural IR Models More Effective. In *The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2022)*.

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. *Proceedings of the VLDB Endowment (VLDB 2024)*.

Zheng Gong and Ying Sun. 2024. Graph Reasoning Enhanced Language Models for Text-to-SQL. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2024)*.

Cathal Gurrin, Björn Þór Jónsson, Duc-Tien Dang-Nguyen, Graham Healy, Jakub Lokoc, Liting Zhou, Luca Rossetto, Minh-Triet Tran, Wolfgang Hürst, Werner Bailer, and Klaus Schoeffmann. 2023. Introduction to the Sixth Annual Lifelog Search Challenge, LSC'23. In *Proceedings of the 2023 ACM International Conference on Multimedia Retrieval (ICMR 2023)*.

Alon Halevy, Yuliang Li, and Wang-Chiew Tan. 2024. Personal Manifold: Management of Personal Data in the Age of Large Language Models. In *IEEE 40th International Conference on Data Engineering (ICDE 2024)*.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*.

Zhen Jia, Philipp Christmann, and Gerhard Weikum. 2024. Faithful Temporal Question Answering over Heterogeneous Sources. In *Proceedings of the ACM on Web Conference 2024 (WWW 2024)*.

Varvara Kalokyri, Alexander Borgida, and Amélie Marian. 2018. YourDigitalSelf: A Personal Digital Trace Integration Tool. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM 2018)*.

George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A Survey on Deep Learning Approaches for Text-to-SQL. *The VLDB Journal*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL 2020)*.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proceedings of the VLDB Endowment (VLDB 2024)*.

Jimmy Lin, Rodrigo Frassetto Nogueira, and Andrew Yates. 2021. *Pretrained Transformers for Text Ranking: BERT and Beyond*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: Table Pre-training via Learning a Neural SQL Executor. In *International Conference on Learning Representations (ICLR 2022)*.

Shicheng Liu, Jialiang Xu, Wesley Tjangnaka, Sina Semnani, Chen Yu, and Monica Lam. 2024a. SUQL: Conversational Search over Structured and Unstructured Data with Large Language Models. In *Findings of the Association for Computational Linguistics: NAACL 2024*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024b. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *arXiv*.

Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. A Demonstration of the Solid Platform for Social Web Applications. In *Proceedings of the 25th International Conference on World Wide Web (WWW 2016)*.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large Language Models: A Survey. *arXiv*.

Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019a. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Jianmo Ni, Larry Muhlstein, and Julian McAuley. 2019b. Modeling Heart Rate and Activity Data for Personalized Fitness Recommendation. In *Proceedings of the World Wide Web Conference (WWW 2019)*.

Barlas Oğuz, Xilun Chen, Vladimir Karpukhin, Stan Peshterliev, Dmytro Okhonko, Michael Sejr Schlichtkrull, Sonal Gupta, Yashar Mehdad, and Scott Yih. 2022. UniK-QA: Unified Representations of Structured and Unstructured Knowledge for Open-Domain Question Answering. In *Findings of the Association for Computational Linguistics: NAACL 2022*.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Advances in Neural Information Processing Systems (NeurIPS 2023)*.

Soumajit Pramanik, Jesujoba Alabi, Rishiraj Saha Roy, and Gerhard Weikum. 2024. UNIQORN: Unified Question Answering over RDF Knowledge Graphs and Natural Language Text. *Journal of Web Semantics*.

Ricardo Ribeiro, Alina Trifan, and António JR Neves. 2022. Lifelog Retrieval From Daily Digital Data: Narrative Review. *JMIR mHealth and uHealth*.

Anna Rogers, Matt Gardner, and Isabelle Augenstein. 2023. QA Dataset Explosion: A Taxonomy of NLP Resources for Question Answering and Reading Comprehension. *ACM Computing Surveys*.

Mohammed Saeed, Nicola De Cao, and Paolo Papotti. 2024. Querying Large Language Models with SQL. In *Proceedings 27th International Conference on Extending Database Technology (EDBT 2024)*.

Wang-Chiew Tan, Jane Dwivedi-Yu, Yuliang Li, Lambert Mathias, Marzieh Saeidi, Jing Nathan Yan, and Alon Y. Halevy. 2023. TimelineQA: A Benchmark for Question Answering over Timelines. In *Findings of the Association for Computational Linguistics: ACL 2023*.

James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Y. Levy. 2021. From Natural Language Processing to Neural Databases. *Proceedings of the VLDB Endowment (VLDB 2021)*.

Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowledge Base. *CACM*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP 2021)*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in neural information processing systems (NeurIPS 2022)*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *Advances in neural information processing systems (NeurIPS 2023)*.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL 2020)*.

Heidi C. Zhang, Sina J. Semnani, Farhad Ghassemi, Jialiang Xu, Shicheng Liu, and Monica S. Lam. 2024. SPAGHETTI: Open-Domain Question Answering from Heterogeneous Data Sources with Retrieval and Semantic Parsing. In *Findings of the Association for Computational Linguistics (ACL 2024)*.

Appendices

A Details on the REQAP Methodology

A.1 Question Understanding&Decomposition

Figure 4 shows the instruction used for the LLM-version of the QUD component. The in-context learning (ICL) samples are judiciously crafted to bring out key difficulties that the model has to deal with, while covering all OTX operators. We hand-craft a total of 40 such ICL examples, based on a diverse set of 40 questions. Table 15 shows 5 such examples. Each ICL example consists of multiple user and assistant turns, enabling the decomposition of questions.

During LLM-inference for a given sub-question, we retain the previous input sub-questions and output operators for context.

A.2 REQAP Operators

This section provides additional detail on the operators currently in-use within REQAP.

`JOIN`. The `JOIN` operator has the following inputs and outputs:

- **Input:** The input comprises *two lists of events*, computed as intermediate results for two sub-questions by the upstream operators (descendants in the tree), and a *join condition*: a comparison between key-value pairs of the two input lists.
- **Output:** The result is a set of *combined events* (the two sets of key-value pairs are merged) that contains all event pairs from the two input lists that satisfy the join condition.

Unlike joins in mainstream multi-hop QA, where all joins are equi-joins (often just link traversal in graphs), our setting requires support for non-equi joins, most notably, for temporal comparisons with before/after and overlap of time intervals.

Our method for this general join processing is a sort-merge algorithm. The two sets of events to be joined are sorted by the keys that the join predicate refers to, and the join result is obtained by scanning and merging the two ordered streams.

`GROUP_BY`. The `GROUP_BY` operator has the following inputs and outputs:

- **Input:** a list of events and a list of keys in this data, by which the events are grouped.
- **Output:** a partitioning of the events, represented as a list of lists. Each partition (i.e., one of the inner lists) consists of all events that share the same values for the input keys.

For the implementation, we adopted a hash-based algorithm that maps events into a hash table by their values of the grouping keys. The simplest case is a single key for grouping, but grouping by multiple keys can often be required as well: an example is for questions like *"in which month did I run the most?"*, which logically require grouping by month and year.

`FILTER`. The `FILTER` operator has the following inputs and outputs:

- **Input:** a set of events and a filter condition which is implemented as a function that can refer to key-value pairs (e.g., `cuisine=`*"Italian"*).
- **Output:** all events that satisfy the filter condition.

The implementation supports arbitrary filter conditions expressed as Python code, including exact matching, numeric matches, temporal comparisons, or sub-string matches. Within the `FILTER` operator, calls to the QUD module are allowed as well, enabling a flexible framework for various situations.

`MAP`. The `MAP` operator has the following inputs and outputs:

- **Input:** a list of events and a Python function.
- **Output:** a list of return values of the function applied to each element in the list (e.g., the weekday for each event).

This operator is key to plugging in arbitrary functions (similar to user-defined functions (UDFs) in SQL) into the execution plan. This is frequently needed, for example, for calendar-centric calculations, such as the weekday of an event or the duration of the period between dates. By enabling such plug-in functions, the REQAP system is easily extensible.

`APPLY`. The `APPLY` operator has the following inputs and outputs:

- **Input:** a list of events and a Python function.
- **Output:** a scalar value computed over the entire list.

One role of `APPLY` is to compute aggregation functions, such as the length of a list (for counting), summing up numeric values, etc.

`UNNEST`. The SQL-inspired `UNNEST` operator has the following inputs and outputs:

- **Input:** a list of events, and a key pointing to the nested lists.
- **Output:** an extended list of events, which holds an independent event for each item in the nested

```
You are given a question and need to generate a plan to infer the answer.
This plan follows Python syntax. You can make use of the following functions:
‘QUD (question: str)‘
    Used to make a recursive call to the QUD module.

‘RETRIEVE (query: str) -> QUExecutionResult‘
    Retrieves all events that match the query.
‘EXTRACT(l: QUExecutionResult, attr_names: List[str], attr_types: List) -> QUExecutionResult‘
    Extracts the values for the provided attributes from each event.
‘GROUP_BY(l: QUExecutionResult, attr_names: List[str]) -> QUExecutionResult‘
    Function to group the results by the provided attributes.
‘JOIN(l1: QUExecutionResult, l2: QUExecutionResult, condition: str) -> QUExecutionResult‘
    Function to join two results based on the condition. Condition is a string, and makes use of i1 and i2 to refer
    to items in the list.
‘APPLY(l: QUExecutionResult, fct: Callable) -> QUExecutionResult‘
    Apply a callable on the list of events in l.
‘MAP(l: QUExecutionResult, fct: Callable, res_name: Optional[str]="map_result") -> QUExecutionResult‘
    Map a callable to each event or group (GROUP_BY result) in l, and store each result in the key ‘res_name‘.
‘FILTER(l: QUExecutionResult, filter: Callable) -> QUExecutionResult‘
    Filter the events using the provided callable.
‘UNNEST(l: QUExecutionResult, nested_attr_name: str, unnested_attr_name: str) -> QUExecutionResult‘
    Derives a single event for each value in the list in key ‘nested_attr_name‘, with the value being stored in
    ‘unnested_attr_name‘.

‘ARGMIN / ARGMAX (l: QUExecutionResult, arg_attr_name: str, val_attr_name: Optional[str]) -> QUExecutionResult‘
    Computes the argmin/argmax based on the provided attribute (‘arg_attr_name‘), and returns the corresponding
    value (‘val_attr_name‘).
‘SUM / AVG / MAX / MIN (l: QUExecutionResult, attr_name: str) -> QUExecutionResult‘
    Computes the sum/avg/max/min based on the provided attribute (‘attr_name‘).

‘QUExecutionResult‘ is a special class that is used to store the results of functions.

Whenever accessing an attribute in a function call, make sure that this attribute is derived in a subsequent call via an
appropriate call (e.g., EXTRACT or MAP). For example, when val_attr_name="workout_type" is used in a ARGMAX call, there has
to be a EXTRACT call with attr_names=["workout_type",...] in the following.

Please provide only the next step, including the recursive QUD call to take (if needed).
Learn from the examples below.
```

Figure 4: LLM instruction for generating operator trees. This is followed by 8 in-context learning examples, which are dynamically chosen per step. Sample ICL examples are shown in Table 15.

lists, flattening this key-value pair.

An example use-case of `UNNEST` is a user question like *“Artist I listened to the most?”*, that requires grouping by individual artist names, while the events themselves have a key `artists` holding multiple names. `UNNEST` creates an independent event for each artist name in the list, enabling to answer such questions correctly.

The implementation of `APPLY`, `MAP` and `UNNEST` is straightforward. The remaining operators (`ARGMIN`, `ARGMAX`, `MIN`, `MAX`, `SUM`, and `AVG`) are self-explanatory.

B Details on the PERQA Benchmark

B.1 Details on the PERQA Construction

Persona details. For each fictional persona, the following information is collected: *name*, *gender*, *birth date/city*, *mother*, *father*, *siblings*, *kids* (incl. birth date), *pets* (incl. type, start/end date), *friends*, *education* (incl. city, start/end date), *career* (incl. company, city, start/end date), *residences* (incl. start/end date). Further, we collect user preferences: *favorite music genres*, *movie genres*, *TV series genres*, *shopping categories*, *travel regions*, *cuisines*, *hobbies* and *workouts*, as well as favorite instances (e.g., favorite songs) and the average frequency (e.g., average movies per week).

Canonicalized event types. PERQA integrates the following canonicalized event types: *anniversaries* (birthday, child’s birthday, partner anniversary, valentines day, halloween, christmas, New Year’s Eve, pet anniversary), *doctor appointments* (dentist, gp, ophthalmologist, dermatologist, paediatrician, veterinarian), *personal milestones* (new job, start of new education, new residence, marriage, new kid, new pet), *traveling* (incl. 2-5 notable events), *meetings* (with friends and family), *streaming* (music, TV series, movies), *online shopping* and *workouts*.

Natural user behavior. A key criterion when creating user data is naturalness. Hence, we aim to derive sequences of events that are as natural as possible. Consider the music streaming behavior as an example: it is very unlikely that someone listens to a single song at 2AM and then to another song at 5AM. We thus base the sequences of events on real user behavior patterns. Such user behavior patterns are derived from streaming data of the authors and colleagues (only the patterns, no critical data is used). These patterns are then used to define the sequence of events for streaming data, workouts and online shopping.

Cleaning of unnatural questions. The generation process may still yield some questions that are misinformed for specific personas. For example, there could be question templates asking for the birth date of the first-born child, but not all personas have children. Such *empty-answer* questions (no output, or a count of 0 for aggregations), are unnatural which is why we removed them for the respective personas.

Quality check. Data quality was essential for this work. To this end, we performed the user study (Section 7) for exactly this purpose: comparing the benchmark data and questions against those from real users, obtaining additional insights from real users, and this way re-assessing the significance of our main results. Section 7.2, first paragraph, compares the structure of questions from these two settings: synthetic vs. real users. We observe that a very large fraction of the user's questions (94%) have structurally isomorphic counterparts in the PerQA dataset. Obviously, the literals between these cases are not comparable (e.g., real users' friends would have names different from the ones in PerQA). But this comparison generally shows that the real-user questions are structurally very similar to those in PerQA.

Apart from the results derived from the user study, we annotated 100 randomly sampled questions in PerQA. We found that 98% of questions are relevant. We further annotated the ground-truth answers for these 100 sample questions. We observed that 97% of the answers are correct, with minor issues otherwise. Finally, we sampled 100 user events in PerQA, and investigated whether the events are realistic user events, and whether the LLM-verbalizations (mails, calendar entries and social media posts) are semantically equivalent. We found that 98% of events do indeed express realistic user events, and LLM-verbalizations are semantically equivalent in all cases.

Usage of LLaMA-70B. The version originally submitted to ACL rolling review was using LLaMA-8B[9] for verbalizing the canonicalized events in PerQA. For enhanced diversity, and even more realistic data, we utilized LLaMA-70B[10] instead in the revised version of this work. This explains slight derivations of results from the submitted version of the paper – generally, through the more diverse data, results of ReQAP improved.

Questions – Workouts
- *"What is the total distance I ran in kilometers since I started working at Shopify?"*
- *"My maximum heart rate during the most recent workout?"*
- *"How many km did I run in the first week of 2019?"*
- *"How many workouts did I do in 2018 with a maximum heart rate of more than 170?"*
Questions – Music
- *"Which song did I listen to the most in the last month?"*
- *"What is the name of the song I listened to the most in the summer of 2022?"*
- *"Which song did I listen to most after a badminton workout?"*
- *"which artist did I listen to most?"*
Questions – Movies / TV series
- *"how many different movies did I watch?"*
- *"the TV series I watched the most in summer 2022?"*
- *"What's the total number of hours spent on TV series?"*
- *"Which day of the week do I stream TV series the most?"*
Questions – Mail / social media
- *"Who was the recipient of the first email I sent this year?"*
- *"Who are my top 3 most contacted people via email?"*
- *"How many emails did I send after 6 PM?"*
- *"What was the last social media post I made?"*
Questions – Others
- *"How many times did I meet with Carla Díaz at a restaurant?"*
- *"What day of the week do I most frequently schedule doctor appointments?"*
- *"What was the total amount spent on online purchases last year?"*
- *"How many times did I meet at a park in 2022?"*

Table 8: Example questions in PerQA.

License. The PerQA dataset is released with a CC BY 4.0 license to enable others to work with our data and advance the state-of-the-art.

Data demographics. All data in PerQA is English. The personas collected are artificial, and do not represent real human beings.

B.2 PerQA Examples

Example questions. Table 8 shows example questions from our new PerQA dataset, demonstrating challenges such as aggregation, temporal conditions, numerical constraints, or joining information from different sources.

Example user data. Table 13 illustrates example events in PerQA. Such events can be of an unstructured, semi-structured or structured nature, and form the personal data in PerQA that systems have to process for answering the questions.

C Details on Configuration

Main results. We fine-tuned the QUD model for 4 epochs. The learning rate was tuned on a with-held dev set, and was eventually set to $5 * 10^{-6}$, with a linear scheduling. The warm-up ratio was set to 0.05, with a weight decay of 0.01, and a batch size of 4. Our train set for the QUD model had 17,933 instances, and the dev set 1,135. These were obtained by running the GPT4o version of ReQAP

---

[9] meta-llama/Meta-Llama-3-8B-Instruct

[10] meta-llama/Meta-Llama-3-70B-Instruct

```
You are an assistant that generates questions, and the corresponding SQL query for deriving the answer, on personal data.

You are provided with the SQL schema of the underlying DB, and a set of example questions and SQL queries.
Your task is to generate 25 questions and their SQL queries.

Make sure that the questions are natural, and the SQL query is compliant with DuckDB -- make use of the provided SQL schema
and examples. Please generate complex but realistic questions and SQL queries -- based on multiple tables and columns, with
joins, group by's, and variable declarations.

Also try to generate diverse questions: formulations can be ad-hoc or well-formed, and their sentence structure could
differ. Remember that question should be natural: do not spell out the exact table names or columns, but make it realistic.

Your output should be a plain JSON object, with no other text.
```

Figure 5: LLM instruction for generating complex questions. In addition, we randomly sample 20 questions that are provided as ICL examples.

Module	Size	#Parameters	Model
QUD	XS	135M	HuggingFaceTB/SmolLM2-135M-Instruct
	S	360M	HuggingFaceTB/SmolLM2-360M-Instruct
	M	1B	meta-llama/Llama-3.2-1B-Instruct
	L	3B	meta-llama/Llama-3.2-3B-Instruct
RETRIEVE	XS	4M	cross-encoder/ms-marco-TinyBERT-L-2-v2
	S	16M	cross-encoder/ms-marco-MiniLM-L-2-v2
	M	23M	cross-encoder/ms-marco-MiniLM-L-6-v2
	L	33M	cross-encoder/ms-marco-MiniLM-L-12-v2
EXTRACT	XS	16M	google/t5-efficient-tiny
	S	31M	google/t5-efficient-mini
	M	70M	lucadiliello/bart-small
	L	139M	facebook/bart-base

Table 9: Models used for the model-size analysis in Table 6.

on the train and dev sets, sampling n=3 generations per input with a temperature of 0.1. This results in multiple candidate operator trees per training instance – we retain only the ones leading to a correct answer, and a maximum of 3 per instance.

The RETRIEVE models (size L) were fine-tuned for 10 epochs with a learning rate of $5 * 10^{-5}$, a batch size of 64, and a weight decay of 0.001. The EXTRACT model (size L) was fine-tuned for 3 epochs with a learning rate of $1 * 10^{-4}$, with a batch size of 16, and a weight decay of 0.01. All our models were trained using an epoch-wise evaluation strategy, and with AdamW as optimizer.

REQAP is implemented in Python, and the generated operators thus follow Python syntax as well.

Baselines. Baselines generally follow the same training scheme. For the RAG baseline, we tuned the number of events in the input on the dev set, and found the performance to be best with 100 events (with GPT4o and LLaMA3.3) and 10 events (with the SFT variant). Note that including >500 events exceeded the context window of GPT4o and LLaMA3.3.

Resources. We trained all our models on a single NVIDIA Tesla H100NVL GPU. Training of the QUD model took up to 2.5 hours (for the 3B LLaMA model), for the RETRIEVE models up to 1.5 hours (for the 33M parameter model), and for the EXTRACT model up to 1.5 hours as well (for the 139M model). For inference of the 70B LLaMA models, we used 4 NVIDIA Tesla H100NVL GPUs.

Model-size analysis. Table 9 lists the details on the models used in the experiment in Table 6, that compares REQAP with models of varying sizes.

Licenses. Our REQAP code is released with an MIT license to enable others to re-use components and further advance the state-of-the-art. TimelineQA was used in experiments under a CC BY-NC 4.0 license.

D Details on Experimental Results

Results per-persona. Table 10 and Table 11 provide results on the test set of PERQA for each persona individually. All of the methods show variance across the different personas. Notably, REQAP performs better than baselines across all personas and models.

Run-time analysis. We conducted a run-time analysis for REQAP, and measured an average run-time of 0.56 s per question for the QUD stage (size M), and an average run-time of 11.53 s for the

Model / Persona →	**Persona 1** (489 questions)			**Persona 2** (495 questions)			**Persona 3** (530 questions)		
Method ↓	**GPT4o**	**LLaMA3.3**	**SFT**	**GPT4o**	**LLaMA3.3**	**SFT**	**GPT4o**	**LLaMA3.3**	**SFT**
Rag	0.180 (0.26)	0.174 (0.25)	0.037 (0.06)	0.154 (0.21)	0.125 (0.19)	0.038 (0.06)	0.147 (0.21)	0.117 (0.17)	0.032 (0.06)
CodeGen	0.344 (0.45)	0.254 (0.33)	0.358 (0.48)	0.349 (0.47)	0.269 (0.35)	0.337 (0.48)	0.326 (0.46)	0.240 (0.33)	0.319 (0.48)
ReQAP (ours)	**0.421** **(0.54)**	**0.348** **(0.46)**	**0.427** **(0.56)**	**0.402** **(0.54)**	**0.331** **(0.47)**	**0.394** **(0.53)**	**0.425** **(0.56)**	**0.353** **(0.49)**	**0.404** **(0.55)**

Table 10: In-depth results for PerQA ***test*** personas 1-3. Metric is Hit@1, and Rlx-Hit@1 in parentheses.

Model / Persona →	**Persona 4** (460 questions)			**Persona 5** (524 questions)			**Persona 6** (546 questions)		
Method ↓	**GPT4o**	**LLaMA3.3**	**SFT**	**GPT4o**	**LLaMA3.3**	**SFT**	**GPT4o**	**LLaMA3.3**	**SFT**
Rag	0.133 (0.17)	0.089 (0.15)	0.015 (0.08)	0.166 (0.22)	0.130 (0.17)	0.029 (0.06)	0.119 (0.17)	0.103 (0.16)	0.024 (0.04)
CodeGen	0.272 (0.41)	0.209 (0.31)	0.285 (0.46)	0.307 (0.44)	0.227 (0.32)	0.292 (0.45)	0.315 (0.45)	0.234 (0.33)	0.299 (0.46)
ReQAP (ours)	**0.341** **(0.49)**	**0.300** **(0.45)**	**0.357** **(0.52)**	**0.385** **(0.53)**	**0.313** **(0.45)**	**0.368** **(0.52)**	**0.342** **(0.49)**	**0.286** **(0.43)**	**0.335** **(0.50)**

Table 11: In-depth results for PerQA ***test*** personas 4-6. Metric is Hit@1, and Rlx-Hit@1 in parentheses.

Operator	**Avg. runtime**
`RETRIEVE`	2.61 s
`EXTRACT`	5.93 s
`JOIN`	2.29 s
`GROUP_BY`	4.82×10^{-3} s
`FILTER`	1.65×10^{-2} s
`MAP`	3.44×10^{-1} s
`APPLY`	4.82×10^{-3} s
`UNNEST`	3.36×10^{-1} s
`ARGMIN`	3.36×10^{-1} s
`ARGMAX`	8.61×10^{-5} s
`MIN`	1.74×10^{-4} s
`MAX`	1.33×10^{-4} s
`SUM`	1.20×10^{-4} s
`AVG`	1.13×10^{-4} s

Table 12: Average operator run-times.

OTX stage (size L), leading to an average run-time of 12.09 s. The median run-time is much lower with 3.57 s. Table 12 shows the average run-time per operator. Unsurprisingly, the `RETRIEVE` and `EXTRACT` operators, that are based on LMs and run on thousands of inputs for each question, are the most computationally expensive.

Anecdotal examples. Table 14 shows anecdotal examples for operator trees generated by QUD.

Anecdotal failure cases. QA over heterogeneous personal data often requires aggregating or ordering information from hundreds of events. For such questions, even minor errors in one of the operators can lead to an incorrect answer – as observed in PerQA and in the user study – making the task a real challenge even with recent advances in NLP.

We report one such anecdote from the user study (we asked the respective user for consent on reporting this anecdote): for the user query *"most expensive purchase"*, ReQAP derived the correct answer among the top few results (based on the available user data). However, a mail from a mailing list, mentioning that an apartment was available for a rent of 730€, was incorrectly classified as relevant by `RETRIEVE`. The corresponding value, 730€ was then found to be the highest, leading to an incorrect answer. Note that such a case would yield a Hit@1 and Rlx-Hit@1 of 0 on PerQA.

E Details on User Study

Data exports. Our current implementation supports data exports from Amazon Marketplace, Spotify, Netflix, Apple, Google, as well as mails in the *".mbox"* format and calendar data in the *".ics"* format. Note that it is trivial to extend our implementation with additional services, such as streaming data from Amazon Prime. This simply requires providing the data in the format expected by ReQAP, i.e., as sets of key-value pairs.

Question creation. Each participant of our user study created 100 questions, that capture their real information needs on their personal data, first. We provided as little instructions as possible, to avoid biasing them towards a specific type of questions. For the evaluation, the participants then chose 20 questions that are answerable with their personal data, the sources we support, and the data in current exports. For example, the Spotify exports for music

Unstructured sources
{“source”: ***“mail”***, “subject”: *“Halloween was a blast!”*, “timestamp”: *“2016-11-01T10:44:21Z”*, “sender”: *“Isabella Ruiz”*, “recipient”: *“Lucia Hernández”*, “text”: *“Hey Lucia, hope you’re doing well! I just wanted to share with you that I had a fantastic time celebrating Halloween with Carla last night. We stayed in and had a movie marathon, but it was still super fun. I loved the decorations and the costumes - Carla’s was amazing! I’ll have to show you the pictures soon. How about we plan something for this weekend? Maybe we can go for a hike or something? Let me know what you think. Take care, Isabella”*}
{“source”: ***“social_media”***, “text”: *“Pumped up after a grueling 117-minute weight training session. My heart rate reached a max of 182, but I managed to keep it steady throughout. Feeling strong and energized now! #weightlifting #fitnessmotivation”*}
Semi-structured sources
{“source”: ***“calendar”***, “start_date”: *“2024-08-19”*, “start_time”: *“12:00:00”*, “end_date”: *“2024-08-19”*, “end_time”: *“13:00:00”*, “summary”: *“Lunch with Mum and Dad”*, “location”: *“The Parthenon”*, “description”: *“Greek food”*}
Structured sources
{“source”: ***“online_purchase”***, . . . , “order”: *“1 x Cosmic Funk”*, “product_quantity”: *“1”*, “product”: *“Cosmic Funk”*, “category”: *“CDs & Vinyl”*, “price”: *“5.99 EUR”*}
{“source”: ***“tvseries_stream”***, . . . , “duration”: *“2460”*, *“stream_end_time”*: *“00:24:17”*, “stream_style”: *“tv_series”*, “tvseries_title”: *“Scrubs”*, “season_name”: *“Scrubs, season 1”*, “episode_name”: *“My Nickname”*, “episode_number”: *“10”*, “tvseries_season”: *“1”*, “duration_unit”: *“seconds”*}
{“source”: ***“movie_stream”***, . . . , “duration”: *“6660”*, “stream_end_time”: *“01:10:41”*, “stream_style”: *“movie”*, “movie_title”: *“Beverly Hills Cop III”*, “stream_full_title”: *“Beverly Hills Cop III”*, “duration_unit”: *“seconds”*}
{“source”: ***“workout”***, . . . , “workout_type”: *“soccer”*, “duration”: *“126”*, “duration_unit”: *“min”*, “minimum_heart_rate”: *“120”*, “maximum_heart_rate”: *“188”*, “average_heart_rate”: *“156.87”*}

Table 13: Example user data for a fictional persona in PERQA.

streams would not include the genre or album of the songs, which appeared in many user questions. Similarly, we observed questions about watchlists on Netflix, like *“What is the name of the movies or series that has been in the user Emma’s lists for the longest time?”*, that are not covered in current data exports. All 2,005 questions collected in our user study, including the ones not used for evaluation, are available at https://reqap.mpi-inf.mpg.de.

QUD model. When conducting the user study, we were using a preliminary version of the QUD model. Specifically, we used CodeT5-large (Wang et al., 2021) with 738M parameters, fine-tuned as outlined in Section 3.1[11]. We later found variants based on LLaMA-1B to perform substantially better (on PERQA), while demonstrating comparable or faster runtime. Notably, compared to sequence-to-sequence models, decoder-only models can be run in a quantized manner much more easily. We hypothesize that the usage of LLaMA-1B might have further improved the results of the user study reported in Section 7. For the `RETRIEVE` operator, we used the model of size S (16M), and for the `EXTRACT` operator we used XS (16M).

Payment. We hired local students. Participants were paid $15 per hour for the user study, which is well above the local minimum wage.

---

[11] Salesforce/codet5-large

Anecdotal example 1 – *“How much money did I spend on online purchases in March 2022?”*

(0) *QUD("How much money did I spend on online purchases in March 2022?")*
(1) SUM(l=*QUD("my online purchases in March 2022 with amounts")*, attr_name="amount_spent")
(2) EXTRACT(l=*QUD("my online purchases in March 2022")*, attr_names=["amount_spent"], attr_types=[float])
(3) FILTER(l=*QUD("my online purchases with date")*,
filter=lambda attr: attr["purchase_date"].year == 2022 and attr["purchase_date"].month == 3)
(4) EXTRACT(l=*QUD("my online purchases")*, attr_names=["purchase_date"], attr_types=[date.fromisoformat])
(5) RETRIEVE(query="my online purchases")

Anecdotal example 2 – *“First football training after I started as Engineer – when was it?”*

(0) *QUD("First football training after I started as Engineer – when was it?")*
(1) MIN(l=*QUD("football training sessions after I started as Engineer")*, attr_name="start_datetime")
(2) FILTER(l=*QUD("my football training sessions with datetime")*,
filter=lambda attr: attr["start_datetime"] >=*QUD("first start datetime as Engineer")*.result)
(3.1) EXTRACT(l=*QUD("my football training sessions")*, attr_names=["start_datetime"],
attr_types=[datetime.fromtimestamp])
(4.1) RETRIEVE(query="I played football")
(3.2) MIN(l=*QUD("start datetime as Engineer")*, attr_name="start_datetime")
(4.2) EXTRACT(l=*QUD("I started as Engineer")*, attr_names=["start_datetime"],
attr_types=[datetime.fromtimestamp])
(5.2) RETRIEVE(query="I started as Engineer")

Anecdotal example 3 – *“which restaurants did we visit when in Bali, Indonesia”*

(0) *QUD("Which doctor’s appointment was the earliest in the day?")*
(1) EXTRACT(l=*QUD("restaurants we visited in Bali, Indonesia")*, attr_names=["restaurant_name"],
attr_types=[str])
(2) JOIN(l1=*QUD("restaurants we visited with date")*,
l2=*QUD("we were in Bali, Indonesia with start and end date")*,
condition="i1.visit_date >= i2.start_date and i1.visit_date <= i2.end_date")
(3.1) EXTRACT(l=*QUD("restaurants I visited")*, attr_names=["visit_date", "restaurant_name"],
attr_types=[date.fromisoformat, str])
(4.1) RETRIEVE(query="restaurants I visited")
(3.2) EXTRACT(l=*QUD("I was in Bali, Indonesia")*, attr_names=["start_date", "end_date"],
attr_types=[date.fromisoformat, date.fromisoformat])
(4.2) RETRIEVE(query="I was in Bali, Indonesia")

Anecdotal example 4 – *“Which doctor’s appointment was the earliest in the day?”*

(0) *QUD("Which doctor’s appointment was the earliest in the day?")*
(1) ARGMIN(l=*QUD("my doctor’s appointments with start time")*, arg_attr_name="start_time",
val_attr_name="appointment_details")
(2) EXTRACT(l=*QUD("my doctor’s appointments")*, attr_names=["start_time", "appointment_details"],
attr_types=[time.fromisoformat, str])
(3) RETRIEVE(query="my doctor’s appointments")

Anecdotal example 5 – *“how many products did I buy online in the last 6 months?”*

(0) *QUD("how many products did I buy online in the last 6 months?")*
(1) SUM(l=*QUD("products bought online in the last 6 months")*, attr_name="quantity")
(2) FILTER(l=*QUD("products bought online with purchase date")*,
filter=lambda attr: attr["purchase_date"] >= (date.today() - relativedelta(months=6)))
(3) EXTRACT(l=*QUD("products bought online")*, attr_names=["purchase_date", "quantity"],
attr_types=[date.fromisoformat, int])
(4) RETRIEVE(query="I bought a product online")

Table 14: **Anecdotal examples** of operator trees generated by REQAP.

ICL example 1 – *"On which day did I listen to music the most?"*

User: Starting with new question. Input: QUD("On which day did I listen to music the most?")
Assistant: ARGMAX(l= QUD("number of songs I listened per day?") , arg_attr_name="num_songs", val_attr_name="start_date")
User: Input: QUD("number of songs I listened per day?")
Assistant: MAP(l= QUD("my songs I listened to grouped by day") , fct=len, res_name="num_songs")
User: Input: QUD("my songs I listened to grouped by day")
Assistant: GROUP_BY(l= QUD("instances I listened to music with date") , attr_names=["start_date"])
User: Input: QUD("instances I listened to music with date")
Assistant: EXTRACT(l= QUD("I listened to music") , attr_names=["start_date"], attr_types=[date.fromisoformat])
User: Input: QUD("I listened to music")
Assistant: RETRIEVE(query="I listened to music")

ICL example 2 – *"how often did I meet with both my parents in the evening?"*

User: Starting with new question.
Input: QUD("how often did I meet with both my parents in the evening?")
Assistant: APPLY(l= QUD("I met with both my parents in the evening") , fct=len)
User: Input: QUD("I met with both my parents in the evening")
Assistant: FILTER(l= QUD("instances I met with both my parents") , filter=lambda attr: attr["start_time"].hour >= 18 and attr["start_time"].hour < 24)
User: Input: QUD("instances I met with both my parents")
Assistant: JOIN(l1= QUD("instances I met with my mum") , l2= QUD("instances I met with my dad") , condition="i1.start_datetime <= i2.end_datetime and i2.start_datetime <= i1.end_datetime")
User: Input: QUD("instances I met with my mum")
Assistant: RETRIEVE(query="I met with my mum")
User: Input: QUD("instances I met with my dad")
Assistant: RETRIEVE(query="I met with my dad")

ICL example 3 – *"how much money did I spend online the last three years?"*

User: Starting with new question.
Input: QUD("how much money did I spend online the last three years?")
Assistant: SUM(l= QUD("my online purchases in the last three years with amounts") , attr_name="amount_spent")
User: Input: QUD("my online purchases in the last three years with amounts")
Assistant: EXTRACT(l= QUD("my online purchases in the last three years") , attr_names=["amount_spent"], attr_types=[float])
User: Input: QUD("my online purchases in the last three years")
Assistant: FILTER(l= QUD("my online purchases with date") , filter=lambda attr: attr["purchase_date"] >= (date.today() - relativedelta(years=3)))
User: Input: QUD("my online purchases with date")
Assistant: EXTRACT(l= QUD("my online purchases") , attr_names=["purchase_date"], attr_types=[date.fromisoformat])
User: Input: QUD("my online purchases")
Assistant: RETRIEVE(query="my online purchases")

ICL example 4 – *"which artist did I listen to most when running?"*

User: Starting with new question.
Input: QUD("which artist did I listen to most when running?")
Assistant: ARGMAX(l= QUD("number of songs grouped by artist while running") , arg_attr_name="count", val_attr_name="artist")
User: Input: QUD("number of songs grouped by artist while running")
Assistant: MAP(l= QUD("songs grouped by artist while running") , fct=len, res_name="count")
User: Input: QUD("songs grouped by artist while running")
Assistant: GROUP_BY(l= QUD("songs listened to while running with artist") , attr_names=["artist"])
User: Input: QUD("songs listened to while running with artist")
Assistant: UNNEST(l= QUD("songs listened to while running with artist names") , nested_attr_name="artist_names", unnested_attr_name="artist")
User: Input: QUD("songs listened to while running with artist names")
Assistant: EXTRACT(l= QUD("songs listened to while running") , attr_names=["artist_names"], attr_types=[list])
User: Input: QUD("songs listened to while running")
Assistant: JOIN(l1= QUD("songs I listened to with start and end datetime") , l2= QUD("I went running with start and end datetime") , condition="i1.start_datetime >= i2.start_datetime and i1.end_datetime <= i2.end_datetime")
User: Input: QUD("songs I listened to with start and end datetime")
Assistant: EXTRACT(l= QUD("songs I listened to") , attr_names=["start_datetime", "end_datetime"], attr_types=[datetime.fromtimestamp, datetime.fromtimestamp])
User: Input: QUD("songs I listened to")
Assistant: RETRIEVE(query="I listened to a song")
User: Input: QUD("I went running with start and end datetime")
Assistant: EXTRACT(l= QUD("I went running") , attr_names=["start_datetime", "end_datetime"], attr_types=[datetime.fromtimestamp, datetime.fromtimestamp])
User: Input: QUD("I went running")
Assistant: RETRIEVE(query="I went running")

ICL example 5 – *"how often did I meet with Robert in the park?"*

User: Starting with new question.
Input: QUD("how often did I meet with Robert in the park?")
Assistant: APPLY(l= QUD("I met with Robert in the park") , fct=len)
User: Input: QUD("I met with Robert in the park")
Assistant: FILTER(l= QUD("I met with Robert with location") , filter=lambda attr: "park" in attr["location"].lower())
User: Input: QUD("I met with Robert with location")
Assistant: EXTRACT(l= QUD("I met with Robert") , attr_names=["location"], attr_types=[str])
User: Input: QUD("I met with Robert")
Assistant: FILTER(l= QUD("I met with someone with participants") , filter=lambda attr: any("robert" in p.lower() for p in attr["participants"]))
User: Input: QUD("I met with someone with participants")
Assistant: EXTRACT(l= QUD("I met with someone") , attr_names=["participants"], attr_types=[list])
User: Input: QUD("I met with someone")
Assistant: RETRIEVE(query="I met with someone")

Table 15: Subset of **ICL examples** used for the LLM-based version of our QUD stage. We handcrafted 40 such examples, and retrieve 8 of them *dynamically*, per step, during inference (via BM25).