BETZE: Benchmarking Data Exploration Tools with (Almost) Zero Effort

Nico Schäfer
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
nschaefer@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

***Abstract*—In this paper, we propose BETZE, a benchmark generator to evaluate the performance of data exploration solutions for semi-structured data. It is tailored to the typical query capabilities of modern JSON document stores and can be extended to match more. At its core, the query generator mimics the behavior of a data scientist through a model similar to the random surfer idea known from PageRank. We propose preset parameters that pose different query loads to the system, intended to reflect novice, intermediate, and expert users interacting with the system. The proposed approach analyzes a given JSON dataset and generates queries into an intermediate representation that is then translated to system-specific query syntax. We have implemented support for MongoDB, PostgreSQL, jq, and our own JSON processor JODA, and describe how additional tools can be supported. To get started, we report on a first experimental study, showing the versatility of the benchmark generator, using the NoBench dataset, and real-world data obtained from Twitter and Reddit.**

I. Introduction

Interactive data exploration [1]–[6] is a fundamental task in data science and related areas to get acquainted to previously unknown datasets, to obtain essential insights, apply data cleaning if needed, and to ultimately transform parts of the data into different application-tailored formats for visualization or further processing. Many a data scientist spend a considerable amount of time in preparing raw data before moving on to more advanced analytic tasks; around 40% or 60% of their time or even higher, depending on study or anecdote [7]–[10].

As an example scenario, consider Alice, who is working as a data scientist in the publicity department of a soccer footwear and apparel company. To increase product visibility in an upcoming German sports event, she wants to analyze the potential impact of placing ads by inspecting discussions in social media, say Twitter. She got her hands on a large file of the raw Twitter stream that does not come with a fixed schema, contains actual tweets and delete messages, creation and changes to user profiles, etc.—utter chaos. She first thinks

This work has been partially funded by the German Federal Ministry of Education and Research under grant number 28DE113C18 (DigiVine). The responsibility for the content of this publication lies with the authors.

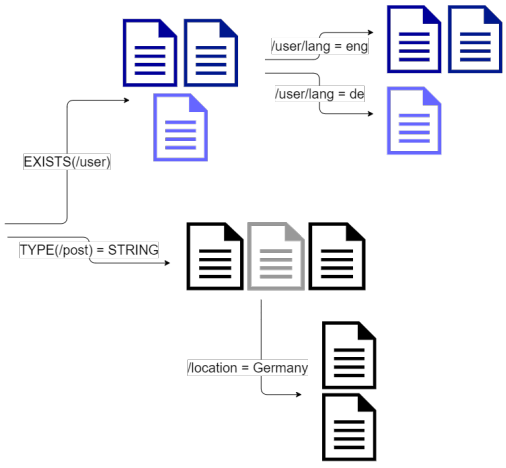


Fig. 1: Data exploration example

she can obtain the tweets by demanding the existence of an attribute 'user' (cf. Figure 1), which, however, just leads to the user profiles, not the tweets. She then discards the results and issues a query asking for all documents that carry a 'post' attribute of type string and then applies an additional predicate asking for the location to be Germany—and so on—until she eventually finds the information she was aiming at.

Depending on Alice's adeptness in using the system's query language, her experience in phrasing queries according to her needs, and existing knowledge of the dataset, she might require more or fewer queries to reach her goal. In this paper, we propose a benchmark generator to evaluate systems designed to explore semi-structured data, precisely, JSON data. JSON is a prominent open data standard that composes data objects consisting of attribute-value pairs. It was designed to be human-readable and has been accepted across various systems and data providers. While the schema-free nature of JSON, together with the support of different data types like arrays and nested objects, offers vast flexibility in capturing data, JSON datasets often become a potpourri of different data concepts. It is then all but impossible to write a one-shot query that delivers the intended results, if the intent is even clearly known. Work on interactive data exploration [5] specifically addresses research challenges around assisting users on their way to find hidden insights through multiple, iterative steps, often via visual support [11]–[13], and features like query

suggestion. Notwithstanding the importance and relevance of such solutions, generated queries need to inevitably be executed by a backend query engine, where low response times are crucial to minimize idle time of users and to keep them focused [14], [15].

Existing benchmarks for semi-structured documents, like NoBench [16], only benchmark generic JSON query features, but do not contain any incremental query loads. In particular, such benchmark datasets often contain simple, artificially generated data values and, thus, do not exhibit the characteristics of real-world data. We opted for a benchmark generator, coined BETZE, that can work with arbitrary JSON datasets, thus, can be used for experiments over different datasets—like Twitter tweets, the aforementioned NoBench dataset, or application scenarios from specific domains [6]. We release the benchmark as an easily executable software package, running in Docker. BETZE first analyzes a given dataset and outputs sequences of queries which can then be used to benchmark the performance of the systems under consideration.

We investigated PostgreSQL, MongoDB, jq, and JODA as representatives of traditional RDBMS, document stores, simple command-line tools, and specialized JSON data processors. Without a standard query language for native JSON stores, we identified a basic set of commonly supported query patterns but kept expandability in mind.

Using a random explorer model, similar to what is known from the famous PageRank algorithm [17], we can set benchmark parameters to reflect different skill levels of users. We give presets for a novice, an intermediate, and an expert user. BETZE consists of two main modules: a dataset analyzer, collecting statistics and insights from a specified JSON dataset, and a query generator component, which generates a set of queries. The dataset analyzer uses JODA [18][1], a JSON data processor developed in our group. JODA is freely available and the entire benchmark can be executed using a simple Docker script/container.

A. Contributions and Outline

In this work, we make the following contributions.

- We propose the use of a random explorer model for query generation and describe an approach that can work with arbitrary JSON datasets to generate query sequences.
- We have implemented support for PostgreSQL, MongoDB, and jq—next to our own approach JODA.
- BETZE is publicly available and is through Docker easily employable [2].
- We present the results of a first experimental evaluation using the proposed benchmark and four competitors.

The remainder is organized as follows. Section II gives an overview of benchmarks, JSON data processing, and data exploration literature. Section III introduces the random explorer model and which types of queries are supported as of now. Section IV discusses how query generation is conducted. Section V describes the default setting and how the benchmark can be employed. It further gives insights into how the benchmark can be adapted to support additional systems. Section VI presents the results of an experimental study, where our JSON processor JODA is compared to three popular systems, using the newly proposed benchmark over two datasets. Finally, Section VII discusses future research directions to extend the benchmark, before Section VIII concludes the paper.

[1]ICDE 2020 demo video of JODA:
https://www.youtube.com/watch?v=HSThB8mxTTA

[2]https://github.com/JODA-Explore/BETZE

II. Related Work

A. Benchmarks

Careful benchmarking is a necessity for understanding the effectiveness and efficiency of data management solutions. It is particularly pivotal when comparing competing solutions to advance the state-of-the-art. For this, across different disciplines, efforts have been made to develop benchmarks datasets and workloads. In particular, in the database community, around SQL query engines, developing and using standard benchmarks has a long tradition. A prominent example of standard benchmarks is the work published by TPC, the transaction processing council, perhaps most prominently the TPC-H or TPC-C benchmarks. Depending on application cases, new benchmarks are proposed to overcome limitations or unrealistic assumptions of existing benchmarks [19]. With the increasing popularity of semi-structured data format XML throughout the early 2000s several benchmarks [20], [21] have been proposed to evaluate XML databases [22], [23], that implemented special query patters like path queries, reflecting the characteristics of the XML data model. In the information retrieval domain, the TREC [24] conference is synonym for a variety of different benchmarks, ranging from traditional text retrieval, to question answering, and temporal data summarization. For XML, a similar attempt was the INEX initiative [25] that organized annual workshops and competitions. Benchmarks like INEX and XMach, assess systems according to the retrieval effectiveness and querying performance, respectively, for fully specified queries using XQuery or XPath expressions. Although we also specifically address semi-structured data, our proposed benchmark generator addresses an entirely different angle. For JSON data, Chasseur *et al.* [16] proposed the NoBench data generator to create benchmarking datasets of variable size. There exist many benchmarks for specialized workloads. For example, G-CARE [26] is a benchmark framework for cardinality estimation techniques of subgraph matching algorithms. While this benchmark concentrates on graphs in relational data systems, similar work exists for graph data in other systems, like LUBM [27], a benchmark for OWL knowledge base systems. They propose a standardized ontology with a set of benchmark queries concentrating on certain characteristics. Similarly, WatDiv [28] was created as a SPARQL benchmark for resource description framework (RDF) data, to evaluate workloads previous SPARQL benchmarks were not suitable for. The need for exploration benchmarks was also noticed by Eichmann *et al.* [5], who proposed IDEBench to benchmark

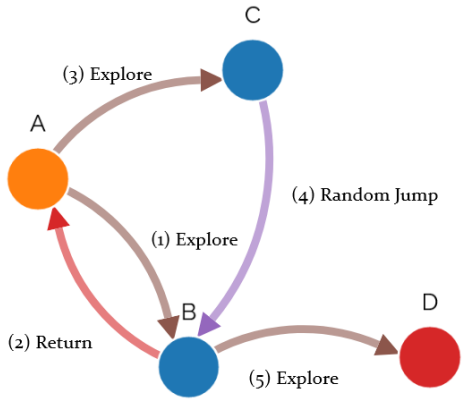


Fig. 2: Possible exploration session in the random explorer model

interactive data exploration on relational data. Our benchmark generator has a similar goal for semi-structured documents and extends it with iterative queries.

B. JSON Data Processing Landscape

With the widespread use of JSON data, the past years have witnessed various approaches that try to incorporate JSON support into existing relational query engines [29], [30]. PostgreSQL supports a JSON data type, too. More specifically, in their very recent work, Durner *et al.* [29] propose to group JSON documents by their available attributes in so-called tiles and describe how the query processing is conducted over tiles in a RDBMS. Bourhis *et al.* [31] focus on establishing a theoretical framework for JSON and propose a lightweight query language for traversing JSON documents. JSONiq [32] and SQL++ [33] are additional query languages for processing JSON documents. In contrast to such efforts, NoSQL systems like MongoDB [34] and CouchDB were specifically tailored to the requirements imposed by JSON and large datasets, but are limited in the query language. Another branch of research considers processing so-called raw data in form of CSV or JSON files, like the NoDB approach by Alagiannis *et al.* [35] and our own JSON processor JODA [18]. Such approaches refrain from classical data indexing and can be considered tools to enable first steps in data exploration, cleaning, and transformation. A recent approach by Baunsgaard *et al.* [2] specifically considers supporting data science workflows over raw data.

C. Work on Query Suggestion and Online Processing

Related to interactive data exploration is also work around query suggestion [3], [4], where the system proposed queries to be evaluated next, according to the previously issued interactions and further data characteristics. For such and related scenarios, where users are constantly issuing new queries as not being satisfied with the results so far, denoted as being in flux [36] work on online processing [37] and approximate query processing can be effective. To improve the exploration capabilities of databases, El-Hindi *et al.* [38] created VisTrees, a multi-dimensional index for interactive computation of histograms. In our own JODA system, we implemented Delta Trees [39], to improve performance for iterative queries, as often found in exploratory query workloads.

	go back probability α	random jump β	Queries per session n
Novice	0.5	0.3	20
Intermediate	0.3	0.2	10
Expert	0.2	0.05	5

TABLE I: Default user configurations

III. Random Explorer Model and Supported Queries

In BETZE, queries are generated using a random explorer model, similar to the random surfer model introduced by Brin and Page [17]. This model simulates a single user exploring one or multiple given sets of JSON documents that we consider the base datasets. By applying queries to base datasets, new datasets are formed—like obtaining a dataset of textual tweets out of the full status updates of Twitter, as in the earlier example around Alice.

Figure 2 shows a sample query session. Here, the simulated user started with dataset A and created a dataset B by issuing a query. After reconsidering, the user went back to the parent dataset A and created another dataset C, for instance, by refining the initial query or starting all over with a new one. Then, the user goes back to dataset B via, what we call, a random jump and creates the dataset D.

That means, after each querying step, the user has four possibilities:

i) **Explore:** Continue with the current dataset by issuing a new query on it, which creates a new dataset (i.e., B in Figure 2)
ii) **Return:** Going back to the parent dataset (e.g., if the extracted knowledge is unsatisfactory or additional insights are required before continuing from there).
iii) **Jump:** Go to any previously created dataset (e.g., if the whole path is deemed uninteresting)
iv) **Stop:** Ending the exploration session. (e.g., the user learned everything they wanted to learn)

The model is supplied with a parameter n that specifies how many queries are generated per exploration session. The remaining choices of the user are modeled by a weighted random decision. α represents the probability of the user going back to the parent dataset, while β is the probability of the user randomly jumping to another dataset. Hence, the probability of the user continuing with the most recent dataset is given by $1 - \alpha - \beta$.

Depending on these parameters, the generated queries have different characteristics. If, for instance, the likelihood to continue *exploring* a newly-created dataset is set high, the load

to the underlying system to execute this more constraint query is likely to be lower due to more effective indexing or re-use of intermediate results. In contrast, if the probability of returning to the parent dataset is higher, the subsequent query will be more costly. Together with the configurable length of a query session, i.e., the number of queries required generated by the random explorer model, we can, thus, control the amount of work the query execution engine has to cope with.

Since we aim at evaluating the query execution performance of underlying systems, we propose three default configurations, as listed in Table I, that cause heavy, intermediate, and low load according to the aforementioned rationales—as the experiments confirm, e.g., in Figure 5. While we do not intend to fully model real users, these configurations should coarsely reflect different skill levels of data scientists working with data management systems for data preparation and the corresponding variety of time they invest in such tasks [7]–[10].

- A **novice user** does not have any knowledge about the tools and no intuition about the dataset and how to achieve their goal. Such a user will issue more queries that backtrack and often jump to another random dataset when noticing that the current path probably will not yield the desired result.
- An **intermediate user**, on the other hand, already has knowledge about the used tools and some intuition about how to reach the information needed. The chosen path is often correct, with only minor adjustments—in the form of backtracking.
- An **expert user** knows the available tools well and either already has knowledge about the dataset or a good intuition about how to reach the goal—nearly no backtracking and only very minor random exploration is needed.

A session represents the interactions of a single user, from starting the exploration to finding the desired result. To evaluate multi-user systems, we could generate multiple sessions and execute them simultaneously. Using different configurations for different sessions is also possible. Figure 3 shows an example session for each introduced user configuration. Orange nodes represent the starting dataset(s), blue nodes intermediate dataset(s), and the red node the finally created dataset. The links are colored depending on whether they are random jumps (purple), backtracking (red), or queries generating new datasets (brown).

A. Query Support

Our initial implementation of BETZE is able to generate queries for JODA [18], MongoDB [34], jq [40], and PostgreSQL [41]. As mentioned before, support for additional systems can be added easily with a few lines of code. We will give more details on how this can be done in Section IV-D. The query features used by BETZE need to be simple enough to be supported by as many external systems as possible but complex enough to be useful and realistic. After analyzing the query expressiveness of the above systems regarding JSON processing, we opted to use the following tasks:

- Loading datasets
- Filtering datasets
  a) Using simple predicates
  b) Combined only with logical binary **AND** and **OR** operators
- Outputting:
  a) the whole content of the selected documents
  b) an aggregation of the documents (grouped by an attribute)

These features were supported by all evaluated systems and are sufficient to enable the creation of realistic exploratory query workloads. We added at least one filter predicate for each data type supported by JSON:

- **`EXISTS(<ptr>)`**: checks existence of an attribute
- **`ISSTRING(<ptr>)`**: checks if attribute is of type string
- **`<ptr> == <int>`**: equality check with integer
- **`<ptr> <comparison> <float>`**: comparison with floating point
- **`<ptr> == <string>`**: equality check with string
- **`HASPREFIX(<ptr>,<string>)`**: checks if attribute is string and has prefix
- **`<ptr> == <bool>`**: equality check with boolean
- **`ARRSIZE(<ptr>) <comparison> <int>`**: comparison of array size with constant number
- **`OBJSIZE(<ptr>) <comparison> <int>`**: comparison of object size (number of children) with constant number

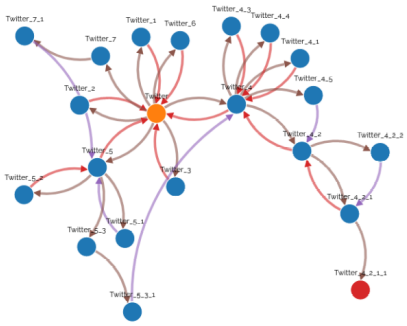
In addition, BETZE can be configured to create (group-by) aggregation queries. The currently supported aggregation functions are:

- **`COUNT(<ptr>)`**: counts the number of documents having this attribute
- **`SUM(<ptr>)`**: sums the numerical attribute, if it exists
- **`<Agg> GROUP BY <ptr>`**: groups one of the previous aggregations by the given numerical, string, or boolean attribute

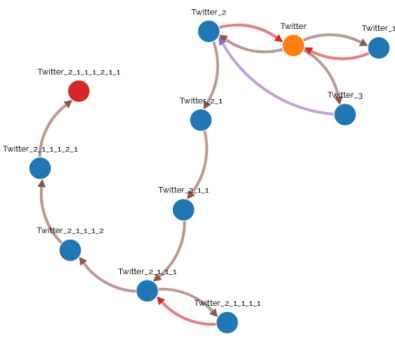
These predicates and aggregations are later translated into system-specific syntax. BETZE is designed to be extendible and additional functions may be added with ease. Listing 1 shows a single query, expressed in the syntax of the four currently supported systems. In this example, a Boolean predicate filters the document set, which is then aggregated by a grouped count aggregation.

IV. Data Analyzer and Query Generator

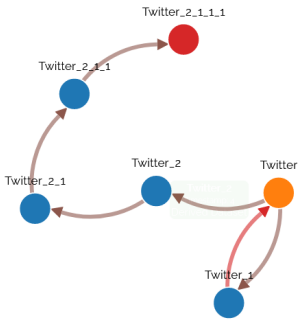
BETZE comprises two main modules, a data analyzer, and a query generator. First, the analyzer creates a statistical and structural summary of the input datasets. This data is stored in a JSON file and used by the generator to create the actual benchmark queries.



(a) Example novice user session



(b) Example intermediate user session



(c) Example expert user session

Fig. 3: Example user sessions

```
# JODA
LOAD Twitter
CHOOSE '/retweeted_status/user/verified' == false
AGG GROUP COUNT('') AS count BY '/user/time_zone')

# JQ
jq -c 'inputs | select(.retweeted_status.user.
    verified == false)' Twitter.json |
jq -s -c 'def agg(s): reduce s as $x (0; . + 1);
   group_by(.user.time_zone)
  | map({group: .[0].user.time_zone,
    count: agg(.[])})'

# MongoDB
db.Twitter.aggregate([{ $match :
  { "retweeted_status.user.verified" : false},
  { $group: { _id: '$user.time_zone',
  count: { $sum: 1 }}}]);

# PostgreSQL
SELECT doc #> '{user,time_zone}' as group, COUNT(*)
FROM Twitter
WHERE jsonb_path_exists(doc,
  '$.retweeted_status.user.verified ? (@ == false)')
GROUP BY doc #> '{user,time_zone}';
```

Listing 1: Example queries for each of the supported languages

```
[{
      "dataset":"Twitter",
      "Count":10000,
      "Paths":{
        "/user":{
            "Count":1000,
            "Objecttype":{
              "Count":1000,
              "MinMembers":1,"MaxMembers":3
            }
        },
        "/user/name":{
            "Count":500,
            "StringType":{
              "Count":500, "Prefixes":[...]
            }
        },
        ...
      }
}]
```

Listing 2: Example analysis file

A. Data Analysis

Given the input datasets, the analyzer uses JODA as a backend to analyze the data. A sample output of this analysis is shown in Listing 2, for the case of 10 000 documents obtained from the Twitter API, 1 000 of the given documents have a `/user` attribute which is always of type object and has between one and three members. One of the children of the `/user` object is a `name`, which only exists in half of the objects. For each distinct path in the source documents, we—currently—store the number of documents that contain this path and additional type-specific statistics.

For every JSON type, we keep the total number of its occurrence separately. We also store the minimum and maximum values for numerical types—split into integer and real numbers. In contrast, for the Boolean type, we store the number of true values—and thus, also the number of false values. The minimum and the maximum number of children is kept for object and array types. We also store a set of prefixes and their number of occurrences for string types. Once the analysis is complete, the statistics file will be used for the actual benchmark generation. It can also be stored and shared for future generator runs without the actual dataset.

B. The Query Generator

After the input datasets have been analyzed, the generator will use the resulting statistics to create benchmark queries. Each execution of the generator creates one session of queries, that is, the simulated interaction of a single data scientist with an exploration tool. In the very beginning, there are only the initial datasets. Then, as queries are created, more and more datasets are available. The random explorer model is used to decide how to proceed in each step. Say, a new query should be issued against the dataset that was created last. The generator analyses the provided statistics and generates a filter predicate by first randomly selecting a JSON path based on the obtained structural information. Then, all implemented predicates (cf., Section III-A) are checked for their validity

to be applied to this path. If no predicate is applicable to the given path, another path is chosen. If no paths remain, another dataset is chosen through a random jump. As soon as an applicable predicate type (e.g., `==int`) is found, the available statistics and parameters are used to instantiate it. This process is different for each predicate type, but each one tries to achieve the desired configured selectivity range (default: $[0.2, 0.9]$). For instance, assuming there is a path with 90% numerical values, 10% string values, and the numerical values are between 1 and 10. Now, the generator decided—by dice roll— that a numerical comparison predicate should be generated. As the numerical type of the attribute already has a selectivity of 0.9, the system will try to generate a predicate with a selectivity in the range of $[\frac{0.2}{0.9}, \frac{0.9}{0.9}] = [0.22, 1]$. The final value is then randomized within this range, which could result in the predicate $[path] \geq 5$. If the desired selectivity cannot be reached using the chosen predicate, the generator will try to augment it with another condition. In case the selectivity is too high, it will be combined using a logical `AND`, and if it is too low with `OR`. Aggregations are generated similarly if BETZE is configured to create them. Again, a path is chosen at random, and all supported aggregation functions are evaluated for suitability. Then, a random suitable function is generated. If the group-by aggregation function is enabled, the generator will try a limited number of times to find a suitable grouping expression by randomly choosing another path. If successful, the previously chosen aggregation will be grouped by this path. Otherwise, the aggregation is performed over all documents.

The generator will then execute each generated query in the data processor and calculate the actual selectivity. If it is within desired selectivity range, the query is kept. If not, it is discarded. The runtime of this step depends on the system specs and size of the dataset, as all queries have to be executed. Hence, having the analyzer generate accurate summaries of the base dataset is very important. Accurate summaries allow the generator to estimate the selectivities better and prevent unnecessary queries to the underlying data processor.

After a query is generated, a name for the new dataset is determined, and a `store` instruction using this name is appended to the query. Query and dataset are then added to a dependency graph (similar to the one shown in Figures 2,3). From there, the next step is given by the random explorer model. When all queries are generated, the generator returns an internal query representation. For each supported system, a query language module is called in order to translate the internal representation into a system-specific query which is then written to a file.

C. Generating Specialized Benchmarks

Due to the randomized nature of the benchmark generator, executing it multiple times on the same dataset will yield different queries. By supplying a *seed* value to BETZE, repeatable generator runs are possible. This is especially useful if benchmarks should be shared or experiments should be reproducible. By sharing the seed value and the means to acquire or generate the dataset, a second party can regenerate the same benchmarks and validate the results or produce new queries for another system. If benchmarking of specialized workloads is required, for example, to stress-test a system prototype in all details, BETZE allows overwriting preset parameters.

Configuring random surfer model: Most importantly, the random explorer model can be configured differently beyond the three types of simulated users. These presets set the probabilities for backtracking to the parent dataset (α), for the random jump (β), and the number of generated queries to completely change the characteristics of the dataset dependency graph. By default, the intermediate user is used. But each of these values can also be set explicitly to either overwrite a part of a preset or create a unique configuration.

Adapting target selectivity range: The user may also change the default minimum (0.2) and maximum (0.9) selectivity that each query should adhere to. However, the range of allowed selectivities should not be chosen too narrow, as the generator may not be able to generate a predicate with the same selectivity accurately. Additionally, the set of permissible predicates can be set via exclusion or inclusion lists, for instance, to allow only string-type predicates to benchmark a newly implemented string index.

Output of query results: Depending on the system under evaluation, an executed query would output all result documents, which can result in a large I/O overhead that may not be part of the desired evaluation. For instance, jq queries would always output the whole content over the standard output stream (stdout), while other systems, like JODA and MongoDB may only return a reference or iterator to the evaluated result set. To reduce this overhead and to evaluate the whole query pipeline, the user may choose to generate aggregation queries, which prevent the materialization and/or output of large quantities of data. When the generation of aggregations is enabled, the user may also specify which aggregation functions should be used and the percentage of queries that should be aggregated (default: all).

Materializing query results: As explained in Section III and shown in Figure 2, the underlying model treats each result of an exploratory query as an independent dataset. Each system would then store the result of every query of a session in an intermediate dataset. For example, may JODA use the `STORE` command and MongoDB an additional `$out` stage to create new internal datasets. jq would simply write to a new file on the filesystem. By default, however, each generated query will only reference the base dataset and extend the predicate. If, for instance, the dataset B in Figure 2 was created by a query with predicate x, then the query creating D with predicate y would be exported as a query based on dataset A with predicate $x \wedge y$. The intermediate set feature cannot be used with the previously described aggregation feature, as the result dataset would only consist

```
type Language interface {
  // Display name of the language
  Name() string
  // Unique identifier name for the language
  ShortName() string
  // Translates a Query into the language
  Translate(query query.Query) string
  // Writes a comment with the system specific
      comment syntax.
  Comment(comment string) string
  // Returns necessary header string to be added
      as preface to the system-specific file
  Header() string
  // Returns the delimiting symbol/string to
      terminate a query
  QueryDelimiter() string
}
```

Listing 3: Language interface

of one aggregated document, which can not be filtered further.

Weighted paths: For datasets where the documents are deeply nested, and most of the attributes are situated in the lower levels, it might be undesirable to choose the attribute to generate a predicate for in a truly random fashion. Especially if the large nested objects exist only in small subsets of documents, as every predicate evaluated on its children will inherit the selectivity of the parents existence. Additionally, real users would choose the top-most fitting attribute to use in a predicate. To simulate this affinity, the generator can be configured to choose the next attribute with a weight that is inversely correlated to the path length. Using this function, an attribute is much more likely to be chosen the closer to the root node it is. For documents that consist of mostly object- and array-type attributes closer to the root, this will result in an increased usage of the `OBJSIZE` and `ARRSIZE` attributes. By default, this setting is disabled, and the next attribute is chosen in an unweighted manner.

D. Extendability

As of now, the benchmark generator generates queries compatible with MongoDB, jq, PostgreSQL, and JODA. In order to add different languages, the simple interface shown in Listing 3 needs to be implemented. The language interface provides the generator with basic identifying information about the system and the means to create a system-specific query file.

To be able to support multiple languages, queries are first generated in an internal representation, which is easy to translate into different query languages. A query is represented by a base dataset on which the query is executed, an optional dataset to store the result in, an optional query predicate tree, and an optional aggregation function. The filter-predicate tree is composed of `OR` and `AND` predicates as inner nodes, and filtering functions (e.g., equality, comparisons, prefixmatching) as leaf nodes. For each implemented language interface, the

```
# ./generate_queries.sh <dataset dir>
#   <dir-to-store-query-files> [<seed>] [<options>]
./generate_queries.sh /data ~/queries 123  --preset
    expert --aggregate

# ./benchmark_queries.sh <dataset dir> <query dir>
#    [<docker run options>]
./benchmark_queries.sh /data ~/queries
```

Listing 4: CLI commands to generate a session and benchmark it with all supported systems.

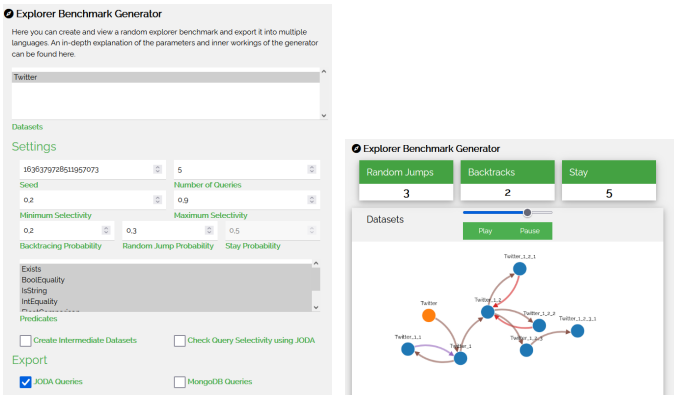
benchmark generator will translate the internal query representations into a system-specific script using the `Translate` function.

It is also possible to easily implement new predicates and aggregation functions in the system. For each function, one `Factory` class has to be implemented with two functions. First, given a specific path of the analyzed dataset, the factory has to decide whether the function can be generated for the given path. For example, if the dataset does not have any statistics about the minimum and maximum numerical values of an attribute or no numerical data exists at all, we cannot create a numerical comparison predicate. After the system chooses one possible predicate factory, it will call its `Generate` function. Given a dataset path with statistics, a random generator, and an exclusion list of already generated predicates to prevent duplicates, it generates a query predicate with a desired selectivity.

In Section IV-A we described how JODA is used to analyze the basic dataset. While it is currently the only analyzer backend, we support the usage of additional backends. It is possible, for example, to write a MongoDB connector and let it compute the analytical data. If, for some reason, the backend cannot provide all supported statistics, the generator is able to generate benchmark queries anyway. That means, for example, if no string prefixes are provided to the generator, it falls back to the string-type checker predicate. For most missing statistics, default values are provided, i.e., if the Boolean type statistics do not provide true/false counts, a uniform distribution is assumed. Similarly, the JODA backend, during query generation, can also be replaced with another system. It can even be omitted completely, in which case the generator will not double-check the generated queries. The statistics of each generated sub-dataset are then calculated by scaling the statistics of the base dataset according to the selectivities. Using no backend to check the query selectivities is currently not recommended, as this scaling does not provide the necessary accuracy to generate queries of acceptable quality.

V. Getting Started with BETZE

We provide multiple ways to interact with BETZE to make it easily accessible. The library itself is written in Go and allows integrating additional data evaluation systems for query generation and data analysis. The software will be open-sourced upon acceptance of this paper. We also implemented



(a) Configuration page (b) Interactive result page

Fig. 4: The web interface of BETZE

a simple command-line interface (CLI) wrapper around the library. With this, all settings introduced in this paper can be modified. Creating a session is a two-step process in the CLI: First, a dataset has to be analyzed and the result stored in a JSON file—this is currently done using a JODA instance. Subsequently, the JSON file is passed to a second program run to generate a session. In this step, having a JODA connection is highly recommended for better queries but not required. We also provide a Docker image of the CLI to make installing and using it as simple as possible. Utility bash scripts have also been written that allow the generation of sessions with a single command pointing to a directory with JSON files. Benchmarking these sessions with all currently supported systems is also realized with an additional bash script. Example usage of these scripts is shown in Listing 4. The first command will pull the public JODA Docker image, build a local BETZE image, and generate one query session to be stored in the supplied directory. The second command will then fetch JODA, PostgreSQL, and MongoDB images, build a local lightweight jq image and execute the previously generated queries. Execution logs of all containers and a summary of all runtimes are then stored in the query directory.

To further improve the user experience, we also integrated the explorer library into our JODA web interface. Figure 4a shows the configuration page, where the imported datasets of JODA can be chosen, and all important settings can be set. The generator will then automatically analyze the chosen datasets and generate a user session. The session is then displayed in an interactive graph, shown in Figure 4b, which allows the browsing of all generated queries. The generated code of all supported systems can then be downloaded to be executed. The JODA web interface is also publicly available as a Docker image, and a docker-compose file is available to run it with a JODA server. A short demo of the web interface is available on YouTube[3].

[3]https://youtu.be/U0rJNEP78vY

VI. Evaluation

To evaluate our approach, we employ the following datasets to generate benchmark queries for the currently supported systems. Although BETZE can use multiple datasets at once, we use the datasets separately.

- A **Twitter** dataset, consisting of a 109 GB file containing a sample of the raw Twitter JSON stream[4]. We chose this dataset as it contains real-world data in the form of heterogeneous JSON documents. In total, there are $29,634,708$ JSON documents, where each document has between 7 and 348 deeply-nested attributes, containing every possible JSON type. It is a perfect example to showcase the potential complexity of semi-structured data.
- To study scalability properties of the systems under comparison, we also use the **NoBench** [16] dataset generator to create datasets containing a variable number of documents. Each created document has exactly 21 attributes of all JSON types—except for null—with only minor nesting.
- For the system comparison, we will use an additional real-world dataset consisting of comments on the social media platform **Reddit**[5]. The data consists of a 30 GB file with $53,851,542$ documents. Each document has a fixed schema with 20 attributes and no nesting.

All experiments are executed in a dockerized environment to facilitate the reproducibility of the experiments. The necessary scripts, the source code of the benchmark generator, and the results of the evaluation are available online[6]. Additionally, non-default parameters and commands will be noted for each experiment. As docker host, a server with 4 Xeon E7-4830 CPUs, each having 12 cores—and 24 threads—with 2.1 GHz, and 1 TB of RAM is used. The input data files, as well as the PostgreSQL and MongoDB data directories are located on a ramdisk to reduce the impact of disk I/O, for fair comparison with JODA which operates solely in main memory. jq operates directly on the input data files and has not been specially configured in any way. For each system, the official docker image is used. Except for jq, for which we created our own image with the current Arch-Linux version of 05.Oct.2021, as they do not provide their own image. For PostgreSQL, we used the tag `postgres:13.4-alpine`, for MongoDB `mongo:5.0.3-focal`, and for JODA `ghcr.io/joda-explore/joda/joda:0.13.2`.

Data is then imported, if possible, and benchmark queries are executed right after—no additional configuration is performed to any system. To measure the overall execution time, we use the start and end times of the docker containers. This time includes setup procedures performed by the systems themselves and the import of the datasets, which we call **wall clock** time. We will also measure the query execution time using the capabilities of the systems themselves. The sum of

[4]https://developer.twitter.com/en/docs/labs/sampled-stream
[5]https://files.pushshift.io/reddit/comments/
[6]https://github.com/JODA-Explore/BETZE-Reproducibility

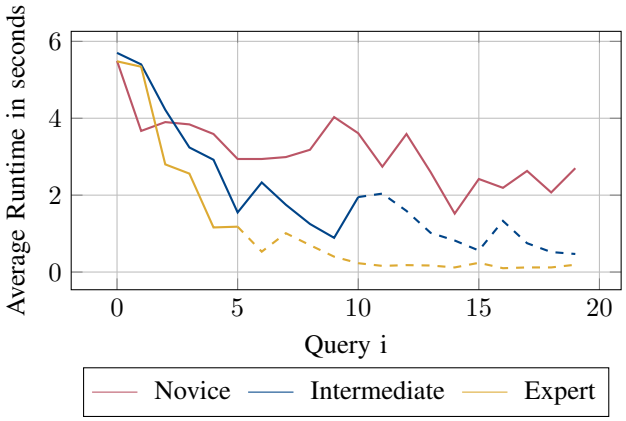


Fig. 5: Trends in execution time for each user preset (20 queries for all users)

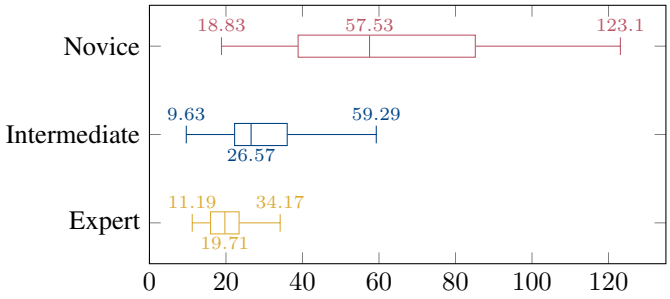


Fig. 6: Execution time (in seconds) of 30 sessions per user configuration

the query execution times without data import is noted as **w/o import**. If not noted otherwise, we will use the **w/o import** time by default as we are more interested in the systems capabilities as in the data I/O.

A. Understanding Impact of User Characteristics

First, we evaluate the influence of different user configurations that model a novice user, an intermediate user, and an expert user. For the *Twitter* dataset and each user configuration, we first fix the number of queries created per session to $n = 20$ to highlight the trends of each user better, regardless of session length. Figure 5 shows the average runtime per query, aggregated over 30 generated sessions with different seeds. For this benchmark-centric experiment, we only used JODA to evaluate the sessions, as we are not interested in a comparison of the individual systems. As we can see, the query runtime generally declines—for all users types—the more queries are executed. This is expected, as with each applied query the datasets get smaller. Further, the more small datasets we have, the higher the probability that we will execute a query on a small set. However, as the results show, the execution time for the novice user often increases. This is due to the higher random-jump probability, which causes the novice user to jump to larger datasets compared to the other two types of users. The query times for the intermediate user also vary by a large degree but reduce faster than the novice user and is, as expected, between the expert and novice user. For the expert user, the reduction of query runtimes is more drastic, and the minimum runtime is reached faster. The query execution times also do not suddenly increase as much as they do for the other configurations. Note that we show here for all user types the execution of 20 queries to understand the trend, although in the following experiments, we consider the query length parameters to be 20, 10, and 5 for the novice, intermediate, and expert users, respectively.

Even though the generation of benchmark sessions is an offline process, it is worth discussing the runtime of the generation process. Generating all 30 sessions, with $1,800$ queries total, took 8h42m overall, of which 8h35m has been spent on dataset analysis and 9m on actual query generation. On average, generating one session took 17m24s, of which 17m10s was analysis time and 14s generation time. The queries have been generated using the complete 109GB dataset and JODA as analysis and selectivity-verification backend. To reduce the generation time, the queries could be generated with a smaller sample dataset at a potential minor loss of query accuracy for the larger dataset. Alternatively, the queries could also be generated without intermediate data analysis and selectivity-verification at a significant risk to not hit the targeted query selectivity.

Figure 6 shows the distribution of the time it takes to execute a session of queries, for each user configuration and 30 sessions, using the *Twitter* dataset. Recall that each session consists of an entire sequence of queries. As each user only uses half the number of queries as the next less-proficient one, one might expect the median execution time to also be exactly 50%. But as we can see, for the expert user, for example, the execution time is actually 74%. This can be explained by larger datasets that have to be evaluated for queries in the beginning, as shown in Figure 5. While the minimum values are relatively close to each other, the maximum values vary wildly. This is a direct consequence of the default selectivity parameters. The best case for every session would be if each query uses the previously filtered dataset as a base and filters it to the smallest possible size. This would result in $N \sum_{i=0}^{n-1} min^i$ evaluated documents, with min being the minimum allowed selectivity and N the number of documents in the original dataset. For session sizes $n = \{5, 10, 20\}$ and $min = 0.2$, this results in $1.2496N, 1.2499N, 1.25N$ evaluated documents respectively. Hence, theoretically, all configurations could have the same minimum query runtime, even if this case is improbable. On the other hand, the worst-case for each query would be if every query uses the original dataset as the base set, filters it once, and then jumps back to the base set. This would result in a maximum of nN document evaluations.

To verify the impact of the different random-jump and backtracking probabilities, we create 20 sessions for each combination of the two probabilities (in steps of 0.1). For each session, we create $n = 10$ queries and measure the

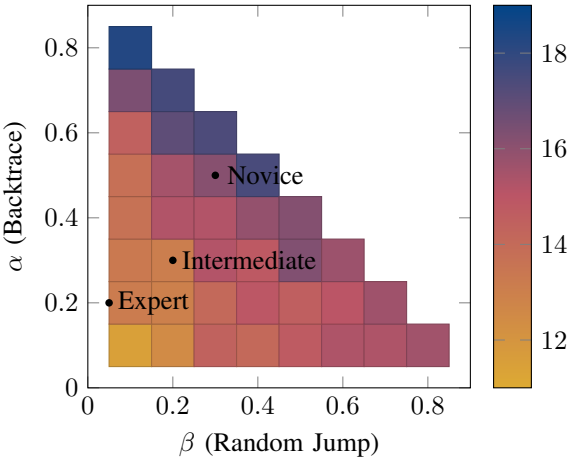


Fig. 7: Aggregated execution times (in seconds) of $n = 10$ queries with varying jump and backtrace probabilities

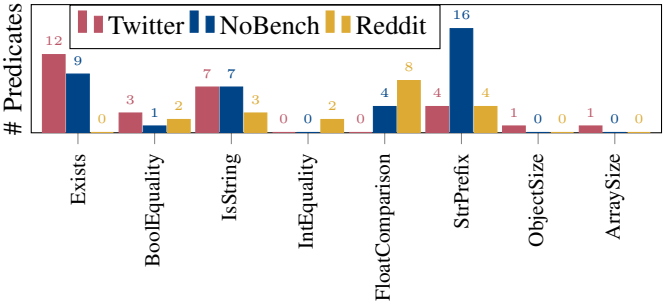


Fig. 8: Number of predicates in the genererated sessions

total execution time of the entire session. Figure 7 shows the average execution time (in seconds) over all 20 sessions, for each probability combination. As expected, having a low α and β value yields the lowest execution times, as jumps to larger datasets are rarer. When either α or β increases, the average session time increases, too. Increasing α has a more significant impact on the execution times, as expected, as backtracking results in queries on larger datasets, while increasing β may also result in jumps to smaller or equally sized datasets.

B. System Comparison

In our first system-centric experiment, we want to evaluate the scalability of the different systems when varying CPU resources and dataset sizes. We used the default (intermediate) preset, with a seed of 123 and default settings, to generate one benchmark session. We executed the session three times for each choice of available CPU threads—in steps of 4—and calculated the average session runtime, shown in Figure 9. Most systems show a stable execution time, regardless of CPU cores. But the execution time for JODA reduces from 4.55 minutes with four threads to 1.51 minutes with 60 threads. This result is expected, as all systems—except for JODA—use only one main thread to evaluate queries.

To evaluate the scalability of the systems in regard to the dataset size, we generate multiple *NoBench* datasets. Figure 10

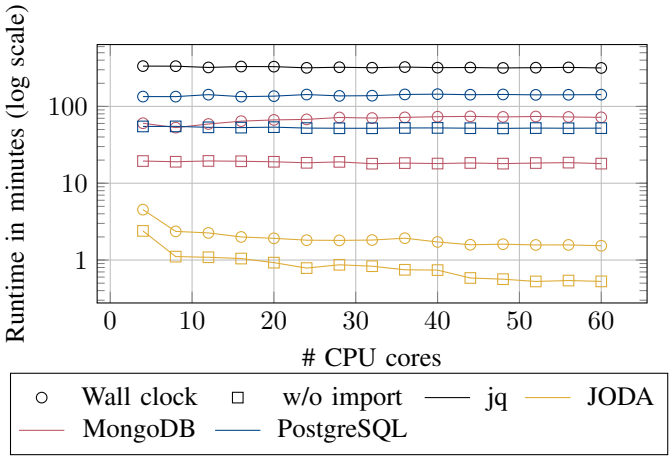


Fig. 9: Runtime of different systems (in minutes) depending on usable CPU threads, using the Twitter dataset

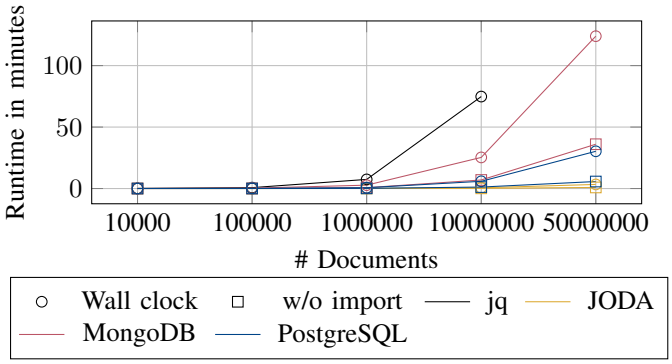


Fig. 10: Runtime of different systems (in minutes) depending on document count, using the NoBench dataset

shows the performance of a default session with seed 123 when executed on a NoBench dataset with x documents. The shown document counts correspond to approximate dataset sizes of 5.5MB, 55MB, 550MB, 5.5GB, and 30GB respectively. We set a timeout of approximately 2 hours for each experiment, which resulted in the omission of jq experiments using the largest dataset.

As we can see, JODA and PostgreSQL are the most stable systems for increasing dataset sizes. For PostgreSQL, the import of the JSON documents takes multiple times longer than the evaluation of the whole session—and even the whole execution of JODA. The reversed performance of the MongoDB and PostgreSQL systems in this benchmark, compared to CPU scalability evaluation, is surprising but can be explained by the different structures of the datasets. While the *Twitter* dataset consists of large, deeply nested documents, the *NoBench* dataset consists of smaller, shallow and sparse documents. PostgreSQL converts each document into a binary form, called JSONB type, and MongoDB stores them in their custom WireTiger storage engine. As the results indicate,

	Twitter	*NoBench*
JODA	1.04m	0.16m
JODA memory evicted	2.13m	0.36m
MongoDB	19.32m	6.94m
PostgreSQL	52.95m	1.19m
jq	330.44m	74.81m

TABLE II: Session execution time with import of data excluded for intermediate user with seed 123.

both systems handle the different dataset characteristics with varying efficiency.

Table II lists the average execution time of a single session, generated with the default user configuration and seed 123 over three runs. The results of the *Twitter* dataset are taken from the previous CPU scalability experiment with 16 cores (109 Gb) and the *NoBench* results from the dataset scalability experiment with $10,000,000$ documents (5.5 Gb). JODA is designed as an in-memory system but can be configured to evict the parsed data from memory after each query. To evaluate the impact on an environment where the available memory is limited, and also to make the comparison fairer, we included numbers for JODA running in this eviction mode. For every query, JODA will re-read the data from the ramdisk, just as the other systems have to. Overall, as we can see, using jq to explore large sets of JSON files is unfeasible. This result is expected, as jq does not import the files into an optimized format but re-reads the input dataset from the filesystem for each query, which causes a substantial I/O overhead. Additionally, MongoDB, PostgreSQL, and jq only use one CPU core, as mentioned earlier. Out of the tested systems, only JODA with in-memory processing can produce tolerable waiting times for interactive data exploration on semi-structured data.

Figure 8 shows the overall distribution of generated predicates for the previous evaluations. In addition, we also generated one default session with seed 123 for the Reddit dataset. As we can see, for the Twitter dataset, the existence and string-type check are used most often. For the given dataset, this makes sense as the heterogeneous structure of the dataset presents the most opportunities for selection. On the NoBench dataset, which will be used later, most of the attributes are of type string that have prefixes with large groups. Hence, the system often chooses them to select sub-datasets. As the Reddit dataset has a fixed schema, no existence predicate was generated.

To further compare all currently supported systems, we created one session for each preset with a seed of 1 and three different configurations:

- **Default:** No aggregation is performed
- **Aggregation (Agg):** All queries aggregate the complete result set with one aggregation function
- **Grouped Aggregation (GAgg):** All queries use `GROUP BY` aggregation functions to aggregate the result set

The sessions have been generated for all three datasets. To reduce the impact of disk operations on the benchmark, we immediately discarded the results of all queries by redirecting them to `/dev/null`. For all systems, we tried to use the optimal way for outputting the results. In JODA, we used the query function for writing results directly to a file (in our case `/dev/null`). The PostgreSQL client was configured to retrieve all results, while we had to write a function for MongoDB which fetches all result documents in batches and uses the `printjsononeline` function to print the result. jq already outputs all results by itself.

Table III shows the execution times of all comparison sessions. Cells marked with a dash timed out after 8 hours. PostgreSQL is missing from the Reddit comparison, as it could not load the dataset because of Unicode issues. As we can see, all systems benefit from aggregating the datasets, which implicates that outputting and writing the result documents is the most expensive step. jq benefits from this the least, as it does not have an optimized built-in aggregation framework, but aggregation had to be implemented in the queries with their built-in scripting language using `map` and `reduce` functions over two jq instances. At the same time, jq has the worst performance of all systems and did not complete the sessions before the timeout for most experiments. The performance difference between the systems is similar to the previous evaluations for all configurations and datasets. As before, the most notable exception is the difference of performance for PostgreSQL on the NoBench dataset, where it is the second best-performing system closely behind JODA.

C. Query Skew

Queries of real users will be skewed towards the most interesting attributes. In Section IV-C we already briefly motivated and introduced a setting to increase the skew towards attributes at the top of the document hierarchy.

Without changing any settings, we can already observe a skew of attributes chosen by the generator. For the preset evaluation, a total of $1,800$ queries have been generated on the Twitter dataset. In these queries there have been $5,267$ references to (405 distinct) attributes, of which 579 ($\approx 10\%$) referenced the top-10 distinct attributes and 986 ($\approx$ 19%) referenced the top-20 distinct attributes. The majority of top-20 attributes consist of attributes that can be used to partition the documents into small subsets, e.g., user names, cities, and URLs. The remaining attributes are either Boolean values or used in existence-predicates to split the documents into two major sets. This shows a clear skew of the generator towards 'interesting' attributes.

Table IV shows the distribution of path depths. The documents column shows the actual depth distribution of all attributes contained in the documents. As we can see, while the depth distribution of the attributes in queries generated with default settings mirrors the distribution of the actual documents closely, it shifts towards the top for queries generated with weighted probabilities.

Config	Novice			Intermediate			Expert			
	Default	Agg	GAgg	Default	Agg	GAgg	Default	Agg	GAgg	
JODA	60m	32s	50s	46m	22s	55s	50m	23s	27s	Twitter
MongoDB	-	1.1h	1.1h	-	24m	26m	-	16m	15m	
PSQL	-	5h	8h	-	1.6h	3.2h	-	1.2h	57m	
JQ	-	-	-	-	-	-	-	-	-	
JODA	3m	2.2s	4s	1.3m	2.4s	23s	1.3m	2.4s	2.8s	NoBench
MongoDB	12m	9m	10m	6.2m	4.7m	4.5m	3m	2.2m	2.4m	
PSQL	6m	1.7m	1.7m	3.3m	52s	2.1m	2.9m	28s	41s	
JQ	1.5h	1.3h	1.8h	48m	39m	1h	24m	20m	30m	
JODA	1.3h	46s	5m	45m	37s	5m	22m	11s	1m	Reddit
MongoDB	-	62m	37m	-	23m	14m	-	12m	9m	
JQ	-	-	-	-	3.7h	-	-	1.6h	3.1h	

TABLE III: Session execution time with import of data excluded for multiple presets and configurations with seed 1

Path Depth	Documents	Queries Default	Queries Weighted Paths
0	0.2%	0.03%	0.0%
1	8.3%	6.2%	8.1%
2	30.7%	32.5%	40.0%
3	40.6%	44.2%	41.2%
4	17.9%	15.8%	10.4%
5	1.9%	1.1%	0.2%

TABLE IV: Distribution of path depths in the original documents and the generated queries

VII. Future Work

To predict the selectivity of generated predicates more accurately, more detailed statistics could be used. For numerical attributes, for example, histograms can capture the distribution of values and prevent wrong decisions due to skewed data. The initial analytics of the dataset could also be included in the generator without the help of external data wrangling tools.

The already supported systems form a strong basis of all kinds of exploration tools, but currently still lacks support for many popular systems. We work on adding more and hope at the same time that BETZE will be widely adopted.

To provide another aspect to benchmarking, we would like to include transformation features into the query generator in the future. These queries would change the structure and content of the dataset as a user would often do. Example transformations could be the renaming, removing, or addition of attributes. To transform the content, many more functions for each data type could be included, like string concatenation or splitting, arithmetic functions, and Boolean algebra. This feature would further challenge the benchmarked systems, as the base dataset cannot simply be used unchanged but would have to be transformed repeatedly with nested queries if no intermediate datasets are supported.

In this work, we focused on describing a benchmark generator to evaluate JSON data stores. The idea of a random explorer that issues sequences of queries against an underlying data store is general enough to be applied on other forms of data, too, specifically also relational data. For single-table relational schemas, this is straightforward and even for multiple tables that are queried independently (i.e., no joins, no union, etc.), BETZE can be employed without major changes to its core functionality. Note that the Reddit dataset used in the experimental evaluation can be considered as relational, but represented in JSON format to be directly usable as a benchmark in this paper. For relational data stored in an RDBMS, the analytical component should ideally be replaced by an RDBMS, too, which is not supported by BETZE yet. Likewise, for graph data, simple filtering tasks could be handled by BETZE, but when considering more meaningful graph queries, like full-fledged SPARQL, the generator needs to be substantially extended.

VIII. Conclusion

In this work, we proposed BETZE, a benchmark generator to evaluate interactive data exploration solutions. We identified common query language constructs and put forward a query generator based on a random explorer model. With this model, it is possible in an intuitive manner to configure the savviness of a user to explore data. We pre-defined three such setups that lead to different query-session characteristics, causing more or less load to be handled by the evaluated systems. The benchmark generator makes use of our JSON processor JODA to analyze data in order to create user sessions. BETZE was created with ease of use in mind and can be used as a library, CLI tool, web interface, and docker image. We also introduced helper scripts that allow creating a benchmarking session with a single command and running this session with all supported systems with another command. We evaluated and validated the characteristics of the different user configurations and provided a performance comparison over four competing systems using the proposed benchmark. The results revealed that even for seemingly simple aggregation and filtering queries, the performance of the investigated systems varies drastically, ranging from response times in seconds to minutes and even hours. We expect that the proposed benchmark will foster the optimization of existing systems and the development of specialized tools, like JODA, to efficiently handle incremental workloads of exploratory queries.

References

[1] A. Nandi and H. V. Jagadish, "Guided interaction: Rethinking the query-result paradigm," *Proc. VLDB Endow.*, vol. 4, no. 12, pp. 1466–1469, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p1466-nandi.pdf

[2] S. Baunsgaard, M. Boehm, A. Chaudhary, B. Derakhshan, S. Geißelsöder, P. M. Grulich, M. Hildebrand, K. Innerebner, V. Markl, C. Neubauer, S. Osterburg, O. Ovcharenko, S. Redyuk, T. Rieger, A. R. Mahdiraji, S. B. Wrede, and S. Zeuch, "Exdra: Exploratory data science on federated raw data," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 2450–2463. [Online]. Available: https://doi.org/10.1145/3448016.3457549

[3] A. Glenis and G. Koutrika, "Pyexplore: Query recommendations for data exploration without query logs," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 2731–2735. [Online]. Available: https://doi.org/10.1145/3448016.3452762

[4] A. Personnaz, S. Amer-Yahia, L. Berti-Équille, M. Fabricius, and S. Subramanian, "DORA THE EXPLORER: exploring very large data with interactive deep reinforcement learning," in *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, G. Demartini, G. Zuccon, J. S. Culpepper, Z. Huang, and H. Tong, Eds. ACM, 2021, pp. 4769–4773. [Online]. Available: https://doi.org/10.1145/3459637.3481967

[5] P. Eichmann, E. Zgraggen, C. Binnig, and T. Kraska, "Idebench: A benchmark for interactive data exploration," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 1555–1569. [Online]. Available: https://doi.org/10.1145/3318464.3380574

[6] G. Doniparthi, T. Mühlhaus, and S. Deßloch, "A hybrid data model and flexible indexing for interactive exploration of large-scale bio-science data," in *New Trends in Database and Information Systems - ADBIS 2021 Short Papers, Doctoral Consortium and Workshops: DOING, SIMPDA, MADEISD, MegaData, CAoNS, Tartu, Estonia, August 24-26, 2021, Proceedings*, ser. Communications in Computer and Information Science, L. Bellatreche, M. Dumas, P. Karras, R. Matulevicius, A. Awad, M. Weidlich, M. Ivanovic, and O. Hartig, Eds., vol. 1450. Springer, 2021, pp. 27–37. [Online]. Available: https://doi.org/10.1007/978-3-030-85082-1_3

[7] M. Stonebraker and E. K. Rezig, "Machine learning and big data: What is important?" *IEEE Data Eng. Bull.*, vol. 42, no. 4, pp. 3–7, 2019. [Online]. Available: http://sites.computer.org/debull/A19dec/p3.pdf

[8] A. Kumar, "Automation of data prep, ml, and data science: New cure or snake oil?" in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 2878–2880. [Online]. Available: https://doi.org/10.1145/3448016.3457537

[9] D. A. Woodie. (2020). [Online]. Available: https://www.datanami.com/2020/07/06/data-prep-still-dominates-data-scientists-time-survey-finds/

[10] F. Eight. (2016). [Online]. Available: https://visit.figure-eight.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport_2016.pdf

[11] M. Vartak, S. Rahman, S. Madden, A. G. Parameswaran, and N. Polyzotis, "SEEDB: efficient data-driven visualization recommendations to support visual analytics," *Proc. VLDB Endow.*, vol. 8, no. 13, pp. 2182–2193, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p2182-vartak.pdf

[12] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer, "Wrangler: interactive visual specification of data transformation scripts," in *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, D. S. Tan, S. Amershi, B. Begole, W. A. Kellogg, and M. Tungare, Eds. ACM, 2011, pp. 3363–3372. [Online]. Available: https://doi.org/10.1145/1978942.1979444

[13] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska, "Vizdom: Interactive analytics through pen and touch," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 2024–2027, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p2024-crotty.pdf

[14] B. Shneiderman, "Response time and display rate in human performance with computers," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 265–285, 1984. [Online]. Available: https://doi.org/10.1145/2514.2517

[15] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska, "How progressive visualizations affect exploratory analysis," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 8, pp. 1977–1987, 2017. [Online]. Available: https://doi.org/10.1109/TVCG.2016.2607714

[16] C. Chasseur, Y. Li, and J. M. Patel, "Enabling JSON document stores in relational systems," in *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, NY, USA, June 23, 2013*, A. Bonifati and C. Yu, Eds., 2013, pp. 1–6. [Online]. Available: http://webdb2013.lille.inria.fr/Paper%2010.pdf

[17] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Networks*, vol. 30, no. 1-7, pp. 107–117, 1998. [Online]. Available: https://doi.org/10.1016/S0169-7552(98)00110-X

[18] N. Schäfer and S. Michel, "JODA: A vertically scalable, lightweight JSON processor for big data transformations," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1726–1729. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00155

[19] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p204-leis.pdf

[20] T. Böhme and E. Rahm, "Xmach-1: A benchmark for XML data management," in *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), 9. GI-Fachtagung, Oldenburg, 7.-9. März 2001, Proceedings*, ser. Informatik Aktuell, A. Heuer, F. Leymann, and D. Priebe, Eds. Springer, 2001, pp. 264–273. [Online]. Available: https://doi.org/10.1007/978-3-642-56687-5_20

[21] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "Xmark: A benchmark for XML data management," in *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 2002, pp. 974–985. [Online]. Available: http://www.vldb.org/conf/2002/S30P01.pdf

[22] T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann, "Anatomy of a native XML base management system," *VLDB J.*, vol. 11, no. 4, pp. 292–314, 2002. [Online]. Available: https://doi.org/10.1007/s00778-002-0080-y

[23] H. Schöning, "Tamino - A database system combining text retrieval and XML," in *Intelligent Search on XML Data, Applications, Languages, Models, Implementations, and Benchmarks*, ser. Lecture Notes in Computer Science, H. M. Blanken, T. Grabs, H. Schek, R. Schenkel, and G. Weikum, Eds., vol. 2818. Springer, 2003, pp. 77–89. [Online]. Available: https://doi.org/10.1007/978-3-540-45194-5_5

[24] (2021, nov) Trec conference website. [Online]. Available: https://trec.nist.gov

[25] N. Fuhr, N. Gövert, G. Kazai, and M. Lalmas, Eds., *Proceedings of the First Workshop of the INitiative for the Evaluation of XML Retrieval (INEX), Schloss Dagstuhl, Germany, December 9-11, 2002*, 2002.

[26] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W. Han, "G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 1099–1114. [Online]. Available: https://doi.org/10.1145/3318464.3389702

[27] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Semant.*, vol. 3, no. 2-3, pp. 158–182, 2005. [Online]. Available: https://doi.org/10.1016/j.websem.2005.06.005

[28] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of RDF data management systems," in *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, ser. Lecture Notes in Computer Science, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. A. Knoblock, D. Vrandecic, P. Groth, N. F. Noy, K. Janowicz, and C. A. Goble, Eds., vol. 8796. Springer, 2014, pp. 197–212. [Online]. Available: https://doi.org/10.1007/978-3-319-11964-9_13

[29] D. Durner, V. Leis, and T. Neumann, "JSON tiles: Fast analytics on semi-structured data," in *SIGMOD Conference*. ACM, 2021, pp. 445–458.

[30] D. Tahara, T. Diamond, and D. J. Abadi, "Sinew: a SQL system for multi-structured data," in *SIGMOD Conference*. ACM, 2014, pp. 815–826.

[31] P. Bourhis, J. L. Reutter, and D. Vrgoc, "JSON: data model and query languages," *Inf. Syst.*, vol. 89, p. 101478, 2020.

[32] D. Florescu and G. Fourny, "Jsoniq: The history of a query language," *IEEE Internet Comput.*, vol. 17, no. 5, pp. 86–90, 2013.

[33] K. W. Ong, Y. Papakonstantinou, and R. Vernoux, "The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases," *CoRR*, vol. abs/1405.3631, 2014.

[34] MongoDB Inc. (2021, jul) Mongodb project website. [Online]. Available: https://www.mongodb.com/

[35] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *SIGMOD Conference*. ACM, 2012, pp. 241–252.

[36] R. Ebenstein, N. Kamat, and A. Nandi, "*FluxQuery*: An execution framework for highly interactive query workloads," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1333–1345. [Online]. Available: https://doi.org/10.1145/2882903.2882945

[37] P. J. Haas and J. M. Hellerstein, "Online query processing," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, S. Mehrotra and T. K. Sellis, Eds. ACM, 2001, p. 623. [Online]. Available: https://doi.org/10.1145/375663.375800

[38] M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska, "Vistrees: fast indexes for interactive data exploration," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, C. Binnig, A. D. Fekete, and A. Nandi, Eds. ACM, 2016, p. 5. [Online]. Available: https://doi.org/10.1145/2939502.2939507

[39] N. Schäfer and S. Michel, "Utilizing delta trees for efficient, iterative exploration and transformation of semi-structured contents," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 1895–1900. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00173

[40] (2021, nov) jq project website. [Online]. Available: https://stedolan.github.io/jq/

[41] (2021, nov) Postgresql project website. [Online]. Available: https://www.postgresql.org/