On Enabling Efficient and Scalable Processing of Semi-Structured Data

Dissertation

vom Fachbereich Informatik der RPTU in Kaiserslautern zur Verleihung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Dissertation

von

Nico Schäfer

Datum der wissenschaftlichen Aussprache	19.05.2023
Dekan	Prof. Dr. Christoph Garth
Berichterstatter	Prof. Dr.-Ing. Sebastian Michel
	Prof. Dr.-Ing. Ralf Schenkel

D-386

Abstract

Semi-structured data is a common data format in many domains. It is characterized by a hierarchical structure and a schema that is not fixed. Efficient and scalable processing of this data is therefore challenging, as many existing indexing and processing techniques are not well-suited for this data format. This dissertation presents a novel approach to processing large JSON datasets. We describe a new data processor, JODA, that is designed to process semi-structured data by using all available computing resources and state-of-the-art techniques. Using a custom query language and a vertically-scaling pipeline query execution engine, JODA can process large datasets with high throughput. We optimize JODA by using a novel optimization for iterative query workloads called *delta trees*, which succinctly represent the changes between two documents. This allows us to process iterative and exploratory queries efficiently. We improve the filtering performance of JODA by implementing a holistic adaptive indexing approach that creates and improves structural and content indices on the fly, depending on the query load. No prior knowledge about the data is required, and the indices are automatically improved over time. JODA is also modularized and can be extended with new user-defined predicates, functions, indices, import, and export functionalities. These modules can be written in an external programming language and integrated into the query execution pipeline at runtime. To evaluate this system against competitors, we introduce a benchmark generator, coined *BETZE*, which aims to simulate data scientists exploring unknown JSON datasets. The generator can be tweaked to generate query workload with different characteristics, or predefined presets can be used to quickly generate a benchmark. We see that JODA outperforms competitors in most tasks over a wide range of datasets and use-cases.

Contents

Chapter 1

Introduction

In recent years, the interest in the JSON [1] file format steadily increased. This semi-structured file format is used in nearly every aspect of computer science. It is easy for humans to read and write JSON documents, while also enabling machines to parse and compose them. Thus, JSON documents are useful for a wide variety of tasks, like scientific experiments, configuration files, and logging. Because of its utility, is it one of the main formats used for interchanging data on the Internet, as the communication between servers and clients, relies on textual data that has to be parsable by the involved systems. Researchers and data scientists may be faced with large amount of—unknown or partially known—documents, which need to be explored and analyzed.

Take, for example, a data scientist responsible for social media interaction at a big company. They are faced with a very large JSON dataset of Twitter tweets. While they may know the general structure of the data by looking at examples and reading the documentation, they may not know the exact structure of their dataset, as the schema changed over the time and it contains many optional attributes depending on the content. In the first step, the data scientist wants to calculate statistics of the attribute distribution to get an overview of the data. After having a good intuition of the data, they want to find out the most common words and hashtags in posts that generated a lot of interaction.

Next, they group the posts by the average sentiment of the replies to the posts, using natural language processing techniques. This information can be used to improve the company's social media strategy by avoiding topics, hashtags, or words that are not well received by the audience.

Finally, the data scientist wants to export the results to give them to the marketing department, so they can use them to create a new campaign. Attached to the exported data, they want to include example tweets that generated the desired interaction so that the marketing department can use them as inspiration.

Performing this complex task requires many operations on the base dataset that may not be known in advance. Some queries may have to be adjusted iteratively until they produce the desired results. Waiting a long time for each result is not feasible, as the data scientist needs to be able to quickly iterate over the data and adjust the queries. Additionally, it is not feasible to infer and maintain the schema of the data in expensive precomputation steps, as the workload involves multiple transformations into intermediate representations.

For these tasks, tools have to be used that are either not designed for these use cases or do not provide the performance needed to explore millions of documents in a timely fashion. Many classic relational database management systems (RDBMS) were adapted with features to support semi-structured data formats like JSON. But these were designed for structured data with a fixed schema and only provide limited functionality. Additionally, they are burdened

by overhead like the ACID paradigm, which is not needed for simple data processing tasks. Thus, these systems are often too slow for the given tasks. As one of the main categories of NoSQL, document stores were created specifically for query evaluation of semi-structured documents. But these document stores are optimized for horizontal scaling, by distributing data over many machines. They often do not make use of the full vertical scale, meaning all available hardware resources, of a given machine to evaluate queries. Furthermore, are they also designed to provide durable storage for documents and often transform them into an optimized internal representation, during a slow import step. As the JSON format is now widely spread, lightweight tools were created for exploration and analysis of JSON files in the terminal. But these tools were often not designed for huge workloads and only make use of a single thread. Thus, often the only choice is to write custom software, which is tailored exactly to the given data set and task. This is time-consuming and may require knowledge of the data and programming languages that may not be available.

1.1 Problem Statement

When faced with a potentially large dataset of JSON documents, whose structure and content may be unknown, a data scientist needs a tool that can efficiently process the data and provide the results of iterative queries in a timely fashion.

We define the following requirements for such a tool:

- Support load, filter, transformation, aggregation, and export operations on the data.
- Scales with available hardware resources to handle arbitrarily large datasets.
- Is easy to use and set up.
- Can be extended with custom code.
- Has subsecond response times for gigabyte-sized datasets.
  - Creates and updates adaptive indices on the fly.

1.2 Contributions & Publications

The contributions of this thesis are the following:

- We introduce JODA, a novel easy-to-use JSON data processor that can process arbitrarily large datasets, using a custom query language, with high throughput by using all available system resources.
- We propose *delta trees*, a novel data structure that can store the results of iterative queries over JSON data in an efficient form by only storing changes to the previous result.
- We describe a holistic adaptive-indexing strategy that iteratively builds and improves structural and content indices on potentially unknown data.

The indices are implemented in JODA to enhance the performance of iterative queries over the datasets.

- We modularize the query execution components of JODA to enable extendability in the form of user-defined modules that allow the integration of external code into the query execution pipeline.
- In the absence of standard benchmarks for exploratory workloads, we introduce the *BETZE* benchmark generator, which generates queries that simulate the behavior of a data scientist exploring a dataset. By generating exploratory query workloads for semi-structured data we can evaluate JODA against competitor systems.

The work presented in this thesis was published in the following peer-reviewed papers and demonstrations:

- We demonstrated JODA at the ICDE conference in 2020 [2] by giving an overview of the system and demonstrating its performance and ease of use.
- *Delta trees* were first introduced at EDBT 2020 [3], where a sketch of the data structure was presented and early results were shown.
- This approach was then further developed and published at ICDE 2021 [4], where the formal model of the delta tree was specified, a more detailed implementation in JODA was presented, and an in-depth performance evaluation was conducted.
- The *BETZE* benchmark generator was introduced at the ICDE conference 2022 [5]. The underlying random-explorer model was presented together with the implementation and an evaluation of the benchmark characteristics was given. Finally, in a preliminary evaluation, the benchmark was used to compare the performance of JODA with other systems.

1.3 Outline

The remainder of this thesis is structured as follows.

In Chapter 2, we give an overview of concepts and technologies that are relevant to understanding this thesis. Relevant research work is additionally discussed in Chapter 3.

Chapter 4 introduces our novel JSON data processor JODA, which can process arbitrarily large datasets with a high throughput. First, we introduce the query language of JODA which also highlights the supported features. Then the architecture of the internal storage, the query execution, and the optimization features are presented.

In Chapter 5, we present our novel data structure, the *delta tree* which can store the results of iterative queries over JSON data in a highly compressed form. The formal model of the delta tree is specified and then implemented in our JODA system. We then evaluate the performance characteristics of the delta tree. By using this data structure, we can significantly reduce the memory requirements and increase the performance of the transformation steps.

To improve the performance of filtering in JODA, we introduce a holistic approach to adaptive indexing in Chapter 6. We introduce an index structure that indexes the structural as well as content information of a JSON dataset and adaptively improves its accuracy and performance during query processing. The performance gains are then analyzed and compared to the original system.

In Chapter 7 we explain the importance of modularizing data processors to enable extendability and flexibility. All major query execution components of JODA will be exemplarily modularized. We then show how the support for user-defined modules can be used to extend the functionality of JODA. We also show example use cases where JODA can be integrated with other systems to leverage the strengths of both systems. In different use cases, we show how JODA can be used to improve workflows it was not originally designed for.

JODA was designed with explorative data analysis in mind. But no benchmark suites exist that are well-suited for this task with semi-structured data. To fill this gap, we created the *BETZE* benchmark generator. Chapter 8 introduces the underlying random-explorer model, simulating a data scientist exploring a dataset. We explain the supported features and queries and describe the architecture of the benchmark generator. JODA is then compared to other data processors using queries generated by the BETZE benchmark suite.

The JODA system is then extensively compared against competitor systems in Chapter 9. Lastly, we conclude our work in Chapter 10 and discuss possible extensions.

Chapter 2

Background

This chapter provides a brief overview of concepts and technologies that are relevant to this thesis. It starts with a short introduction to the JSON data format. It then describes the RapidJSON library, which is used to parse and store JSON documents in JODA. Finally, it gives a short overview of the JSON data processors evaluated in this thesis.

2.1 JSON

Semi-structured documents are used in virtually all fields of computer science. The most well-known semi-structured formats are XML [6] and JSON (JavaScript Object Notation) [1]. They are human-readable open-standard file formats used for storing and transmitting textual data. Both have similar capabilities, storing different data types with associated attribute names. Attributes may be nested within other attributes or stored within a list/array.

The basic building blocks of JSON are atomic values, arrays, and objects. Atomic values may either be a (signed) number in integer or floating-point formatting, strings, Booleans, or the null value. Arrays store a set of elements, where each element may be any of the valid JSON building blocks. An array is written as a comma-separated list of values, enclosed in square brackets. Objects consist of one, or multiple, key-value pairs, combining an attribute name with one valid JSON building block. Objects are enclosed in curly brackets; the keys are separated from the values by a colon, and the key-value pairs are separated by commas. Keys must be strings and must be unique within the object. Listing 2.1 shows one valid JSON document, containing all JSON types.

As the name implies, JSON was derived from JavaScript but has since been adopted by many languages. It is supported by virtually every programming language, either natively or through a library. Its ability to be read and written

```
{
  "number": 5,
  "nested": {
    "array": [
      "string",
      1.1742,
      {"deep": "nest"}
    ],
    "bool": True
  },
  "null": null
}
```

Listing 2.1: Example JSON document with different data types

equally well by humans and machines makes JSON suited for a large variety of use cases. For example, it is widely used as a data interchange format on the internet. Many web APIs use it to receive and send data. Twitter, for example, provides an API to create, retrieve, and delete tweets within their service.

2.1.1 JSON Pointer

To access a specific value within a JSON document, a JSON Pointer [7] can be used. A JSON Pointer is a string that contains a sequence of zero or more reference tokens, separated by a slash character. Each token is either a key, representing an object attribute, or a numerical index, representing an array element. By following the tokens in order, the desired value can be found in the document.

Take for example the JSON document in Listing 2.1 and the JSON Pointer "`/nested/array/1`". The root value of the document is an object. By following the first token, the value of the attribute `nested` is found, which is again an object. The second token references the attribute `array`, which is an array. The third token is now a numerical index, referencing the second element of the array. This element is the floating-point number `1.1742`. As we followed all tokens, we found the value we were looking for.

2.2 RapidJSON

RapidJSON [8] is a C++ library for reading, manipulating, and writing JSON documents. It describes itself as *small*, *complete*, *fast*, and *memory friendly*. It is distributed as a self-contained header-only library, which means that it can be used without any additional compilation steps. JODA uses RapidJSON to parse, manipulate, and store JSON documents.

Figure 2.1 shows a typical workflow using RapidJSON. Given a raw string or character stream, RapidJSON can parse the JSON document into a tree-like structure. This structure is called a *DOM* (Document Object Model). Each node in the tree represents a JSON value. The nodes can be accessed by manually traversing the tree, or by using a *JSON Pointer* to directly access a specific node. The DOM can be modified by adding, removing, or changing nodes. In our example, the value of the `stars` attribute is changed from 10 to 11. The DOM can be traversed completely using a visitor pattern by using the `Accept()` method. Every single node in the DOM will be visited, and the visitor can perform an action on each node. RapidJSON implements a number of visitors, including a *writer* and *pretty writer* that can serialize the DOM back into a JSON string.

Object and array nodes contain an array of children nodes which are sized dynamically. The parser only iterates the string once, which means that the number of children nodes is not known beforehand. Hence, if the array of children is full, it is resized to twice its original size, which may lead to partially empty memory blocks. Every document has a reference to an allocator, which is responsible for allocating memory for the nodes. By default, RapidJSON uses a *Memory Pool Allocator* which allocates memory in chunks using the default

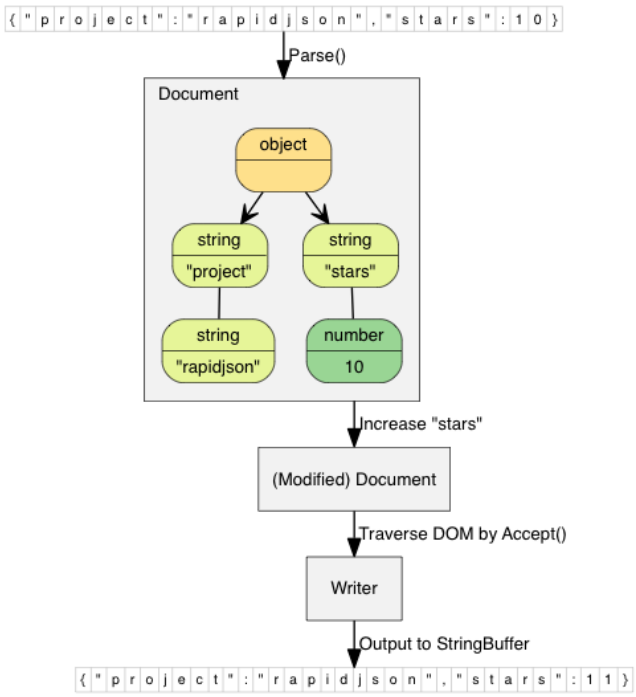


Figure 2.1: Example RapidJSON workflow. Taken from [8].

system allocator. Each document may use its own allocator or share an allocator with other documents. By sharing an allocator, and using a larger chunk size, the amount of expensive memory allocation calls can be reduced.

2.3 JSON Data Processors

As mentioned in Section 2.1, JSON is a widely supported data format. Multiple data processors exist that can be used to process JSON documents. This section gives a brief overview of the data processors that were evaluated in this thesis and the reasons for their selection.

2.3.1 JQ

jq [9] is a command-line JSON processor. It is a small program written in C and is available for most operating systems. Similar to `grep`, `sed`, and `awk`, jq evaluates a fixed expression on a given input. But unlike these tools, jq parses the input as JSON and can use the structure and content of the documents. In the most basic form, jq can be used to extract a specific value from a JSON document. Similar to JSON Pointer, jq uses a path expression to access a value. In this case, every token is separated by a dot and array indices are enclosed in square brackets.

Storage
jq works on raw JSON data, either piped into the program or read from a file. It does not provide the means to store the data in an internal representation for later use. The result of each query is printed to the standard output and has to be redirected to a file or piped into another program if it is to be stored.

Query Capabilities
jq uses a custom query language based on multiple chained *filters*. Each filter takes the input, performs some operation on it, and passes the result to the next filter. Filters can make use of many inbuilt functions, such as lookups, mathematical operations, string manipulation, search, sort, map, and reduce. This allows for a wide range of operations to be performed on the data. jq does not provide any straightforward way to aggregate data, but it is possible to use the reduce functions or multiple chained jq invocations to achieve this.

Indexing & Query Optimization
As jq only performs single-pass queries over an input, it does not provide any means to index the data. It also does not perform any query optimization.

Performance
jq is a command-line tool that works on raw data and performs all operations in a single main thread. This means that it is not able to make use of multiple cores, indices, or in-memory data structures. Hence, compared to the other data processors, jq is the slowest, as Chapter 9 shows.

Setup
Being a command-line tool without any dependencies, jq is easy to install. Virtually every Linux distribution provides a package for jq. Alternatively, the program can also be compiled from the source, or precompiled binaries can be downloaded for all major operating systems. It can also be tested online in a web browser without any installation. No setup is required in any form for jq to work.

2.3.2 PostgreSQL

PostgreSQL [10] is a relational database management system (RDBMS). It is a mature and widely used database that is available for most operating systems. PostgreSQL is a full-featured database that supports a wide range of features, including transactions, views, and triggers. It supports JSON data in the form of `json` and `jsonb` data types. The `json` type stores the JSON document in its textual form without any additional processing. `jsonb`, on the other hand, stores the JSON document in a decomposed binary format that is optimized for querying and can be indexed. Both data types are augmented with a set of operators and functions that allow for querying and manipulation of the data. While other RDBMS like MySQL [11] also support JSON data, we focus on PostgreSQL as it is a popular and mature database that is widely used in production environments as well as research.

Storage
PostgreSQL is a full-featured database with persistent storage. Its main purpose is to store and query data. It is not designed to be used as a temporary data store or processor. The data is stored in a relational format, which is not the

most efficient for storing JSON documents. However, when using the `jsonb` data type, the data is decomposed and stored in a more efficient format. If the user wishes to reuse the intermediate results of a query, they can be stored in a temporary table or view.

Query Capabilities
SQL is a powerful query language that allows for a wide range of operations. Filtering, sorting, grouping, and aggregation are all supported. Many functions are implemented, that query and manipulate the JSON data types. These can be integrated into all parts of the SQL query.

Indexing & Query Optimization
If knowledge about the JSON data is known in advance and the data is stored in the `jsonb` format, it is possible to create indices on the data. These indices can either be created on the structure of the JSON or on the content of specific attributes. The full power of the RDBMS query optimizer can then be used to optimize the query execution.

Performance
PostgreSQL is a full-featured database that adheres to the ACID properties. It is designed to be durable and consistent, which means that it requires a lot of overhead to ensure that the data is stored correctly. This overhead is not present in the other data processors. As a result, PostgreSQL is one of the slower systems evaluated in this thesis when it comes to importing data. When using the `jsonb` data type, the import takes even longer, as the data has to be decomposed and translated. However, once the data is imported the query performance is comparable to the other data processors.

Setup
Installing PostgreSQL is easy on most operating systems. The database can be installed from the package manager of the operating system or downloaded from the official website. But correctly configuring and setting up the database is a more involved process, where users have to be created, permissions have to be set, and the database has to be initialized. For simple temporary data processing tasks this is unnecessary overhead. Importing JSON files into PostgreSQL can be achieved using the `COPY` command. But files that work with all other data processors may not work with PostgreSQL, as it does not allow some special characters in the JSON data. Importing this data then requires an additional preprocessing step.

2.3.3 MongoDB

MongoDB [12] is a NoSQL database that stores data in the form of JSON documents. It is designed to scale horizontally and supports ad-hoc querying and indexing of the data. Documents are parsed into the *BSON* format, which is a binary representation of JSON. The emphasis of MongoDB is on storing and querying data in a flexible and distributable way.

Storage
Documents are stored in persistent collections, which are similar to tables in relational databases. The data is stored in a binary format using the WiredTiger storage engine. It uses document-level concurrency control to ensure that the data is consistent. Additional caches are used to improve the performance of the database.

Query Capabilities
MongoDB exposes its query language through a JavaScript API that allows for ad-hoc querying of the data. The database provides an object for every collection that can be used to query the data. Simple queries can be performed using the `find()` function, which takes a predicate and returns an iterator over all documents that match the predicate. More complex queries can be performed using the `aggregate()` pipeline, which takes a list of stages that are executed in order. The result is again provided as an iterator over the documents. Many functions are provided to manipulate the data, such as filtering, sorting, grouping, and aggregation.

Indexing & Query Optimization
If knowledge about the data is known in advance, it is possible to manually create indices on the data. Multiple index types are supported, including compound indices, text indices, and geospatial indices. These indices are used by the query optimizer to reduce the number of documents that have to be read from the disk.

Performance
Our experiments in Chapter 9 show that MongoDB is one the slowest system under evaluation. Especially the import of data takes a long time. But the query performance is also not as good as the other systems under consideration.

Setup
MongoDB is available as a package for most major operating systems. Once started, it can be immediately used with the default configuration. Importing JSON files requires the usage of an external tool `mongoimport` that is provided with the database. While this process takes a long time, it is easy to use and does not require any additional configuration.

2.3.4 Spark

Spark [13] is a distributed general-purpose data processing framework. It is designed for large-scale data processing tasks distributed over multiple machines. The data is stored in RDDs (Resilient Distributed Datasets), which are immutable collections of data. The user interacts with the system using a high-level API written in Scala, Java, R, or Python. All interactions with RDDs are translated into a directed acyclic graph where the nodes are RDDs and the edges are operations. The graph is then executed in parallel on a cluster of machines as soon as the results are required.

Storage
Data is stored in RDDs that are distributed over the cluster. Spark does not aim to provide persistent storage for the data, but rather a framework for processing. By default, RDDs only store data in memory. But they can also be configured to serialize the data to disk if not enough memory is available.

Query Capabilities
Spark is deeply integrated with their supported programming language. It is often possible to use native functions of the language to manipulate the data. This enables a vast range of operations that can be performed on the data. Additionally, Spark includes general operations that allow iterating, mapping, and aggregating data in an RDD. This flexibility comes at the cost of usability, as the user has to be familiar with the programming language and the Spark API. Spark also allows the usage of *DataFrames*, which are similar to tables in relational databases. They have a fixed schema and can be queried using a SQL-like syntax.

Indexing & Query Optimization
As Spark is not a data management system, it does not provide any indexing features. The user has to ensure that the data is stored in a way that is efficient for the query. While indexing is out-of-scope for Spark it still tries to optimize the query execution using late materialization. As mentioned before, RDD interactions are translated into a directed acyclic graph. But no actual execution takes place until the final results are required. Hence, multiple queries can be posed to the system without actually executing them. Only when a result tuple is fetched, the required nodes of the graph are executed.

Performance
Spark is designed to be used on large-scale data processing tasks under the usage of all available system resources. As a result, it is one of the fastest systems under consideration.

Setup
Installing and setting up Spark is a complex process. The system is designed to be used on a cluster of machines, which requires a lot of configuration. The user has to install libraries for the programming language, Spark itself, and set up the environment. If used in cluster mode, Spark also needs a cluster manager. But even in local mode, configuring spark to use all available resources is not trivial. Connecting to Spark and using the API also requires extended knowledge of the programming language.

Chapter 3

Related Work

In this chapter, work related to JODA and its components is discussed. First, we give an overview of current research in JSON data processing systems. Then we discuss existing work related to our delta trees approach, shown in Chapter 5. Next, current adaptive indexing approaches are discussed. We then give an overview over user-defined modules in different systems. Lastly, we discuss current benchmarking approaches for JSON data processing systems.

3.1 Research in JSON processing

The support for the JSON file format is rapidly increasing. Many systems are now able to handle JSON files, either as input or output. In this section, we provide an overview of the current state of research for JSON data processing systems.

3.1.1 JSON in RDBMS

Classic relational database management systems (RDBMS), like MySQL and PostgreSQL, use fixed-schema relations to store data. This data is distributed over multiple relations, each having fixed columns of specific data types. The data is represented by rows within the relations, with values for each column. As the data type of a column is known, the data can be stored and retrieved efficiently in binary format.

RDBMS are the most commonly tool used for data storage and analysis. Most of these systems use SQL (Structured Query Language) to alter and query the stored data.

With the emerging success of semi-structured data, many RDBMS were extended with support for such file types. These systems are able to store XML or JSON documents in special text columns, checking the validity of the document upon insertion. Supporting functions are implemented, for querying and modifying these columns.

As mentioned in Section 2.3.2, PostgreSQL and MySQL both support JSON columns out of the box. But the past years have witnessed various approaches that try to incorporate deeper JSON support into existing relational query engines [14, 15]. More specifically, in their very recent work, Durner *et al.* [14] 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.

3.1.2 Document Stores

In recent years, NoSQL systems rapidly gained in popularity. The abbreviation NoSQL stands for "non SQL", "non relational", or "Not only SQL" and describes a collection of database systems, not adhering to the classic relational model.

NoSQL systems are characterized by their focus on horizontal scaling, meaning that data is distributed over multiple instances. Queries are then evaluated on all instances simultaneously, and the result is collected. In addition to distributing data, it may also be replicated on more than one instance to improve failure safety.

These systems mostly implement weaker consistency models than the ACID transaction paradigm. Some systems replaced it with the BASE model, which stands for "Basically Available, Soft state, Eventually consisten" [16]. By not adhering to the ACID model, higher performance and better horizontal scalability can be achieved.

There exists a large variety of NoSQL systems split into different categories. Some of the most notable categories of NoSQL systems are:

- **Key-Value Stores**, storing data—as the name implies—as simple key-value pairs.
- **(Wide) Column Stores**, which, like RDBMS, store data in columns within tables. But unlike relational databases, each row may have a different set of columns. Thus, data with sparse columns can be stored more efficiently.
- **Graph Databases**, representing their data as graph structures with nodes and edges.
- **Document stores**, storing semi-structured documents in collections.

There are many document stores or document-oriented databases. Each with its own set of features and supported types. Some document stores like CouchDB and MongoDB are specialized on JSON documents, while others, like BaseX and eXist use XML. There are also multi-type document-oriented databases, like the Clusterpoint Database. These systems may either store the documents in textual representation or in a specialized binary format.

MongoDB [12] is one of the most well-known document stores. It uses the BSON (Binary JSON) [17] format as the underlying storage model. Documents may be split into shards and distributed over many instances. This can be used to provide load balancing, replication and distributed querying. The binary format enables the platform to efficiently query attributes and to create a wide range of indices. But using the BSON format comes at the cost of increased import times, as these documents first have to be parsed and decomposed into this format. Exporting JSON files also requires a special exportation step to recreate a human-readable document.

As document stores are mostly optimized for distributed deployment, the setup of such database systems tend to be complex. Furthermore, are they designed, like RDBMS, to persistently store documents and enable concurrent queries by different users. This overhead, combined with the complex setup step and

potential slow import/export of documents, make document stores unsuitable for exploration and analysis or unknown data.

3.1.3 Others

There exist many tools and languages supporting JSON files, in addition to the previously discussed database systems.

Most programming languages support JSON either directly or through third-party libraries. This extensive support for the format only helps its increasing popularity. Many programs and web services provide a JSON-based API, which makes them compatible with all these languages.

Programmers familiar with any of the supported languages may use them to explore and transform JSON files with custom programs. But creating such a program not only requires knowledge of the language but in most cases also knowledge about the data itself. Additionally, is it not always trivial to write these programs in such a fashion, that the evaluation of the documents is performant and scales well with increasing amounts of data.

The need for simple JSON processors is a recognized one, as tools like jq [9], which provide command-line JSON processing capabilities, are integrated into the standard repositories of many Linux operation systems. This tool enables the user to pipe JSON text into the program and process these documents using a query language. The output can itself be piped into other processes, thus providing JSON capabilities to the system. But usage of such tools is often confined to single threads or other restrictions. This makes them apt for exploring small document sets but does not scale well if the exploration and analysis of millions of documents may be required.

As an alternative to classical RDBMS, Alagiannis *et al.* proposed a new paradigm for database systems, called NoDB [18]. They created an extension for PostgreSQL, which is able to evaluate queries in situ on raw data files, thus skipping the import step. They compared their implementation against common RDBMS and concluded that NoDB systems can achieve competitive performance with traditional database systems.

3.2 Delta & Change Management

Change detection in hierarchical data, which is often based on tree edit distances [19, 20], is a well-studied topic [21]. XML trees are a special case of hierarchical data, for which many approaches have been created [22, 23]. One of the most cited approaches is X-Diff by Wang *et al.* [24]. They propose an algorithm to find changes in XML trees by only using ancestor relationships. Most algorithms created for XML can be adapted to work with JSON trees, as they have a similar hierarchical structure and path expressions. While these approaches aim at identifying differences between two given trees, our approach wants to store changes without duplicating any data.

Almeida *et al.* proposed to create a map as a Conflict-free Replicated Data Type (CRDT) [25], which can be synchronized by sending delta changes of the modification action to replicated instances. This data type can also be nested

and JSON documents can be translated to and from nested maps. But the computational overhead required for conflict-free synchronization is too large for our use case.

In software development, version control systems (VCS) are often used to manage source code. These systems enable developers to have a full history of all changes made to a given code base. Some VCS always store the complete file when a new version is checked in, which requires a lot of memory, as for each change redundant data has to be stored. Delta-based VCS only store the whole file on first check-in and use delta encoding to manage changes. This delta-encoding reduces the required storage, but has the drawback of a more expensive retrieval operation. Popular VCS that use some kind of delta-encoding include Git [26], SVN [27], and CVS [28]. Similar methods are also used by command line tools like Diff [29] and standards like VCDiff [30], to find deltas in text files. But most, or all, of these approaches only use the textual or binary representation and do not exploit the structure of semi-structured documents.

Some database systems create query plans that decide between late and early materialization of intermediate query results [31]. Early materialization means that result tuples are constructed immediately and then processed further by remaining operations or queries. On the other hand, late materialization tries to push expensive operations as far back as possible, to potentially reduce the required work of result tuple construction. In our case, early materialization is similar to the default approach of immediately constructing and storing the result documents. The approach we are suggesting is a compromise between early and late materialization, by immediately materializing the changes of a document, while deferring the potential construction of a complete result document to a later point in time. the changes of a document, while deferring the potential construction of a complete result document to a later point in time.

3.3 Adaptive Indexing

Semi-structured documents are often nested and mostly do not possess a strict schema, which can be used to create indices. However, one can, in principle, interpret a path in a document as a column name and use it for indexing. The problem however is that not every document may contain the given path and different documents may store a different data type. There has been previous work on using the structure of semi-structured documents for indexing. For example, Kaushik *et al.* [32] described a structural index that uses the XML data structure by transforming it into a graph. The approach collects all available paths in the documents and creates a "summary graph". The graph does not contain any content nodes but only represents the structure of a set of documents. Each document is then indexed by storing the level in the graph and the XML document ID in the node, for usage in queries. Additionally, Chung *et al.* [33] created a similar summary graph that adapts to the current workload. Only frequently accessed paths will be kept in the graph.

Shukla *et al.* [34] introduced a similar approach for the JSON document database Azure DocumentDB. A JSON document can be interpreted as a tree. DocumentDB will index, all paths of all documents by creating a new tree that merges all possible paths and values. For example `/user/name: "Mike"` can

be thought of as a tree with three nodes: `"user"` → `"name"` → `"Mike"`. Since nodes may only be available in specific documents, each node is associated with a set of document references. We extend this idea by building sub-trees on-demand and augmenting this structure index with additional content indices.

Kissinger *et al.* [35] argue, that the trend of growing data means that a column index used in traditional, relational systems, contains a considerable amount of data that is not needed. Many techniques to improve indexing are still based on creating and dropping an index, while it should be more closely related to the data that gets queried. They introduce SMIX as an exemplary partial column index. SMIX is based on pages, the internal basic data blocks used by most SQL systems to store data. When a value is queried, it gets indexed in a separate tree data structure. For each page, the index keeps track of the fraction of indexed values. With subsequent queries, more data is indexed. For almost indexed pages, SMIX can decide to index them fully. All pages which are fully indexed can then be skipped by the next query and only rely on the index. SMIX can also be restricted in the amount of memory or storage it may use. To enforce these quotas, SMIX may evict data on a per-page basis. Similarly, our system also contains partial indices, which may be dropped when memory resources are used up.

Instead of changing the types of indices, to support adaptive or automatic indexing, Das *et al.* [36] created a system on top of traditional SQL systems, which analyzes historical user queries and creates indices fitting for the previous query loads. They presented an automatic indexing service that is supposed to improve millions of databases in the cloud without human interaction. It is implemented with a recommender system that evaluates user input and data to decide if an index on a column should be created. Since it is sometimes hard to estimate the value of an index, a validator system compares performance before and after the index creation to detect wrong decisions and adapt in case of query load changes.

Instead of deciding which index should be created after multiple specific queries to certain data, Arulraj *et al.* [37] proposed "predictive indexing". A machine learning approach, which predicts what kind of index is needed next. This way, the database can apply measures earlier to improve performance. They introduced a "hybrid scan" mechanism to execute queries with support for partial indices, which execute an index scan on indexed pages, and a table scan on the non-indexed pages.

Cracking databases, as proposed by Idreos *et al.* [38], Adaptive Merging, as introduced by Graefe *et al.* [39], and the hybrid approach [40], are examples of indices that are built at query execution time and improved on each use. While database cracking is very performant for the first query, it converges very slowly to a full index. Adaptive merging is the opposite; the first query is slower than cracking, but the index converges faster. The hybrid approach tries to combine the two approaches to find a way to achieve a more balanced adaptive strategy. Recently, Holanda *et al.* [41] introduced a novel progressive indexing technique, which has similar goals and use cases as our approach. They created a system, which works for traditional SQL systems and may be configured with an indexing budget that can be spent on creating and maintaining incremental indices.

3.4 User-Defined-Modules

There exist a plethora of work on augmenting (relational) database systems with features around user-defined functions. Gupta and Ramachandra [42] investigate and report on the results of executing procedural extensions in an RDBMS. Through the analysis of stored procedures, triggers, and user-defined functions, they outline the limitations of extensions and possible opportunities to improve their execution. Friedman et al. [43] present a framework, called SQL/MapReduce (SQL/MR), to implement user-defined functions making it possible to include them as SQL sub-queries, without the limitations of poor parallelizable execution and the definition of input and output specification. Crotty et al. [44] describe a novel architecture that automatically compiles workflows of user-defined functions, which include more complex operations such as selection or join. Hellerstein et al. [45] present an open-source library allowing the incorporation of SQL-based algorithms for machine learning, data mining and statistics run within a database engine, such as Postgres. Passing et al. [46] asses different possibilities of including data analytics in database systems, and allow for the integration of analytical operators directly in SQL, using their novel user-defined lambda expressions. Schüle et al. [47] use the PostgreSQL just-in-time compilation feature to allow for user-written lambda functions and they demonstrate them in combination with data mining algorithms. Others [48, 49] allow for interpretation of procedural programs as subqueries by an SQL engine, by transforming them entirely to recursive CTEs. Recently, Sichert and Neumann [50] present user-defined operators allowing the inclusion of custom algorithms from a programming language of their choice into an existing database system, while still preserving the ACID properties. They test UDOs in Postgres and Umbra and show that their execution is as efficient as regular queries.

Differently, Boehm et al. [51] introduce an open source system for end-to-end execution of machine learning models including data preprocessing, model training, as well as debugging. Similarly, Schüle et al. [52] make use of database systems for data inspection and provide full support for running end-to-end pipelines including model training and testing.

3.5 Benchmarking

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 benchmark 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 [53], the transaction processing council, and 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 [54].

With the increasing popularity of semi-structured data format XML throughout the early 2000s several benchmarks [55,56] have been proposed to evaluate XML databases [57, 58], that implemented special query patterns like path queries, reflecting the characteristics of the XML data model. In the information retrieval domain, the TREC [59] conference is a 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 [60] 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.* [61] proposed the NoBench data generator to create benchmarking datasets of variable size.

There exist many benchmarks for specialized workloads. For example, G-CARE [62] 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 [63], a benchmark for OWL knowledge base systems. They propose a standardized ontology with a set of benchmark queries concentrating on certain characteristics. Similarly, WatDiv [64] 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.* [65], who proposed IDEBench to benchmark interactive data exploration on relational data. Our benchmark generator has a similar goal for semi-structured documents and extends it with iterative queries.

3.5.1 Query Suggestion and Online Processing

Related to interactive data exploration is also work around query suggestion [66, 67], 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 [68] work on online processing [69] and approximate query processing can be effective. To improve the exploration capabilities of databases, El-Hindi *et al.* [70] created VisTrees, a multi-dimensional index for interactive computation of histograms. In our own JODA system, we implemented Delta trees [4], to improve performance for iterative queries, as often found in exploratory query workloads.

Chapter 4

JODA—Concepts & Architecture

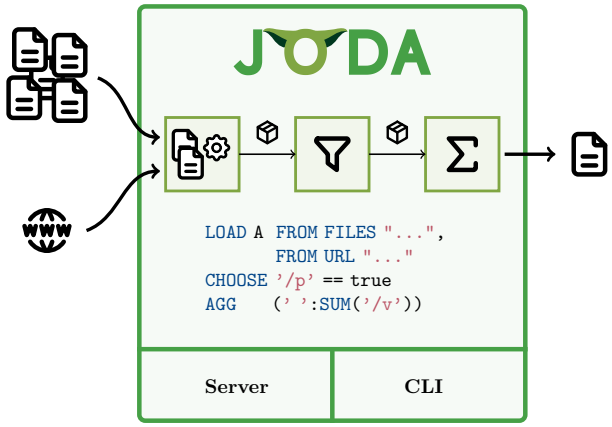


Figure 4.1: Overview over the JODA system

To solve the challenges of semi-structured data exploration, processing, and analysis, we introduce JODA, a novel data processor for JSON data. JODA is a data processor that is designed to be fast, flexible, and easy to use. It is written in C++ and employs modern techniques to achieve high performance. The core philosophy of the JODA design is to create a system that is scalable to all sizes of data and optimally utilizes all available system resources. To achieve this, JODA performs as much work as possible in parallel and in memory. This includes the parsing of the data, the execution of queries, and the storage of the data.

Figure 4.1 shows a high-level overview of the JODA system. At the core of the system, we have our pipeline-based query execution engine. Every query is transformed into separate tasks that are connected using queues until they form a single query pipeline. Data flows from one task to the next, until the final result is produced. JODA can import data from a variety of sources, including files, websites, and streams.

To parse and interact with the data, JODA uses the fast *RapidJSON* library [8]. The parsed documents are then stored in *collections*, which are themselves partitioned into independent *containers*. Each container includes all data necessary to execute a query on this given partition. This allows completely independent and parallel execution of queries on the data.

The user can interact with JODA by either including it as a library in their own application, by using the JODA command line interface, or by starting a JODA instance as a server and communicating with it via HTTP.

```
LOAD FROM FILE "Twitter.json"
CHOOSE '/user/verified' == True && '/user/followers_count' > 1000
AS (''':'/user'),
   ('/popularity':
      SUM('/user/followers_count','/user/friends_count'))
AGG (''':GROUP AVG('/popularity') AS avg_popularity BY '/lang')
STORE AS FILE "popular_verified.json";
```

Listing 4.1: Example query calculating the average popularity by language for all verified users with a minimum followers count

4.1 Query

To support all features that JODA provides, and present them in a way that is easy to understand and use, we created a custom query language for JODA. Queries in JODA consist of multiple straightforward statements representing stages in the execution pipeline. Figure 4.2 shows the general syntax of a JODA query.

Each stage passes its result on to the next stage, as can be seen in the example query in Listing 4.1. The `LOAD` statement loads the data to be used for the remaining stages. Using `JOIN`, data from another collection or query can be joined with the current data. The `CHOOSE` statement filters documents and passes on all documents that match the given condition, which can be transformed using the `AS` statement. All documents in the pipeline can then be aggregated using the `AGG` keyword. Finally, the resulting documents can be stored or exported using `STORE`.

4.1.1 Collection

All documents in JODA are stored in collections. A collection is comparable to a table in a relational database or a collection in MongoDB.

Named collections are created by the user using queries. But queries that do not specify any collection name will still create internal temporary collections that can not be reused in different queries.

Collections are completely schemaless and do not restrict the number or types of documents that can be added to them.

4.1.2 Pointer

Pointers are used to reference attributes within a single document. In JODA, a pointer is denoted by a path string surrounded by single quotes. JODA pointers use the same syntax as JSON-pointers [7]. The path string is a sequence of attribute names or index numbers separated by a slash. Starting from the document's root, each path string segment is used to traverse the document tree. For example, the pointer `'/a/1'` would point to the value 2 in the document `{"a":["x",2,3]}`.

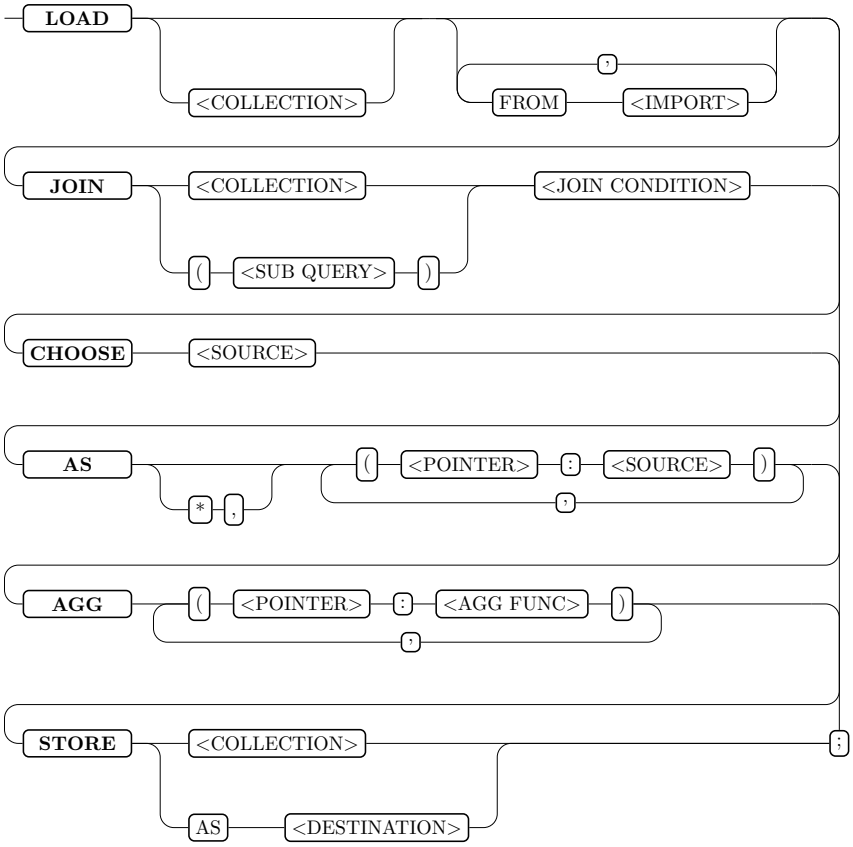


Figure 4.2: JODA query syntax

4.1.3 Source

To access and modify data of documents in JODA queries, *sources* are used. A source can either be a pointer, as described in Section 4.1.2, or a function. JODA supports many different functions that provide means of accessing metadata, modifying existing data, or creating new data. For example, the `LEN()` function can be used to get the length of a string, or the `CONCAT()` function to concatenate two strings. Appendix A lists all currently supported functions in JODA.

Sources can also be nested to create more complex expressions. The expression `LEN(LTRIM('/text'))` would first trim the leading whitespace of the attribute `text` and then return the length of the resulting string.

In addition to prefix notation, JODA also supports infix notation for most Boolean and arithmetic operators. For example, the expression `'/num' > 0` is equivalent to `GT('/num',0)` and checks whether the number contained in the attribute `num` is greater than zero.

4.1.4 Loading Data

Every query in JODA starts with a `LOAD` statement. It specifies on which data the rest of the query should be executed. If a collection name is specified, the data is loaded from the internal JODA data store. Data can also be loaded from external sources, like files or URLs, by providing import clauses. A `LOAD` statement requires either a collection name or at least one import clause. If both are defined, the imported data is first added to the internal collection, and then the collection is used for the rest of the query. By storing the data in the internal data store, it can be reused in future queries.

Each import clause starts with the `FROM` keyword, followed by the source type and additional parameters, as shown in Figure 4.3. The source types `FILE`, `FILES`, and `URL` can be used to load data from single files, directories, or URLs, respectively. JODA always reads a file or URL until the end and parses any valid JSON document it finds. These may be atomic values, objects, or arrays. By specifying the `LINESEPARATED` parameter, JODA expects each line to contain a single JSON document. While parsing these kinds of files also works without the parameter, it is more efficient to specify it. JODA can also load a partial dataset by specifying a `SAMPLE` parameter. It expects a floating point number between

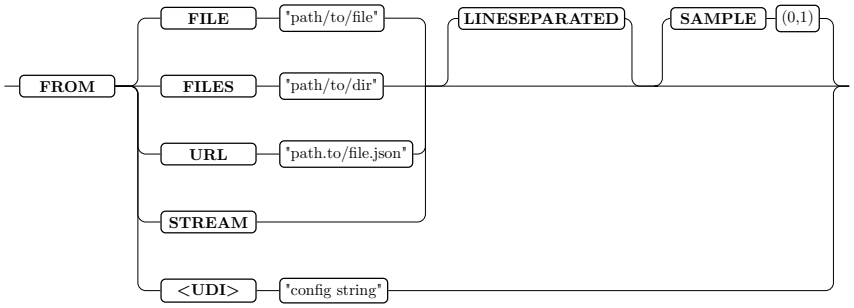


Figure 4.3: Syntax of import clause

```
LOAD FROM FILE 'data.json' LINESEPARATED SAMPLE 0.5;

LOAD Twitter FROM FILES 'twitter',
   FROM URL "https://api.twitter.com/1.1/tweets.json";

LOAD Twitter;
```

Listing 4.2: Example queries showcasing different `LOAD` usecases

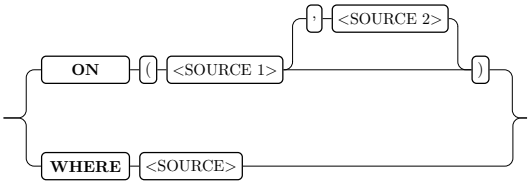


Figure 4.4: Syntax of join condition

0 and 1, which specifies the fraction of the data that should be loaded. This is useful for fast queries on large datasets where no exact results are required.

In addition to the previously mentioned source types, JODA also supports loading data from the input stream. The same options as for files and URLs are supported. Streaming data into JODA is only supported in non-interactive environments, like the execution of a query or query file provided as a command line argument.

Finally, user-defined import modules (UDI) can be used to load data from custom sources. UDIs are always configured with a single string parameter. As they return finished documents instead of using the JODA parsing pipeline, no additional options like `LINESEPARATED` or `SAMPLE` are supported.

Listing 4.2 shows multiple example queries that load data from different sources. The first query loads half of the documents stored in the local `data.json` file, which contains line-separated JSON documents. The second query, on the other hand, loads all documents from the `twitter` directory and queries the Twitter API for additional data. Both sources are stored in a collection called `Twitter` for future use. The third query then loads the previously imported `Twitter` collection again.

4.1.5 Joins

Two collections can be combined using the `JOIN` statement. As seen in Figure 4.2, the data of the `LOAD` statement can be either joined with another, already existing, collection or with the result of a subquery. The data of the `LOAD` statement is called the *left* join partner, while the collection or subquery is called the *right* join partner. All *left* documents are collected and then joined with each *right* document. For best performance results, the *left* collection should be the smaller one. The result of the join will be one JSON object for each joined pair of documents, containing two attributes *left* and *right*.

```
LOAD Users
JOIN Twitter ON ('/id', '/user/id');

LOAD Users
JOIN Twitter
  WHERE '/left/id' == '/right/user/id';

LOAD Twitter
JOIN (
  LOAD FROM URL 'languages.json'
  ) ON ('/lang');
```

Listing 4.3: Example queries showcasing different `JOIN` usecases

Which document is joined with its partner is determined by the join condition. As can be seen in Figure 4.4 there exist two types of join conditions: *equality* and *theta*. Equality joins, denoted by the `ON` keyword, are used to join two collections based on the equality of attributes. If only one source is specified in the `ON` clause, the value is evaluated in the *left* and *right* join partner and then checked for equality. If two sources are specified, each join partner is evaluated with its own source.

For more complex join conditions, JODA supports theta joins. Two documents are joined if the specified source evaluates to `true`. To be able to access both join partners in the theta-join condition, the system has to calculate the cross-product by joining all pairs of documents. Then the join condition is evaluated, and only the pairs that match are kept. This makes theta joins significantly slower than equality joins.

The first two queries in Listing 4.3 will have the same result, but one uses an equality join and the other one a theta join. An existing *Users* collection is joined with the *Twitter* collection based on the equality of user ID. The third query shows the usage of a subquery, which fetches an external dataset from a URL and joins it with the *Twitter* collection based on a common attribute.

As Section 4.2 explains, JODA splits the data into multiple partitions, called *container*. Within the join executer, each container of the left collection is first stored in a list. Then each container in the right collection is sent to the executor, which performs a pair-wise join with every container in the list. The system performs a nested loop join over each document in the containers to join two containers.

4.1.6 Filtering Collections

Collections can be filtered using the `CHOOSE` keyword. The source is then evaluated for every single document in the collection. If the result is a Boolean `true`, the document is passed on to the next step; else, it is filtered out. The type of the source has to be either a Boolean or a pointer to a Boolean attribute. No *truthy* conversion of other types is performed automatically. If the user desires a conversion, the `TRUTHY` function can be used. If no `CHOOSE` statement is given, all documents are passed on. Listing 4.4 gives example `CHOOSE` conditions.

```
LOAD Twitter CHOOSE EXISTS('/text') && '/lang' == "en";

LOAD Twitter CHOOSE TRUTHY('/user/follower_count');
```

Listing 4.4: Example queries showcasing different `CHOOSE` usecases

```
LOAD Twitter AS ('/text':'/text'), ('/length', LEN('/text'));

LOAD Twitter AS *, ('/text':SUBSTR('/text',0,10));
```

Listing 4.5: Example queries showcasing different `AS` usecases

4.1.7 Transforming Collections

After the filter step, collections can also be transformed into a new format. Using the `AS` keyword and one or multiple transformation tuples, each source document is transformed to have the desired structure and content. Every tuple consists of a pointer and a source. The pointer denotes where the value should be written in the new document, while the source denotes which value should be written. The source is evaluated against the source documents. Tuples are evaluated in order and can override previously written values.

As it is often desirable to only change a part of a document—e.g. replacing a name with initials or calculating a new numerical value—JODA provides a shortcut to copy the whole source document. Instead of a tuple, the first term after the `AS` statement can also be a `*`, which is syntactic sugar for the tuple `('':'')`. All successive tuples will then add to the source document or replace attributes. If no `AS` statement is present in the query, the documents are passed on as-is.

The first query shown in Listing 4.5 takes the set of twitter documents and simplifies it by extracting only the *text* attribute, and adding a *length* attribute containing the string length. The second query uses the `*` function to copy the whole tweet object but afterward truncates the *text* string to contain only ten characters.

4.1.8 Aggregating Collections

JODA also supports aggregation queries, which create a single document from a collection. Similar to the `AS` command, aggregation is performed using the `AGG` keyword, followed by a list of tuples. Each tuple consists of a pointer and an aggregation function. Again, the pointer denotes the location of the new value, which in this case, is given by the aggregation function. Every aggregation function is evaluated independently, and the result is written in a single document in the order of the tuples. The aggregation function parameters are arbitrary sources with full support for pointers and functions.

Each aggregation tuple can also be grouped and aggregated separately, like the `GROUP BY` statement in SQL. To group an aggregation function, the second part of the tuple has to start with `GROUP <AGG Function> <As> BY <Source>`. For

```
LOAD Twitter AGG ('/min_length', MIN(LEN('/text')));

LOAD Twitter AGG ('':GROUP COUNT('/text') AS count BY '/lang');

LOAD Twitter AGG WINDOW(100) (''':GROUP COUNT('/text') AS count BY '
    /lang');
```

Listing 4.6: Example queries showcasing different `AGG` usecases

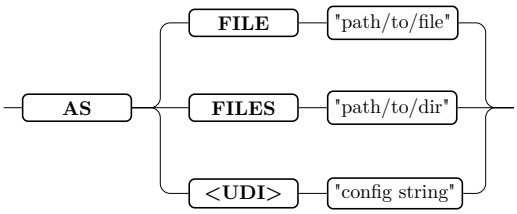


Figure 4.5: Syntax of store destination

each value of the source, a new group is created, and the aggregation function is evaluated on all documents in that group. Using the optional `AS` keyword, the group value can be written to a new attribute.

The first query in Listing 4.6 calculates the minimum length of all tweets in the collection. The second query counts the number of tweets grouped by their language The result will be an array of objects, where each object has a `group` attribute containing the language and a `count` attribute containing the number of tweets in that language.

In streaming queries, the `AGG` statement can also be used to aggregate over a window of documents. After the keyword, the window size in number of documents has to be given using the `WINDOW(<size>)` function. Query three in Listing 4.6 counts the number of tweets grouped by their language in a window of 100 documents. Currently only tumbling windows are supported, i.e., the aggregation resets after each window.

4.1.9 Storing Collections

Using the final optional `STORE` statement, the result of the query is either stored in a collection or exported from the system. If `STORE` is only followed by an identifier, the result is stored in a collection with that name. If no collection with this name exists, the system creates a new one and stores the result in this collection. Else, the results are appended to the existing collection.

The `STORE` statement can also be followed by a `AS` keyword followed by the export type. Figure 4.5 shows the different export types. Currently, the system natively supports exporting the JSON documents into either one or multiple line-separated JSON files. Additionally, results can also be streamed to standard output in a line-separated format if the system runs in a streaming-compatible mode. The export statement also supports user-defined export types, which can be used to export documents into other formats.

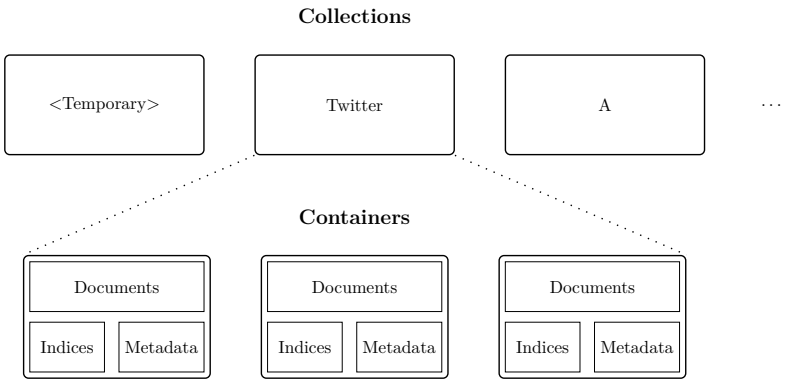


Figure 4.6: Overview of the storage hierarchy of JODA

4.2 Storage

In JODA, documents are logically divided into *collections*. Queries reference collections to store and retrieve a set of documents. They are comparable to the collections in MongoDB and tables in SQL. Within each collection, documents are partitioned into self-sustained *containers*. Containers bundle all necessary data and meta-data required to evaluate any part of a query. Figure 4.6 summarizes the basic JODA storage architecture.

4.2.1 Collections

A collection is a semantic user-defined group of documents. When the user imports data into JODA or a query returns a result set, the data is stored in a collection. If no name is specified by the user, the collection is called a *temporary collection*. These collections can't be referenced by the user in a future query and are removed automatically when they are not needed anymore. If a name is specified, the collection is called a *named collection*. Named collections can be referenced in the `LOAD`, `STORE`, and `JOIN` commands. They have to be explicitly deleted by the user to free all allocated memory. Collections are mutable, as new documents can be added at any point by referencing an existing named collection in the `LOAD` and `STORE` commands. Internally, collections do not store the documents directly. Instead, they store a list of containers, which will be explained in Section 4.2.2. Each collection has a unique directory assigned in a temporary directory on the file system. If in-memory data has to be evicted to disk, the data is stored in this directory.

4.2.2 Containers

Containers partition the actual documents into self-sustained units. A container is initialized with a maximum size. Documents are added into containers until the maximum size is reached. Then, a new container is created. After it is filled, the container is finalized and indexed. The list of documents within a container is immutable. No document can be removed, added, or modified.

But containers also store indices and meta-data that may change over time. For example, the container can store a bloom filter of all paths in all documents. This allows the container to quickly determine if a query can return a non-empty result. They also store a filter-predicate cache, which is used to speed up the evaluation of queries. After every filter operation, a Boolean vector

representing the chosen documents is stored together with the filter predicate. If the same filter predicate is used again, the Boolean vector can be returned directly without accessing the documents. This cache can also be used to restrict the documents that have to be evaluated if a stored predicate is a subset of the current predicate.

Documents of containers can also be serialized to disk if needed. This allows the in-memory representation of the documents to be removed. Indices and meta-data remain in memory and can be used to answer queries. If the indices are not sufficient to answer the queries, the documents are reparsed from the disk. Additionally, containers can be initialized lazily. This means that the documents are not parsed, but only a reference to the file on disk, and the position within the file is stored. This allows the documents to be parsed only when they are needed.

4.2.3 Memory Management

Most operations of JODA are performed in memory. This allows for fast and efficient processing. However, the amount of memory required to store the documents and indices can be very large. To avoid out-of-memory errors, JODA uses a memory manager to manage memory usage. The memory manager is responsible for monitoring and freeing memory.

For each collection, the timestamp of creation and the last access is stored. If the system does not have enough free memory to import new data or perform a query, a cleanup procedure is started. In this procedure, the documents of a set of collections are removed from memory and serialized to disk. JODA implements multiple strategies to determine which collections to remove: LRU, FIFO, size, dependencies, and random explorer. The default strategy is to evict the least-recently-used collection first. But users can also configure JODA to use different strategies. The first-in-first-out strategy prioritizes the oldest collections, while the size strategy prioritizes the largest collections. The dependencies strategy prioritizes collections that have the least depending collections (see Chapter 5). Finally, the random explorer strategy selects collections that are least likely to be used if the user follows the random explorer model (see Chapter 8). Once the cleanup procedure is finished, the import or query can be performed.

For every collection, JODA creates a temporary directory on disk. This directory is used to store one JSON file for each evicted container. Every document within the container is translated into the string representation of a JSON object and stored in a single file. In memory, the container is updated with a reference to the file on disk and the byte positions of every document.

Every container stores a reference counter to prevent the eviction of documents currently in use. If the reference counter is larger than zero, the container documents are not evicted. On the other hand, if the reference counter is incremented from zero to one and the container is evicted, the documents are reparsed from disk. To prevent unnecessary reparsing, functions can also specify which documents they need, such that only these documents are reparsed. If a later function needs different or more documents, the evicted documents are reparsed again.

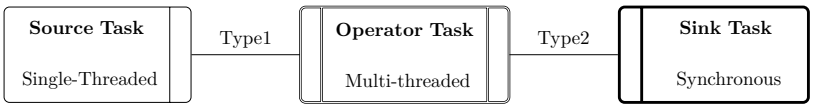


Figure 4.7: Example tasks connected by two queues

4.3 Query Execution

At the heart of JODA query processing is the *pipeline*, which is a sequence of connected *tasks* that are executed in a specific order. A query is translated into multiple tasks that are added to the execution pipeline. Within the pipeline, the tasks are connected by I/O queues. Queues transfer different types of data, depending on the connected tasks. The tasks are then executed by multiple threads, depending on the *scheduler* strategy.

4.3.1 Tasks

A *task* is a self-contained unit of work that is executed by a single thread. It is responsible for processing a specific part of a query. There are three types of tasks in JODA: *source tasks*, *operator tasks*, and *sink tasks*. Source tasks generate data from outside the system, collections, or other centralized structures. They only have an output queue through which they send data to the rest of the pipeline. Operator tasks process data from their input queue and write the result to their output queue. Not every input tuple has to produce an output tuple, it is also possible to output more or fewer tuples than the number of input tuples. Sink tasks only consume data from their input queue and may write results to centralized structures, storages, or even outside systems.

Every task can have one of three types of parallelism: *synchronous*, *single-threaded*, and *multi-threaded*. Synchronous tasks are executed by the main execution thread and serve as a synchronization point for other tasks. A synchronous task can only be started if all previous synchronous tasks are finished. Single-threaded tasks are executed by a single thread, with no insurance on the order of execution. I/O heavy operations that are not CPU-bound or non-parallelizable tasks are executed as single-threaded tasks. But multi-threaded tasks can be executed by multiple threads in parallel. They are used for CPU-bound tasks that benefit from being parallelized. For every thread executing a task, a new *task instance* is created. For single-threaded and synchronous tasks, only one instance exists at any time.

Figure 4.7 showcases the task visualization for the remaining thesis. Single-threaded tasks are drawn with a normal border, synchronous tasks use a thick border, and multi-threaded tasks a double border. Source tasks have a single output on the right, sink tasks a single input on the left, and operator tasks show both connectors.

Depending on the task type, the *scheduler* decides when and how many task instances are started. Every task instance has a status that indicates its current state, which may be any of the following:

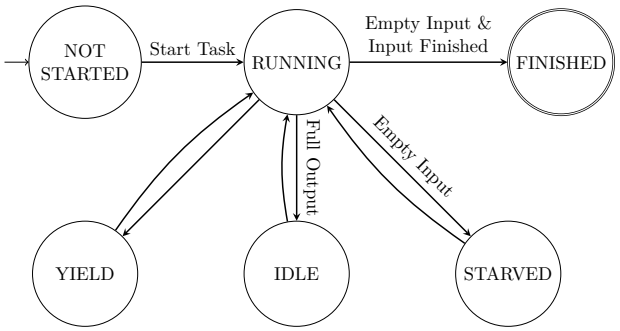


Figure 4.8: Transitions between task instance states

- `NOT_STARTED`: The task instance has not been started yet.
- `RUNNING`: The task instance is currently running.
- `FINISHED`: The task is finished and no more tuples will be read or written.
- `STARVED`: The task is not finished, but no more tuples can be read from the input queue.
- `IDLE`: The task is not finished, but no more tuples can be written to the output queue.
- `YIELD`: The task is not finished, but yields the execution to another task.

Figure 4.8 shows the possible transitions between the different task-instance states. First, a task instance with the default status `NOT_STARTED` is created. Then, it is started and transitions to `RUNNING`. Every task instance has an `execute` method that depending on the type, reads from the input queue, processes the data, and writes to the output queue. How many tuples are read and written is determined by the task. The execute method then returns the new status of the task instance. Depending on this status, the scheduler decides what to do next. If the task instance is finished, it is not considered by the scheduler anymore. An instance is finished if the input queue is empty and marked as finished. The queues connecting the tasks keep track of how many producers are filling the queue. If all producers are finished, the queue is marked as *finished*. If the input queue is empty, but not yet finished the task instance transitions to `STARVED`. On the other hand, if the output queue is full, the task instance transitions to `IDLE`. Additionally, may the task instance yield the execution to another task by returning `YIELD`. This happens after a certain number of tuples have been processed to ensure all tasks down the pipeline are executed. Every waiting task instance may be restarted by the scheduler at a later point in time and transition back to `RUNNING`. A task in the pipeline is considered finished if all of its task instances are finished. A list of all implemented tasks can be found in Appendix B.

```
LOAD Twitter
CHOOSE EXISTS('/user')
AS ('':'/name')
STORE Users;
```

Listing 4.7: Example JODA query loading, filtering, transforming, and storing twitter documents.

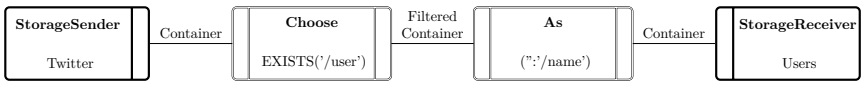


Figure 4.9: Pipeline created by query in Listing 4.7

4.3.2 Pipeline

An ordered sequence of tasks, optionally connected by queues, is called a *pipeline*. Queries are translated into tasks that are added to a given pipeline. For each added task, the pipeline creates a queue for the task's output if the task is not a sink task. If the task is not a source task, queues of compatible previous tasks are connected to the task's input. JODA keeps a list of all compatible tasks, which are tasks with the same output and input types that are allowed to happen after each other in a query.

For example, consider the query in Listing 4.7. Figure 4.9 shows the resulting pipeline. First, the `LOAD` statement is translated into a source task that loads the Twitter collection from JODA storage and writes the containers to the output queue. After the `StorageSender` task is added, the pipeline creates a queue of type `Container` and connects it to the task's output. Then, the `CHOOSE` statement is translated into an operator task that reads the containers from the input queue, filters the containers, and writes the filtered containers to the output queue. As this task has an empty input, the pipeline searches backward for a compatible unconnected task and connects the previously created queue to the task's input. As seen previously, a new queue is created for the task's output. Next, the `AS` statement is translated and added similarly to the `CHOOSE` statement. Finally, the `STORE` statement is translated into a sink task that reads the containers from the input queue and stores them in the JODA storage. The resulting pipeline begins and ends with synchronous source and sink tasks, and has two multi-threaded operator tasks in between.

After the query is translated into a pipeline, it is optimized by the systems as described in Section 4.4. The scheduler then starts the pipeline and schedules the tasks depending on their parallelism type.

4.3.3 Scheduler

The scheduler is responsible for creating and scheduling task instances in a pipeline. Given a maximum number of threads and a list of tasks, the scheduler first creates the task instances. For every task, the scheduler creates one instance for each single-threaded task and as many instances as the configured number of threads for multi-threaded tasks. The instances are stored in an internal stage list. Synchronous tasks are executed in the main query-execution loop and are not added to the stage list.

The scheduler then starts the execution of the pipeline. For every stage, a task instance is started in round-robin order until all threads are occupied. When a task instance returns (i.e., finishes or yields), the scheduler reschedules a new task instance depending on the returned status. After the initial scheduling, the scheduler also executes the synchronous tasks in order.

When a task instance returns, by default a task instance in the next stage is started. But if the task instance is `STARVED`, the scheduler searches for a task instance in the previous stage. No matter the stage, the scheduler tries to start a task instance that is not `FINISHED`. If no such instance exists in the chosen stage, the scheduler continues the search in the next one. If no task can be found in any stage, the rescheduling returns. As soon as all task instances in all stages and the synchronous tasks are `FINISHED`, the scheduler stops the execution of the pipeline.

4.4 Optimization

JODA implements a rule-based optimizer that uses a set of rules to transform a pipeline into an optimized version. Each rule is a class that implements a `optimize` method that takes a pipeline and returns a range of tasks to be replaced and the replacement tasks. If the rule can not be applied, the method returns an empty range. The optimizer applies all configured rules in a loop until no rule can be applied anymore.

The default rule-class consists of a set of task ids to be replaced and a set of replacement tasks. If the set of tasks to be replaced is contained in the pipeline and connected with each other, the `optimize` method replaces the tasks with the replacement tasks. But the rule can be extended to implement more complex optimizations. Additional checks can be performed on the pipeline and tasks to ensure the rule can be applied. For example, is it possible to apply a rule depending on the filter predicate in the choose task.

It is generally desirable to have as few tasks as possible in a pipeline, as the I/O overhead of queue communication and task scheduling reduces the performance. Hence, optimization rules should concentrate on either replacing high-impact tasks with versions optimized to a specific corner case or replacing common multi-task patterns with single tasks.

4.4.1 Parsing Optimization

JODA tasks are created with modularity and interoperability in mind. Specifically, the parsing of a line-separated JSON file is split into multiple tasks. As seen in Figure 4.10, first the file is opened and a character stream is passed on to a reader task which scans the file for a `newline` character. Every line is then extracted from the stream and passed on as a string to a JSON parser task. This ensures that streams from different sources, like HTTP requests, decompression libraries, or similar can be parsed by the same tasks. But if only local files are parsed, the reading can be improved by performing it in a more efficient single task. The `FileMap` optimization rule replaces the file opener and file reader tasks with a single file mapper task, which uses the `mmap` system call to map the file into memory and efficiently scan for the newline delimiter.

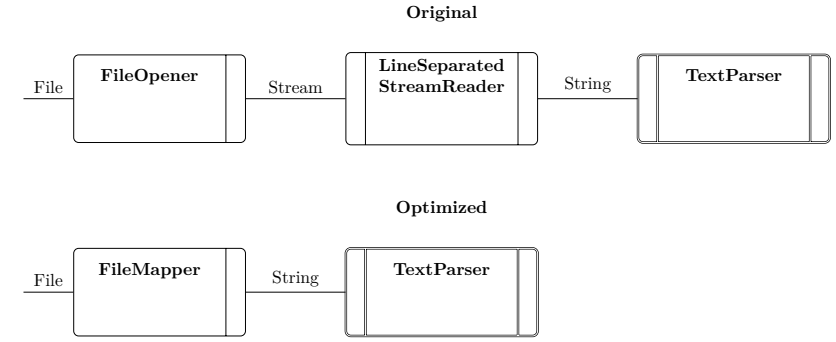


Figure 4.10: Optimization rule `FileMap`

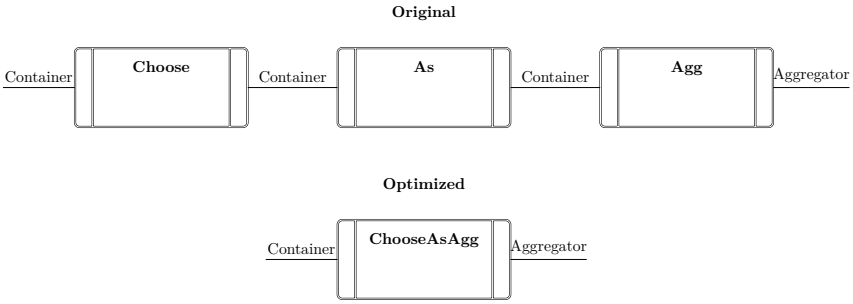


Figure 4.11: Optimization rule `ChooseAsAgg`

4.4.2 Main-Query Evaluation Merging

The most common operations in every JODA pipeline are the `CHOOSE`, `AS`, and `AGG` tasks. They all are multithreaded operator tasks and take a container as input and produce a container as output. The evaluation of every task is very similar, in that they have a set of predicates, tuples, or aggregators and call a method on the container to evaluate it. Hence, it makes sense that they are executed in the same task in a tight loop over the container. But as not every query contains all three statements, they have to be split into multiple tasks. To reduce the overhead of queue communication, JODA includes optimization rules to merge all combinations of these tasks into a single task. As a result, the main query evaluation of a single container is always performed in a single task, where the resulting container of the previous step is directly passed to the next one in the same thread. Figure 4.11 shows the example case where all three operations are present in the pipeline and can be merged into a single task.

4.4.3 Multi-Query Optimization

Generally, JODA translates a single query into one pipeline. For interactive queries that always return a single result, this is sufficient. But JODA also accepts a query file as input that contains multiple queries, but only returns the last result. Similarly, can multiple queries also be supplied through the HTTP API. These features can be used to perform multi-stage queries where the result of one query is used in the following. Listing 4.8 shows an example of two related queries.

```
LOAD Twitter FROM FILE "twitter.json";
LOAD Twitter
CHOOSE EXISTS('/user');
```

Listing 4.8: Three partly unrelated example queries

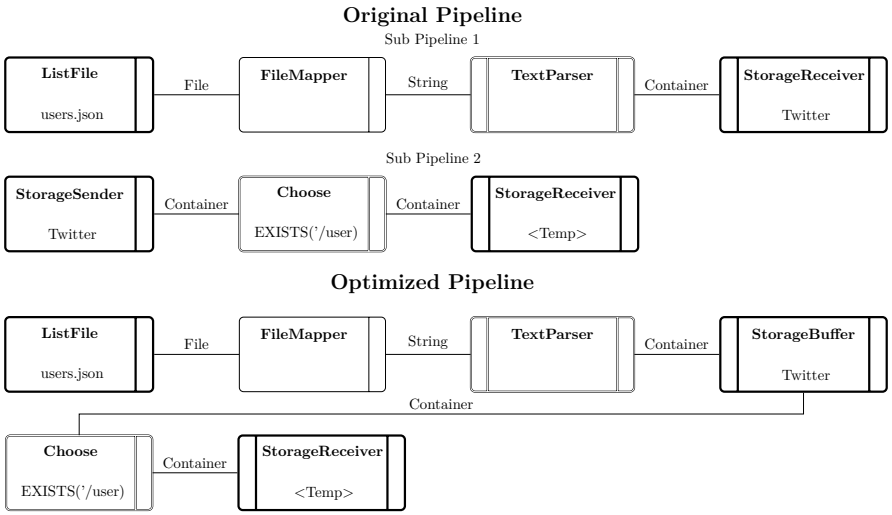


Figure 4.12: Multi-query optimization example pipelines

The first query parses the Twitter dataset and stores it in the `Twitter` collection. This dataset is then used in the second query to filter all tweets that contain a top-level `user` attribute. JODA could now create one pipeline for each query and execute them in sequence. But this would result in some overhead as available computing resources could not start processing the second query until the first one is finished. To avoid this, JODA will merge all queries into a single pipeline. Before the query is executed, the scheduler analyzes any given pipeline for connections and splits unconnected parts into *sub-pipelines*. Then, each sub-pipeline is executed as if it were its own pipeline. In itself, this feature does not improve performance, but it allows the optimizer to optimize the pipeline as a whole over all queries.

Figure 4.12 shows the example pipeline of the two queries. It contains two unconnected sub-pipelines that end and start with the `Twitter` collection respectively. The QueryCombination rule searches for a successive unconnected `StorageReceiver` and `StorageSender` task and merges them into a single `StorageBuffer` task. These buffers send all already existing containers to the queue, and store all new containers in the collection while also passing them on. In an additional pass of the optimizer, the `StorageBuffer` task can even be removed, if this query is the last query to be executed before the system is shut down. This optimization is only possible if the query set has been supplied through the CLI in a non-interactive session.

4.5 Applications

Data processors are used for many different use cases. To support a wide range of applications, JODA includes multiple execution modes and supporting tools that can be used to solve different problems. This section gives a brief overview of the ways a user can interact with JODA.

4.5.1 CLI

The first and most basic way to interact with JODA is via the command line interface (CLI). By simply invoking the JODA executable without any arguments, the user is presented with an interactive shell that allows to execute JODA queries. The shell also provides a set of commands to manage collections, set some basic settings, and get information about the current state of the system. The result of each query can be interactively navigated by browsing through each individual result document.

If the executable is started without a valid interactive shell (e.g. when started from a script) or the user supplies the `--noninteractive` flag, the system allows the execution of streaming queries. One or multiple queries can be supplied via the command line and the results are printed to the standard output as a stream of line delimited JSON documents. Input documents can be imported via the standard import functionalities, or by piping them into the executable. This allows JODA to be combined with other tools. For example, could the user get a JSON file from the web, process it in JODA, and compress the result using `curl <url> | JODA -q "<query>" | gzip`.

JODA is extensively configurable. Every optimization and feature can be enabled, disabled, or tweaked to fit the user's needs. These settings can either be set via command-line arguments or by using a configuration file.

4.5.2 Client/Server

Instead of using the CLI to directly interact with JODA, the user can also start a server instance that can be accessed via a client. The server can be started by supplying the `--server` command-line argument. The JODA instance will then listen on a specified port for incoming connections. It uses an HTTP API to communicate with the client. Via this API the current collections can be retrieved, queries can be executed, and results can be fetched using paginated requests.

A simple client program is provided with JODA. It has similar functionality as the CLI, but instead of executing queries directly, it sends them to the server, fetches the results, and displays them. But as the server uses a simple HTTP API, it is also possible to write custom clients that can be used to interact with JODA.

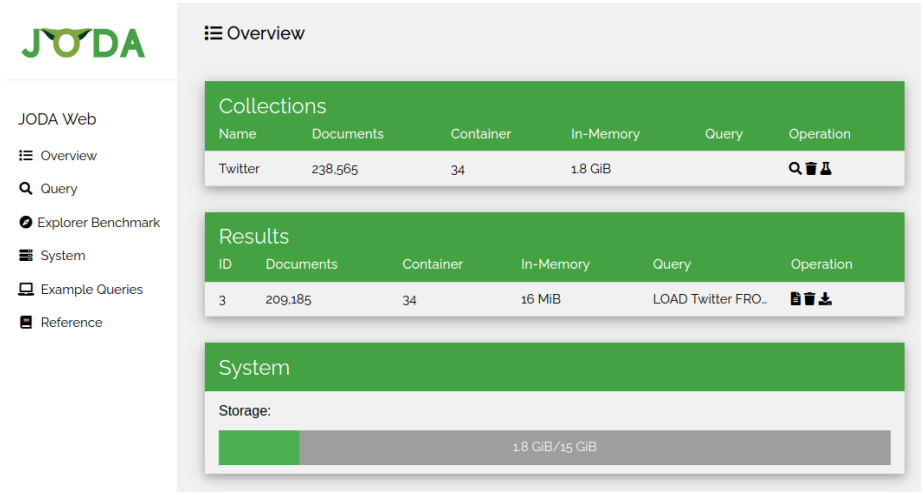


Figure 4.13: Screenshot of the web interface start page.

4.5.3 Web

To improve the user experience, we also implemented a web interface for JODA using GoLang. This is an additional server that communicates with a JODA server instance and provides an easy-to-use web interface. Figure 4.13 shows the start page of the web interface. Here an overview of all collections and stored result sets is shown. For each collection, the user can execute a query, remove it, or get a deep analysis of the structure and content of the collection. Results of previous queries are cached in the system and can be viewed with an interactive JSON viewer with syntax highlighting. Single documents and whole result sets can also be downloaded as JSON files.

For each query, statistics about the execution time and the number of documents that were processed are stored. These statistics are shown after each query as can be seen in Figure 4.14. The query pipeline is also visualized as a graph. The web interface also contains an exhaustive reference of the JODA query language and example queries to help the user get started.

Explorer

To further help the user explore unknown data, we implemented an exploration tool that analyzes the structure and content of a collection and recommends queries that can be used to create interesting datasets. For example, if a dataset is split into two or more structurally different parts, the tool can recommend queries that extract each of these partitions. Or if a certain attribute contains a statistically significant value, the tool can recommend queries that extract all documents that contain or do not contain this value. Each recommended query can be executed and the tool will continue to analyze the result. New recommendations are generated based on the new data and the process is repeated until the user is satisfied or no more recommendations can be made. This allows the user to explore a dataset and create interesting subsets of it without having to write any queries.

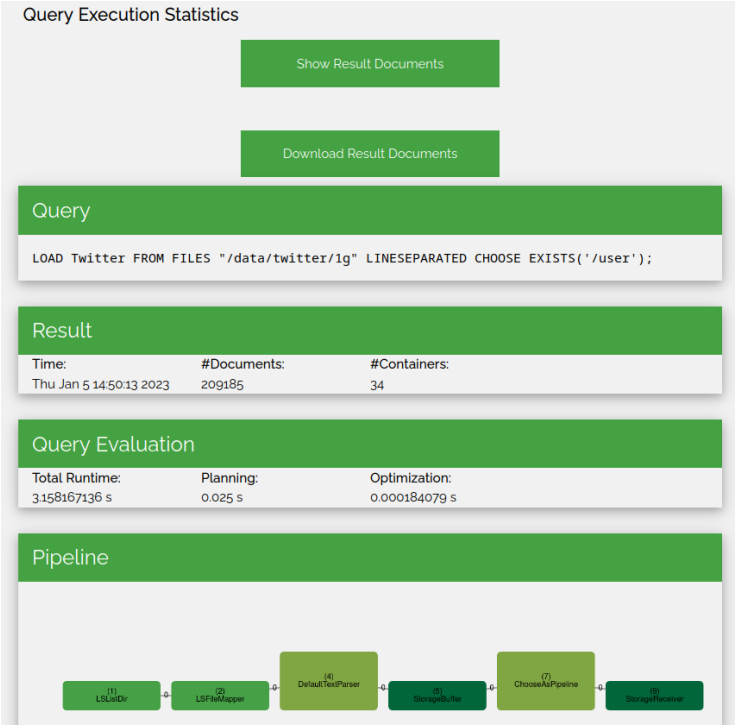


Figure 4.14: Screenshot of the query statistics page.

BETZE

In Chapter 8, the BETZE benchmark generator for exploratory queries is presented. The generator itself is a GoLang library with an included CLI. The JODA web interface includes this library and provides a visual interface to generate and execute exploratory queries. This allows the user to set benchmark parameters, generate queries, and view and export them in different query languages. The benchmark session is also visualized as a graph with statistics about each individual query and the generation process.

Chapter 5

Delta Trees — Optimizing for Iterative Queries

As mentioned in the previous chapter, the results of each task within the query pipeline are passed on to the next task. For transformation tasks, this means that a new container has to be created with the transformed documents of the incoming container. Take for example the query in Listing 5.1. It loads an existing Twitter dataset, adds a new attribute to each document, and removes the `user` attribute. The original container in the Twitter collection can't be modified, as it may be used by other queries in the future. Thus, a new container has to be created, which contains the transformed documents. If the changes to the original document are minimal, this is a waste of resources, as the new container will mostly contain the same data as the original container.

This kind of query workload is especially common in the field of data science, where data is often iteratively filtered and transformed until a certain result is achieved. For large datasets, duplicating most of the data multiple times is not feasible. Even if intermediate results are removed after the query is finished, the memory and storage requirements are still high.

In this chapter, we present delta trees [3, 4], which succinctly represent data modifications while keeping original datasets intact. Instead of duplicating and modifying the original data, we only store the changes in the document and reconstruct the full document on demand. They also support partial materializations as a tradeoff between storage capacity and performance.

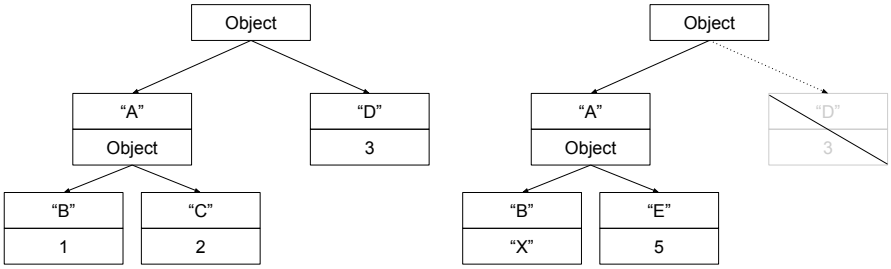
The remainder of this chapter is structured as follows. Section 5.1 gives an overview of the approach. Section 5.2 introduces the theoretical model underlying delta trees, whose implementation in JODA is shown in Section 5.3. This implementation is evaluated in Section 5.4. Lastly, we give a short summary in Section 5.5.

5.1 Overview

In many systems, semi-structured documents are represented as trees (or nested hash maps, which can be seen as trees) in the memory. For example, HTML and XML documents are mostly stored according to the document object model (DOM) [71]. This model provides an interface to dynamically query and edit the represented documents. Each document is represented as a tree, where each node is an object representing a part of the document.

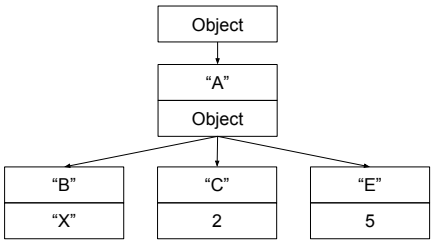
```
LOAD Twitter
AS *, ('/parsed_at': NOW()), ('/user':);
```

Listing 5.1: Query transforming documents in the Twitter dataset



(a) Base document

(b) Delta tree - document



(c) Combined delta tree

Figure 5.1: Example of base document with a delta tree

JSON documents, the currently dominant semi-structured document type used to exchange data, can also be represented as a DOM tree. For example, the RapidJSON [8] library uses the DOM representation when parsing JSON documents into main memory. We use RapidJSON in our JODA software for parsing and storing JSON documents in memory. The proposed improvements to the system consist of additions to the storage hierarchy, algorithms to interact with delta trees and documents in a transparent manner, and adaptions of the target system to use these algorithms.

Normally, to change a document, while keeping the original intact, the system has to deep-copy the whole document tree and modify the copy. But traversing a tree and allocating memory for its nodes is a performance-heavy task. The idea of our approach is to create an additional delta tree on top of the original, which only contains the minimal information required to represent the changes made to the original tree.

Figure 5.1 visualizes this idea. In Figure 5.1a the internal tree structure of a normal JSON document is given. A possible delta tree, based on the previous tree is shown in Figure 5.1b. It changes the values of the B member, adds a E member, and removes the D member. Combining these trees results in Figure 5.1c.

5.2 Model

Most semi-structured documents can be represented as a directed tree. A directed tree is an acyclic-connected graph $T' = (V, E)$, with V being the vertices in the graph and E the edges. This representation is also often used as the

in-memory storage model of the documents to allow easy traversal and modification. More specifically, the Document Object Model (DOM) [71] is often used to represent XML and HTML documents. Every document has one root node, which can contain children nodes that are themselves root nodes of subtrees. The leaves of the tree represent the stored data. Additionally, every node is labeled with metadata, like the attribute name and/or type.

We augment the tree definition to obtain a document $T = (V, E, I, A, L, D)$. V and E are still denoting vertices and edges, respectively. The vertices are distributed into the distinct subsets $I \cup A = V$, with I being the inner structural nodes and A the atomic leaf nodes. An edge $(x, y) \in E$ represents a parent/child relationship with y being the child node of x. L is a set of labels, a mapping of $I \to \Sigma^*$, where Σ is a domain specific alphabet. For each parent node, the labels of all its children are unique. D represents the actual stored data and is a mapping $A \to \Sigma^*$ from the leaf nodes to the data. This is a very generic abstraction, to which JSON, XML, and YAML documents comply.

5.2.1 Path

A path p is an ordered list of labels $p = (l_1, \ldots, l_n)$ with $l_i \in \Sigma^*$. As the child labels are unique in the scope of the parent node, it is possible to uniquely identify each node in the tree by a path. For each inner node, $i \in I$ there exists a mapping $vtp(v) = v \to p$. There is also a reverse mapping, path-to-vertice. Given a tree T and a path p:

$$ptv(T, p) = \begin{cases} v & \exists v \in V, vtp(v) = p \\ \varnothing & \text{else} \end{cases}$$

Every input data that can be represented as the aforementioned tree, can also be used with this concept of paths. For example, the W3C recommendation for XPath [72] describes a query language for XML documents in this style. A document can be queried by specifying an ordered list of labels, separated with the '/' symbol. Similarly, RFC 6901 [7] proposes a JSON pointer with a similar format to query JSON documents.

5.2.2 Delta Tree

Using the previous definitions, we can now formally introduce *delta trees*. A delta tree $D = (b, t, P_D)$ is a tuple consisting of a base tree $b = (V_b, E_b, I_b, A_b, L_b, D_b)$, a tree $t = (V_D, E_D, I_D, A_D, L_D, D_D)$ containing the changes relative to b, and a set of paths P. The base tree b and the change tree t are both valid documents.

The set of paths P describes all the paths that have been changed in this delta tree. For example, would the paths `{(<Root>, "A","B"), (<Root>, "A","E"), (<Root>, "D")}` denote all changes of the delta tree in Figure 5.1b. Given this set and the base tree b, we can define the set of overwritten vertices with $OV(b_D, P_D)$. For all $v \in V_b, v \in OV(b_D, P_D)$ iff:

1. $\exists p \in P_D, vtp(v) = p$
2. $\exists v' \in OV(b_D, P_D), (v', v) \in E_b$

This means, a vertex of b is overwritten in D if it is either directly changed in the delta tree, or if it is a child of a changed vertex.

The document resulting from combining the base tree b with the delta tree t is again a tree $R = (V_R, E_R, I_R, A_R, L_R, D_R)$ with:

$$V_R = \{v | v \in V_b \wedge v \notin OV(b_D, P_D)\} \cup V_D$$

$$E_R = \{(x, y) | (x, y) \in E_b \wedge x, y \notin OV(b_D, P_D)\} \cup E_D$$

I_R, A_R, L_R, and D_R are defined analogously to V_R.

5.2.3 Delta Hierarchy

As mentioned previously, a delta tree describes changes to a base tree. Thus, it is possible to have a base tree b and a delta tree $D = (b, t_D, P_D)$ derived from it. Let the result of merging these trees be R_D. We can now use this result tree as a base tree for an additional delta tree $D' = (R_D, t_{D'}, P_{D'})$. Analogously, we obtain the result tree $R_{D'}$.

This means, we can build delta trees on the results of previous trees. A set of delta trees $H = (D_0, D_1, \ldots, D_n)$, where each delta tree references the result of the previous one is called a *delta hierarchy*. D_0 represents the base tree b, which can be seen as a delta tree $D_0 = (\varnothing, b, \{< \text{Root} >\})$, with no base document and where the root node is overwritten, thereby using the whole change tree. The result of the delta hierarchy equals the result tree of the uppermost delta tree $R_H = R_{D_n}$.

5.2.4 Costmodel

The primary enhancement of delta trees is the reduced memory footprint. We now define the cost of a delta tree as its memory requirement.

Let the cost of a JSON tree node be C, which is the cost of the data contained in the node, as well as all children nodes. For atomic nodes, this cost is exactly the size of this datatype, or in the case of strings, the length of the string plus the memory size of a character.

For the remaining nodes, C is defined as follows:

$$C(n) = \begin{cases} \text{size}(val(n)) & n \in A \quad (atomic) \\ \sum_{(n,n') \in E} C(n') & n \in I \quad (array/object) \end{cases}$$

The total cost of a document T is thereby $C(T) = C(root(T))$ where $root(T)$ is the root node of the document. Similarly, the total cost of a delta tree D is the cost of its derived document plus the cost of the overwritten paths $C(D) = C(t) + C(P_D)$ with

$$C(P_D) = \sum_{p \in P_D} \sum_{x \in p} size(x)$$

Multiple delta tree documents may share a single P_D instance, if they were created by the same query. In this case, the cost of a single delta tree document is $C(V) = C(t) + \frac{C(P_D)}{N}$ for N delta tree documents.

As we can see, for large N the path cost becomes irrelevant.

$$\lim_{N \to \infty} C(D) + \frac{C(P_D)}{N} = C(D)$$

Also, the cost $C(P_D)$ is always a constant per set of delta trees with $C(P_D) << \sum_{n \in N} C(D_n)$.

The cost of a delta hierarchy is simply the sum of the base tree and all delta trees in the hierarchy.

5.3 Realization & Optimizations

We now describe how delta trees are implemented inside JODA. JODA uses the RapidJSON parser, which creates an in-memory DOM tree representation. But the introduced algorithms should be straightforward to implement in similar systems, as long as the requirements stated in Section 5.2 are fulfilled. Given a delta hierarchy $H = (D_0, \ldots, D_i, \ldots, D_n)$, as defined above. The document trees are represented internally as DOM trees, which can directly be mapped to our theoretical model of document trees presented before.

H supports the following actions:

- Traverse a result document R_i using the visitor pattern.
- Materialize changes of a sub-tree at path expression p.
- Get the subtree of a path expression p.

5.3.1 Traversal with Visitor Pattern

A well-known method of traversing structured data without changing it is the visitor pattern [73]. With it, we can traverse our internal tree structure to perform many different tasks. For example, it is used for stringification and duplication of whole documents or parts of them within our system. An efficient implementation of this pattern is crucial for the efficiency of our delta tree implementation.

As mentioned in Section 5.2.3, the result of a delta hierarchy can be computed by iteratively creating an intermediate result for every level of the delta hierarchy. Using this approach on a delta tree D_1 which is based on the document D_0, we would combine both trees, to build the result document R_1. However, this approach does not work well with delta hierarchies of many levels, as too many intermediate results have to be computed. Instead, Algorithm 1 traverses a delta hierarchy as if it was one document (illustrated in Fig. 5.2), for an arbitrary number of delta DOM trees. We start at the root document of the upper most delta tree. First, we check using a function `IsShared`, if the children of the given node are distributed over multiple trees. By storing all overwritten paths in a suitable structure, like a map, we can check this by performing one hash lookup per delta tree.

Data: p = ' '; $D_0, \ldots, D_m$
1 **if** *IsShared(p,*$D_0, \ldots, D_m$*)* **then**
2 O = GetLastOverwrite(p,$D_0, \ldots, D_m$));
3 **for** i = *O to* D_m **do**
4 members += GetMembers(D_i);
5 **end**
6 **for** *member in members* **do**
7 Visit member;
8 Recurse(p+'/'+member.id,$D_0, \ldots, D_m$);
9 **end**
10 **else**
11 n_D = GetBaseNode(n);
12 Visit n_D;
13 **end**

Algorithm 1: Simultaneous traversal algorithm

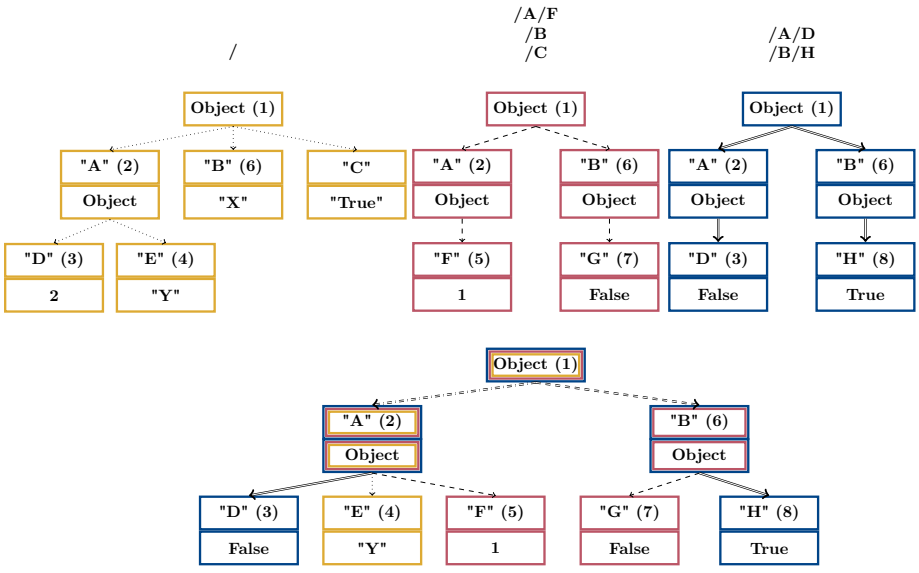


Figure 5.2: Simultaneous traversal

If this node is shared—like the root, "A", and "B" nodes in the example—we get the upper most overwrite for the given node, with the `GetLastOverwrite` function, which returns the document highest up in the delta hierarchy which completely replaced this attribute. In case of node "A", the base document was the last one to overwrite the value, as the base document overwrites everything. For node "B", the second layer overwrote this sub-tree last. For this, we also only have to do a prefix check on the overwritten paths. In our implementation, we include this check in the `IsShared` function call, to reduce the runtime. We then collect all unique paths of children of this node and nodes with the same path in the trees above. Each member is then visited once, and the function is called with each of their paths again.

5.3.2 Retrieval of Atomic Values

Most query functions only have to read atomic values. As atomic values are never shared and can be read from a single delta tree directly, we introduce a `getAtomic` function which optimizes these accesses. By using the previously explained `GetLastOverwrite` function, we get the delta tree which overwrote the given path last. The atomic value we try to find is either in this tree, or does not exist at all. Hence, we can simply extract the value from this one delta tree.

5.3.3 Partial Materialization

Simultaneous traversal of delta hierarchies is still more expensive than directly accessing a normal (materialized) tree. To mitigate this issue, we introduce a method that allows *partial materialization* of a given delta tree. Given a delta hierarchy $H = (D_0, \ldots, D_n)$ and a path p, we can materialize p into D_n.

Algorithm 2 shows the materialization procedure. Lines 1–6 traverse the delta hierarchy to the required path p and prune delta trees, which are not required for further computation, from the search space. The resulting delta hierarchy is then used to take a special `CopyVisitor` object, which creates a deep-copy of the sub tree, in Line 7—we use Algorithm 1 to traverse this sub tree in the delta hierarchy. This copy is then assigned to path p within the topmost delta tree and the materialized path is added to the set of overwritten paths.

```
   Data: p; D_0, ..., D_n;
1  for D_i ∈ D_0 to D_{n-1} do
2  |  Node = D_i.find(p);
3  |  if Node is null then
4  |  |  remove D_i from list;
5  |  end
6  end
7  D_n.set(p,Accept(CopyVisitor,p, D_0, ..., D_n))
8  D_n.P+ = p;
```

Algorithm 2: Materialize_P()

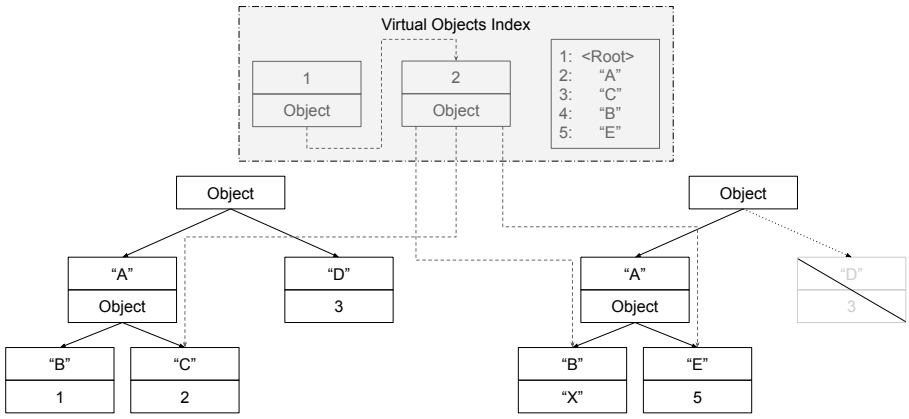


Figure 5.3: Object index

If the path to be materialized points to the root, the entire delta hierarchy is materialized. In this case, the uppermost delta tree is converted into a normal tree, and removed from the delta hierarchy. This is called a *complete materialization*.

Materializing (parts of) the delta hierarchy increases memory consumption, as we store duplicated data. Partial materialization could be used by the system, if specific paths in the delta hierarchy are accessed frequently by queries and the increased memory footprint is tolerably small. Hence, we can choose, depending on the available memory, if we want to materialize a path to decrease query runtime.

5.3.4 Object Indexing

To prevent the partial materialization of objects, a *virtual object index* is created. This index combines the key benefits of delta trees with the advantage in read performance of materialization. A virtual object is a list of tuples containing an attribute id and a pointer to a value or nested virtual object, as shown in Figure 5.3. The attribute id is a numerical value, retrieved by mapping a string attribute name to a numerical value using a hash map. This has two advantages. (1) Having a numerical value reduces the cost of comparisons needed for the linear search of children. (2) The string dictionary is stored in the container and shared by many documents, thus, reducing the required memory of this index.

We create these virtual objects as soon as an object, that is distributed over multiple trees in the delta hierarchy, is traversed for the first time. During traversal, we map the attribute names of the children to the attribute id and add it to the virtual object, together with the pointer of the actually traversed value. Each value may reside in a different tree within the delta hierarchy. The traversed object is then replaced by the virtual object in the highest delta tree of the hierarchy. Future accesses of the object can then use the created index without traversing multiple trees.

5.3.5 Adaptive Algorithm

The main advantage of delta trees is the reduced memory requirement. Evidently, each delta tree is smaller or equal in size as the result tree that is given by combining the whole delta hierarchy—as all of its nodes are contained in the result, plus potential additional nodes from the base document. Thus, the memory cost of delta trees should always be smaller or equal to materializing the whole result. However, this may not always be the case in practice, as the RapidJSON library, which is used to create JSON documents, creates each new object and array with 16 placeholder children. In many cases, this is a sensible decision, as reallocating memory for more children is an expensive operation and objects and arrays often have more than one child. For delta trees that mostly consist of a few nodes, this decision can often be a disadvantage. We extended our previously introduced cost model by these implementation-dependent factors. Additionally, we added a sample step to our system before deciding which execution method, delta trees or complete materialization, to choose. The transformation is performed for $\leq 1\%$ of documents with both execution methods. Then the memory requirement of these documents is calculated and the method with the lowest requirement is chosen. This decision is performed once for each container.

5.4 Experimental Evaluation

5.4.1 Settings and Data and Workloads

The experiments are executed on a machine with 4 Xeon E7-4830 CPUs, each having 12 cores—and 24 threads—with 2.1 GHz, and 1 TB of RAM. The data is stored on one HGST Ultrastar 7K4000 HDD. Ubuntu 16.04.3 LTS is used as the underlying operating system. The described delta tree approach and optimizations are implemented as extensions to JODA.

The **dataset** used is a 109 GB file containing a sample of the raw Twitter JSON stream[1]. It consists of 29,634,708 JSON documents, where each document has between 7 and 348 attributes, containing every JSON type. The documents are split into two major groups. Around 23.5 million (79.33%) documents are normal tweets, while around 6.1 (20.67%) million documents are deletion instructions. The tweets have a varying number of attributes, depending on their status, e.g., retweets and favorites, while the deletion documents consist of seven attributes.

5.4.2 Delta Hierarchy Creation and Shared Reads

In the first evaluation, we execute a number of queries that illustrate the core features of our approach. The first query in Listing 5.2 loads the Twitter dataset. Then a collection is created which adds one member to the user object of the previous dataset. Derived from this collection, another attribute is added to the user object. In the following query, only the data added in Q2 is used in an aggregation. Then the member count of the user object is queried in the next two queries. The last query copies the shared user object into a new collection.

[1] `https://developer.twitter.com/en/docs/labs/sampled-stream`

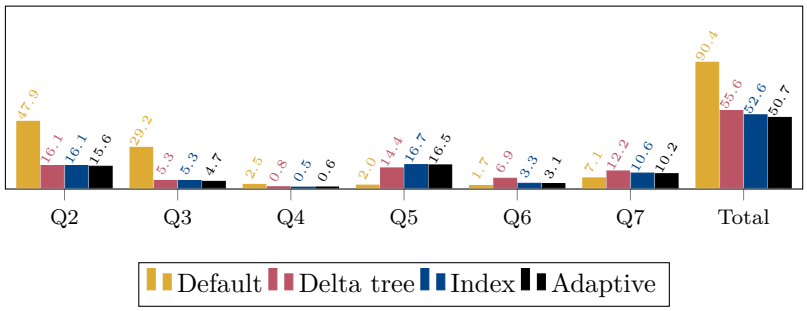


Figure 5.4: Runtime of different execution methods (in s)

```
Q1: LOAD t1 FROM FILES "/data/twitter";
Q2: LOAD t1 CHOOSE EXISTS('/user')
    AS *,('/user/v1':1) STORE t2;
Q3: LOAD t2 AS *,('/user/v2':2) STORE t3;
Q4: LOAD t3 AGG ('':SUM('/user/v1')) STORE a;
Q5: LOAD t3 AS ('':MEMCOUNT('/user')) STORE c1;
Q6: LOAD t3 AS ('':MEMCOUNT('/user')+1) STORE c2;
Q7: LOAD t3 AS ('':'/user') STORE user;
```

Listing 5.2: Queries iteratively changing an object and reading it

We compare our introduced approaches against the default execution method, which copies and modifies the full JSON documents. The delta tree approach is based on our implementation within the system, as explained in Section 5.3. The index approach uses the same implementation, but with enabled virtual object indexing, as described in Section 5.3.4. Lastly, the adaptive approach—described in Section 5.3.5—is compared.

Runtime

The query time plot in Fig. 5.4 is omitting the first data import query, as it is unaffected by the execution method and requires the same time for all of them. As we can see in Fig. 5.4, the implementation without delta trees, which transforms the documents by copying the source data, requires for queries Q2–Q7 about 90.4 seconds. The delta tree implementations on the other hand executed these queries significantly faster, with 55.6, 52.6, and 50.7 seconds in total respectively.

For the modification queries Q2 and Q3, the default execution method requires up to 5.5× as much query time as the delta tree approaches, as the whole document has to be copied, and memory has to be allocated. Q4 is the first reading query and aggregates one of the previously added attributes. We expect reading operations on shared objects to be slower for delta trees. But as this function only reads one of the atomic values, we see that the delta tree approaches require only around 1/5th of the query time. This is possible because the delta trees—in this case—are way smaller than the full document tree, which means that fewer comparisons are needed to find the required value. In Q5, we now read the `/user` attribute, which is shared over multiple delta trees in the non-default

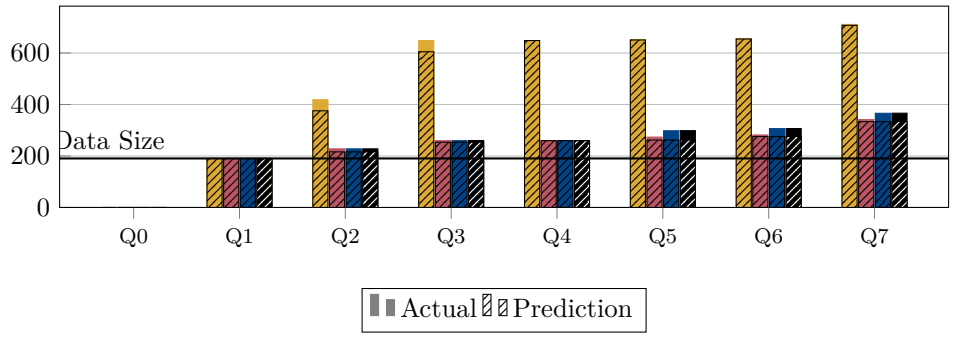


Figure 5.5: Memory consumption (GB) of different execution methods

implementations. Here we see the drawback of our approach. As the delta trees have to simultaneously traverse multiple trees with additional algorithms, the query execution time is approximately 7 times longer than the normal delta tree execution. The indexed delta tree needs additional time for creating the indices on the first traversal of a shared object. Q6 shows now that creating the index pays off, as we now require less than half the execution time of the normal delta tree approach. This is a significant speed up, but still slower than the default reading speed, which is to be expected. Query Q7 now copies the shared attribute into a new collection. As before, the default execution method is faster, but the indexing helps to close the gap.

Memory Usage

Fig. 5.5 immediately shows the main advantage of delta trees. All implementations start with the same memory usage after the data import query Q1. For Q2 and Q3, the default implementation has to copy the data it wants to modify. This of course has a heavy impact on memory usage. The delta tree implementations on the other hand only have to store the additional data, with a little overhead for the internal tree structure.

When using the object index, memory usage is increased by up to 9% compared to the normal delta tree implementation. But compared to the baseline data ($190GB$ in-memory), delta trees with indexes only required an additional 87%, while copying the data increases the memory usage by 274%.

Table 5.1 compares the costs calculated by the theoretical cost model with the model adapted for our specific implementation, from Section 5.3.5 and the actual value measured in the system. All costs are given in (expected) increase of memory consumption in MB. The first query is the parsing query, which is the same for all execution methods. We thereby cannot compare any costs or make any choices. For Q2 and Q3, which add one attribute to a given document, the theoretical model gives a lower estimate for the delta trees than the real value. The adapted model on the other hand is much closer to the correct value. For both queries, the delta trees would be correctly chosen by both the theoretical and adapted model. As aggregations depend on multiple documents, we cannot compute it with delta trees and hence, make no estimation for this query. The

	Theoretical Model			Adapted Model			Actual		
	Default	Delta Tree	Choose	Default	Delta Tree	Choose	Default	Delta Tree	Choose
Q1	-	-	-	-	-	Default	-	-	-
Q2	180,385	1,881	Delta Tree	185,077	25,578	Delta Tree	231,765	26,331	Delta Tree
Q3	181,137	1,881	Delta Tree	186,457	25,578	Delta Tree	233,898	26,331	Delta Tree
Q4	-	-	-	-	-	-	-	-	-
Q5	376	376	Default	2,257	2,257	Default	2,257	2,257	Default
Q6	376	376	Default	2,257	2,257	Default	2,257	2,257	Default
Q7	50,191	50,191	Default	51,695	51,695	Default	54,876	54,876	Default

Table 5.1: Cost model comparison (in ΔMB)

```
Q1: LOAD t1 FROM FILES  "/data/twitter";
Q2: LOAD t1 CHOOSE !EXISTS('/delete')
    AS *, ('/user/id':HASH('/user/id')) STORE hashed;
Q3: LOAD t1 CHOOSE EXISTS('/delete')
    AS *, ('/delete/status/user_id':
        HASH('/delete/status/user_id')) STORE hashed;
Q4: LOAD hashed AS ('':'/user/id') STORE hashes;
Q5: LOAD hashed
    AS ('':'/delete/status/user_id') STORE hashes;
Q6: LOAD hashes AGG ('/min':MIN('')), ('/max':MAX(''));
```

Listing 5.3: Hashing user IDs in different documents

last queries only create a single value and do not select other values from the base document. This results in equal costs for delta trees and default executions. But as delta trees incur an overhead during execution, the default execution is chosen for all of these queries.

The adaptive algorithm was only executed with indexing enabled. As the adapted model makes the same choices as we did without the model, the memory consumption and runtime is similar to the index solution.

5.4.3 Adaptive Execution Method

In this experiment, we evaluate the capabilities of the adaptive execution method. The queries shown in Listing 5.3, once again, import our Twitter dataset and then replace the user IDs with a hash value. Queries Q2 and Q3 replace the user IDs in the differently structured tweet and delete objects and store the adapted documents in the same collection. Q4 and Q5 then extract these hashes into their documents with only the hash value as the root. Lastly, Q6 aggregates all hash values to find the minimum and maximum.

Runtime

Fig. 5.6 shows the runtime of each of these queries executed by the default, delta tree, delta tree with indexing, and adaptive execution methods. As before we omit the parsing step as it is uninteresting for this evaluation. In Q2, we replace the user id of all tweet documents. These documents are large and hence, require a lot of time alone for copying the source documents before modifying them in the default execution. All delta tree implementations require less than half of the query time to evaluate this query.

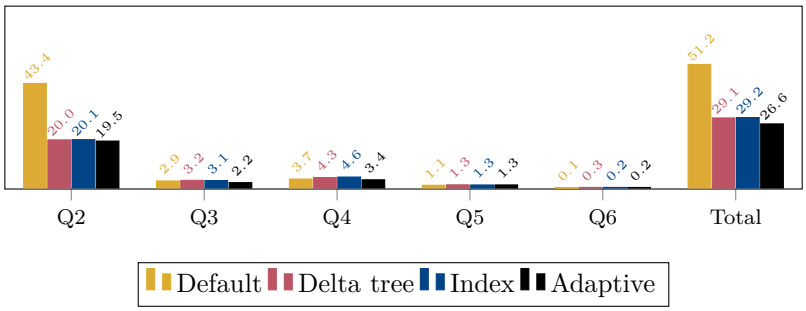


Figure 5.6: Runtime of queries for different configurations (in s)

	Theoretical Model			Adapted Model			Actual		
	Default	Delta Tree	Choice	Default	Delta Tree	Choice	Default	Delta Tree	Choice
Q1	-	-	-	-	-	-	-	-	-
Q2	179,632	1,881	Delta Tree	170,783	25,578	Delta Tree	189,009	26,331	Delta Tree
Q3	1,602	686	Delta Tree	1,798	9,702	Default	2,136	9,996	Default
Q4	376	376	Default	2,257	2,257	Default	2,257	2,257	Default
Q5	98	98	Default	588	588	Default	588	588	Default
Q6	-	-	-	-	-	-	-	-	-

Table 5.2: Cost model comparison (in ΔMB)

Query Q3 does the same for the delete documents. They are very small and there are only a few of them in the dataset. Hence, this query is evaluated fast. Here the delta tree approach is slightly slower, even for a modifying query. This is the case, because of the nested structure with only a few attributes, for which the implementation allocates placeholder memory. The adaptive algorithm correctly predicts this situation, as shown in Table 5.2, and chooses the default execution method.

Queries Q4–Q5 now read the created datasets and extract the hash values. For Q4, the default and adaptive execution methods are again faster, as the model correctly predicts an advantage for this query. The delta tree approaches are slightly slower.

In total, the delta tree approaches are faster than the default execution method, as the complete document set does not have to be copied. The adaptive approach improves this total runtime, as it predicts the advantages of the default execution for some queries.

Memory Usage

The memory requirements of all approaches for the given queries are shown in Fig. 5.7. Once again, we can see that delta trees require less memory for large documents where only parts of them are changed. Table 5.1 compares the costs calculated by cost models. The first query is again the data import and cannot be compared. The second query replaces a single attribute within a large document and a large object. For these kinds of queries, the theoretical model, as before, estimates a too low value for the delta tree execution. The adapted model has a better estimation, which correctly chooses delta trees to perform this query. Q3, on the other hand, replaces a single attribute in a

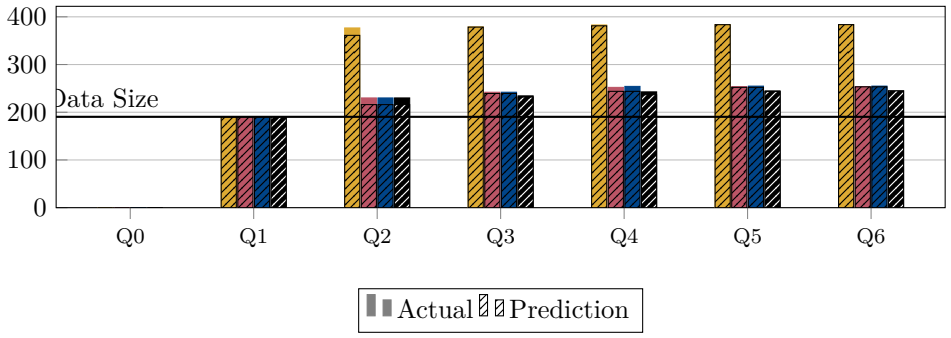


Figure 5.7: Memory consumption (GB) of queries for different configurations

twice-nested small object (2 attributes) within a very small document. The theoretical model assumes that the costs of delta trees are lower, as it would still prevent the duplication of some data, and chooses them to execute the query. But the adapted model correctly assesses that the overhead of object creation is larger than simply duplicating the data, and chooses the default execution method. The remaining queries, as before, can either be not compared or have the same costs for both execution methods and are thereby executed by the default method.

The adaptive implementation chose to execute Q3 with the default execution method. This results in the lowest memory consumption of all tested execution methods.

5.5 Summary

In this chapter, we introduced the concept of delta trees, for materializing only the differences of a document transformation, to reduce the memory footprint of exploration systems. We explained the basic idea, theoretical model, and specific implementation details, based on JODA. Additionally, we introduced improvements to the systems to mitigate the performance bottlenecks introduced by the approach. Delta trees enable systems to perform queries, with a fraction of the required memory, leaving original datasets intact. An adaptive algorithm was introduced, which helps to avoid the drawbacks of this approach. This increased transformation performance is bought, by sacrificing performance in some read-heavy operations, especially copying shared objects. We mitigated this performance loss by creating a special index for these shared objects. This increased the read performance, at cost of slightly higher memory usage.

5.5.1 Potential Extensions

In Section 5.2.4 we introduced a cost model for memory consumption. But the proposed delta trees also incur an increased runtime cost in some cases and improve runtime in others. We are working to create a model to estimate the impact of delta trees on the query runtime. Currently, we are able to accurately estimate it for most cases. But important corner cases remain, for which the

estimation is too inaccurate. When this estimation is working properly, an execution method could be created, which optimally uses available memory and CPU resources.

We introduced virtual object indices in Section 5.3.4 to reduce the runtime of reading queries for delta trees at the cost of slightly increased memory usage. But delta trees also support the modification of arrays. Currently, arrays are always materialized completely if they are modified, as the performance overhead of traversing them through a large delta hierarchy, like objects, is too large. To solve this bottleneck, a virtual array could be created, similar to the introduced virtual objects. This would decrease memory consumption, compared to completely materializing the array while giving a similar read performance.

Our cost model as introduced in Sections 5.2.4 and 5.3.5 currently disregards any virtual objects that may have been or will be created by our implementation. We could adapt the model to predict when the indices will be created and estimate how much memory they will require. This would improve the prediction if indexing is enabled.

Chapter 6

Adaptive Indexing

The key idea in JODA is to organize documents into immutable containers that are independently and perfectly in parallel processed by the available CPU resources. While this achieves virtually unlimited vertical scalability, as expected, the lack of index structures vastly wastes computational resources for low-selectivity queries. At the same time, in the spirit of raw data processors like the NoDB approach by [74], introducing upfront indexing would violate the core design principles and intent behind such systems. In this chapter, we describe how adaptive indexing can be useful for processing semi-structured data and how this is implemented and evaluated as a proof-of-concept in JODA. The idea of adaptive indexing is not new, in fact, there is ample work [38,41,75] that describes how indices are iteratively constructed, refined, and maintained over time.

Most importantly, the indices should be created automatically without any human intervention. The creation of the indices should be lightweight, such that the creation of potentially unnecessary indices does not negatively impact query runtime. After the creation of the index, each successive usage should improve the index until it converges to a full index. The query runtime should already be improved when using the index after the initial creation phase. Lastly, the index should make use of the document structure without an exhaustive analysis step.

As an example, assume a data scientist receives a large dataset of Twitter messages in JSON format. While the general structure of the documents may be extracted from the documentation, the specific properties of this dataset may not be known. The data scientist read in the documentation that the stream dataset is composed of tweets and delete statements, which have different top-level attributes. In a first query, all documents containing the top-level attribute "/delete" are selected and aggregated, while a second query checks for the existence of the "/user" attribute. During the first query, the system could start building a structural index of all top-level attributes first query and, thus, improve the runtime of the second.

We consider an immutable set of semi-structured documents with potentially unknown and non-uniform schema. The documents are loaded into the main memory and can be then queried by users. In this chapter, we present an adaptive indexing scheme, developed with S. Lang during his Master's thesis work [76], which will create and improve indices on these documents with each query. To achieve this goal we make the following contributions:

- We propose an adaptive structural index, which keeps track of the structure of the documents.

- We further propose two proof-of-concept adaptive content indices, for string and number data types, respectively:

  - One adaptive trie, based on patricia tries by Morrison [77], for prefix/equality/comparative string predicates.

  - One adaptive histogram-like number index for equality/comparative number predicates.

- We give details on a query evaluation algorithm that can use multiple indices for a single query.

This chapter is organized as follows. Section 6.1 gives an overview of our approach and presents the architecture of the system. The exemplary implementation is then explained in Section 6.2. Section 6.3 evaluates the performance of these indices and discusses their strengths and weaknesses. At last, Section 6.4 will summarize the extension.

6.1 Overview and Preliminaries

To achieve the goal of adaptive indexing in JODA, three main components are designed. Differences in the structure of JSON documents are handled with a **structural index** that gathers information about the availability of different JSON types in specific paths. The index processes this information in a way to enhance queries for simple JSON types, like null and Boolean values, to optimize future requests that concern the structure of JSON documents, and to prune documents that do not have the required attribute and types for the given queries. The other two components are **content indices for strings and numeric values**. Both integrate with the structural index, are specialized for each value type, and support complex operations on the data. All indices are created on-demand and incrementally improved while incoming queries are processed.

The structural index reflects the structure of an arbitrarily large set of documents. Each attribute in the document trees may have a node in the structure index with additional information. The structural index also manages the available content indices. If the system deems it advantageous, the content indices are initialized at attribute nodes. Figure 6.1 shows a simplified example of the index composition.

For example, a query may contain the predicate `/user/name == "Mike"`, which checks a given document's paths for equality with a specific string value. First, the node of the structural index for the path `/user/name` is looked up or created if it does not exist. Only documents known to have this attribute with a type of string are considered for further evaluation. The structural index can if possible, further create and consult a fitting content index. All content indices always refer to a specific JSON path. Therefore, they are directly owned by the structural index node. When a structural index node is created, it collects information about which documents contain which data types at the specified path. In this case, a string index could be used to efficiently answer the query, and potentially improve future string queries on the given attribute.

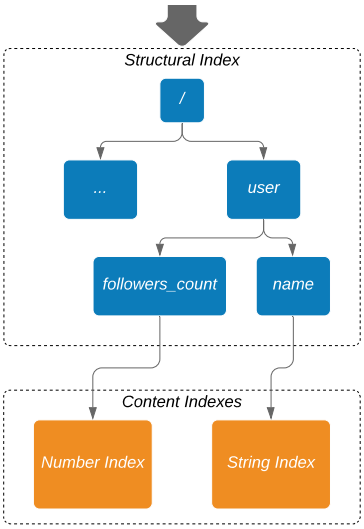


Figure 6.1: Predicate execution

6.2 Adaptive Indexing using Structure and Content Indices

As laid out in Section 6.1, a structural index and two content indices are implemented to support basic operations on strings and numbers. In general, these indices should be of adaptive nature and are created and executed while running queries without further configuration beforehand. Creating the indices should be lightweight and keep the querytime impact to a minimum. Each query to an index should improve the situation for following similar queries, until a full index is created.

6.2.1 Handling Document References and Document Sets

At the leaf nodes of each index structure, a set of matching documents is stored. In our indices, depending on the situation we used three possibilities of storing these sets. The system splits collections of documents into containers, with each container having its own set of indices. As containers are immutable, we can identify a document by its position index within the container.

If a set of documents is very small, we can use a document index list, which is a simple list of all positions as integers. This works well, for small sets, as we can iterate this list and access all documents by their given index. But this approach requires a lot of storage if the document sets become larger. To reduce the storage requirements, we can also store the indices as ranges. If many sequential documents are contained in the list, using ranges can free up a lot of space. In the best case, all documents are contained in the set, which can be represented by two integers. On the other hand, if the result set is large, but it does not contain large consecutive ranges (e.g. every second document), then the range set could have a worse storage requirement. In this case, we can use a bit-vector to represent the set.

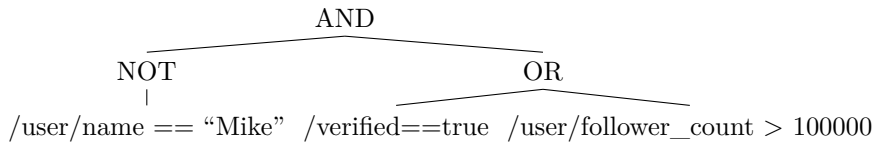


Figure 6.2: Example query predicate

For example, the index list $[1, 2, 5]$, range list $[(1, 2), (5, 5)]$, and bit-vector $[1, 1, 0, 0, 1]$ all represent the same set of documents and can be used interchangeably and be converted into each other.

6.2.2 Query Evaluation

To be able to use indices for the selection phase of query evaluation, we have to analyze the filter predicate for compatibility and execution effort. Our system parses the textual query predicates into a tree structure or nested predicate functions, as shown in the example in Figure 6.2.

We traverse this tree using a simple visitor pattern, in order to augment it with index specific information. For each predicate function, a list of applicable indices with their estimated execution cost is stored. Additionally, important parameters for each supported index are already extracted and stored. The extracted attributes serve as a decision parameter on which index to execute later, and offer potential to optimize the call in the index. Some predicate functions will also be combined in this step, if supported. For example, if one knows that the result of a function will be negated, a more appropriate index call could be chosen instead of inverting the result afterward. Additionally, a range query with upper and lower bound can be executed at once, instead of using two index calls.

By default, the filter step is executed by checking the predicate tree for each document. Each predicate function returns true or false, given a document. The system checks the root node of the predicate tree, which in return calls all of its potential children. The implementation of the “AND” and “OR” functions already includes lazy execution, as it will stop checking additional children as soon as the result is clear.

As indices return all matching documents as a set for a given predicate function, the execution of index-only filter steps is slightly different. If all predicate functions are supported by indices, the result sets will be retrieved bottom up. For “AND” or “OR” nodes, the children are executed first, and the result document-sets are merged. The “NOT” function will invert the result set of its child function. The set returned by the root node is then the final result.

If the predicate tree contains both index-supported and unsupported functions, additional logic is required. The naïve approach would be to check all predicates one by one and merging the results afterward. While index checks would be fast, each unsupported predicate would trigger a complete document scan with predicate evaluation. Therefore, we implemented a mixed-execution lazy evaluation approach. For this approach an additional lazy-evaluation tree is created. The

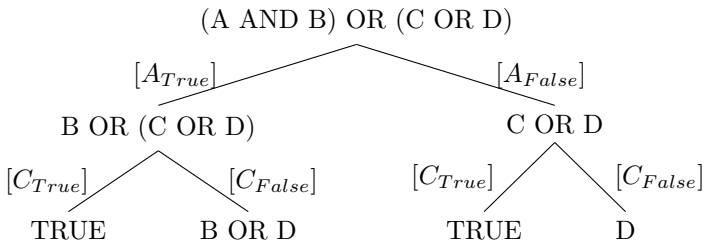


Figure 6.3: Example lazy-evaluation tree

root node of this tree represents the full logical filter-predicate. Each index-supported predicate adds an additional level to the tree with two children for each node in the current level. One child represents the same predicate where all occurrences of this predicate function are replaced by true, and one by false. The predicate is then simplified using Boolean algebra. This decision tree will then be used to execute only the necessary unsupported predicate functions.

For example, the filter predicate given in the query could be (A AND B) OR (C OR D), where A and C are index-supported sub-predicates. Figure 6.3 now shows the lazy evaluation tree of this predicate. The results of each sub-predicate execution is cached for the lifetime of the query, to prevent duplicate execution of the same predicates. The documents of the data-set are then iterated and depending on their index results, evaluated. For example, documents where the function A returned false, but C returned true can be added immediately to the final result set. But for documents where C returned false, the unsupported function D has to be evaluated on the document. In this example, we can also see that the order of tree creation can be optimized. If the C function would had been added first, only one sub-tree would have to be created. Optimizing the lazy-evaluation tree is a potential future improvement.

6.2.3 Adaptive Structural Index

As we are working with potentially heterogeneous data, the structure of the documents may differ. Query predicates always implicitly check for certain constraints on the structure of the document. For example, the predicate `/str == "a string"` will check if the document contains the first-level attribute "str" of type `String`, before evaluating the equality. Documents without this attribute, or storing other types of data within the given attribute, would not have to be considered for this predicate.

To improve the performance of this search-space pruning, we propose the *union tree*, a tree mirroring the structure of the union of all indexed documents. In each node, we store for each type the set of documents containing the path with the given type.

We decided to use a tree structure, as it directly represents the underlying data. This allows us to use already existing components of JODA with only minor adaptions. Figure 6.4 shows an example of a partial union tree.

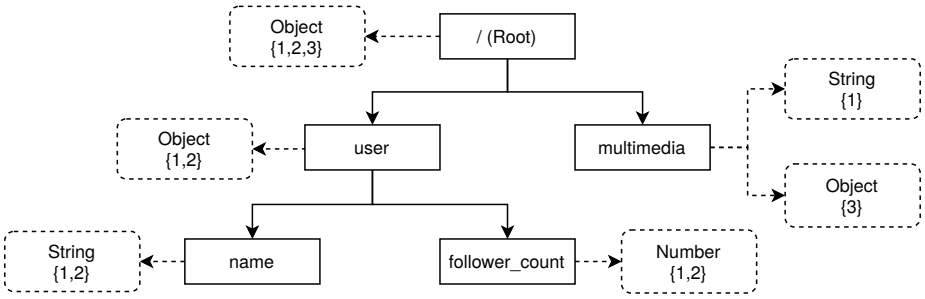


Figure 6.4: Partial union tree with document sets

The union tree is initialized without any nodes. Every time a new path is used in a filter predicate, the tree is expanded up to the provided node. If the next query requires path "/user/location/street", we only have to check all documents that are known to contain "/user". The tree will then be expanded to also contain the newly queried path—with all of its parents.

Using this tree, documents can be pruned from future query evaluation and some predicates may even be answered completely. The union tree additionally manages the content indices. Whenever a predicate is executed on an already existing union-tree node, the query predicate is, if supported, passed on to an index. If no index is registered at the node, the most fitting one is created. Only content indices that are relevant to the current predicate are initialized. Algorithm 3 summarizes the filter predicate evaluation using the union tree. For our approach, we only implemented adaptive indices, which are improved with every further predicate execution. But the implementation is modular and allows the addition of arbitrary content indices. As proof of concept, we implemented one content index for strings, and one for numerical values, which will be introduced in the following.

Data: Predicate-tree node P, Union tree U, Documents D
Result: Result document set
if *P is leaf with path p* **then**
  Create and traverse nodes to p in U;
  if *P compatible with an index I* **then**
    Create I if not exists;
    return Evaluate P with I;
  else
    return Evaluate P for each d in D;
  end
else
  S = Recurse with children of P;
  return merge S;
end

Algorithm 3: Union tree filter predicate evaluation

Index	Value
5	a
7	hello
8	hell
33	xenias
344	albert
556	bye
585	xenial
2893	xylophone

Table 6.1: Example mapping of document indices to their string values

6.2.4 Adaptive Trie Content Index

The short-string index implementation should work well in an adaptive manner without a costly analysis of all strings and needs to support basic string queries. Hence, we opted for the usage of a trie, as it can easily be used as an adaptive index. Additionally, it supports the most functions also supported by the JODA query language, like prefix matching and lexicographic ordering. Each path from the root node represents a different string prefix. To reduce the space consumption, tries can also be compressed as shown by Patricia tries, introduced by Morrison [77]. When several nodes in a path have a single descendant and do not constitute a word at this point, they can be merged into one node with several characters. In our implementation, each node representing a completed word, will be augmented with a set of documents, which contain this word in the given attribute.

Instead of fully building the index for all documents, we incrementally improve it. With each additional call to the index, another level will be added to the trie, using all the relevant documents. An initial string predicate results in the creation of a trie, having the first character of all documents. For some example values shown in Table 6.1, this first query will create the nodes “a”, “b”, “h”, and “x”, as visualized in Figure 6.5. For each node, we store document sets for documents where there is an exact match, and a prefix match. When all children of a node are extracted, the prefix document set can either be discarded, as the information is now stored in the children implicitly, or kept. Keeping the document set may improve performance at the cost of memory, as for some retrieval operations the child-trees do not have to be traversed. Per default, we opt to discard these values, as the performance improvements are too specific for only a few predicates, while negatively impacting the memory footprint, and hence the ability to create more content indices. If a node of a trie cannot be expanded further with the given set of documents, it is marked as finished. To improve the accuracy and usefulness of the trie, we adapted it to always fully index—and compress—the full query keyword given by the query. The complexity of the first predicate execution is not dependent on the queried value length and keeps the call relatively close to the runtime without any index structures. The adaptive trie makes use of a proven search tree concept and transforms it into an adaptive approach. While it does not store all values, like some other partial indices, it stores partial values that have been used by predicates to help improve future similar predicates.

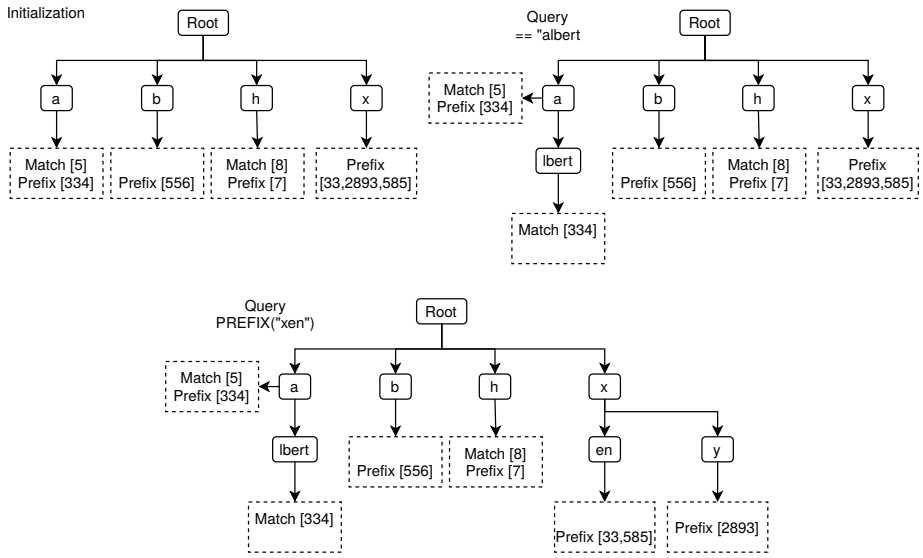


Figure 6.5: Adaptive trie example

To query an additional value, we traverse the tree. If we encounter an unfinished node, all documents referenced as prefix matches are loaded and the trie is expanded further as explained previously. In our example in Figure 6.5, the index is queried with an equality predicate for the string "albert". As described, the queried string is expanded completely and then compressed. Finally, a prefix query for "xen" will then create two additional nodes, one for the complete and compressed query value and another one for the only other direct descendant of the "x" prefix.

As the relevant set of documents can only be filtered further, each child's document sets will be a subset of the parent node. As a consequence, if the trie did load documents once, no further documents need to be loaded. This implementation supports the following predicate functions:

Equality By traversing the trie—and expanding it on demand—to the given query string, we extract the document set where all documents have exactly the given value as attribute.

Inequality is implemented by retrieving the equality result and inverting the result set.

Prefix Matching Similar to equality matches, we traverse the trie and retrieve not only the exact matches, but also the partially matching documents.

Lexicographical comparison To evaluate a greater/smaller-than predicate, we traverse the trie to the given string value—again while potentially expanding it. Then the document sets of the whole sub-tree to the left or right of the given node—and parent nodes—have to be collected. If the index kept the partial matching document sets of the parent nodes while expanding, then fewer nodes have to be traversed to collect the result.

In any case, documents are only loaded at most once, and document values only checked if a node needs to be extracted.

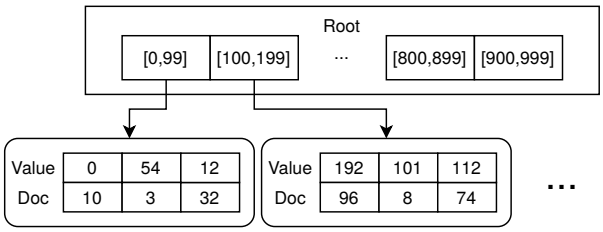


Figure 6.6: Root node with a value range of $[0, 999]$ and k = 10

6.2.5 Adaptive Histogram Tree for Numbers

As previously described, there exist many adaptive numerical indices. For our system, we opted for implementing an adaptive tree index with histogram-like properties, as it is easy to implement in an adaptive manner and already improved the performance of number-based predicates significantly. It also supports all filtering number predicates implemented in JODA. Equi-width and equi-depth histograms are the most widely known and simplest forms of histograms. While equi-width histograms are simple and fast to create, they are not well suited for skewed data which is often found in real-world applications. In most cases, it is more useful to have an equi-depth histogram that divides the values into similar-sized buckets with varying ranges. For example, Sprenger *et al.* [78] use equi-depth ranges to divide data in an index tree. But creating equi-depth histograms requires a large up-front computational overhead.

The idea of our implementation is to create an equi-width histogram, whose buckets will be refined in further queries until it approximates a B^+-tree. Each node in this tree is composed of up to k ranges—similar to histogram buckets—which contain the numerical attribute values together with the document reference. Initially, the tree is created with a root node, which contains k equi-width ranges within the found numerical values. Figure 6.6 shows the creation of the index with $k = 10$ and a value range of $[0, 999]$.

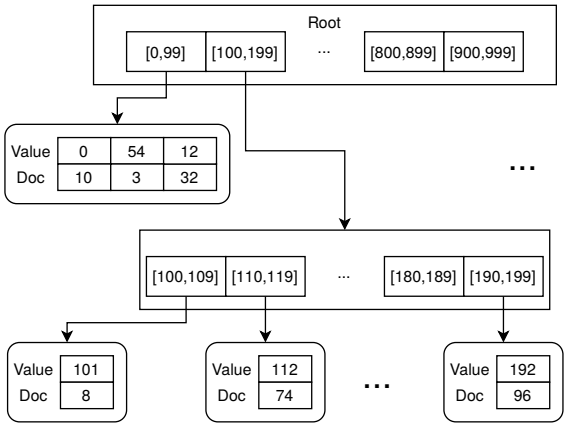


Figure 6.7: Index with a value range of $[0, 999]$, k = 10, and t = 2, after a split

If an additional query predicate is evaluated using the index, for instance, `/user/followers < 150`, the fitting ranges are selected, and their document sets are returned, after being optionally filtered if required. If a filtered range contains more values than a specified threshold t, it is further split into k sub-ranges. This ensures that the first creation of the index is fast—as all values have to be loaded and compared anyway—while the index is adaptively improved with each query. If the data is heavily skewed, predicates selecting outside of the bulk are immediately very fast, without any additional processing. Predicates selecting an overfull range trigger another cheap refinement, which is always cheaper than the first. This index supports all numerical predicates. Additionally, in future work, the query evaluation could be improved to use the index to immediately answer aggregation queries like min, max. The index does not need to support deletion, updates, or inserts, as the containers and documents are immutable, but a sketch of how these operations could be implemented is shown in Section 6.2.6.

To improve the performance of edge cases, each range additionally stores the minimum and maximum value contained within. Instead of using the previous range for splitting the values into k sub-ranges of same width, we split the $[\min, \max]$ range. This prevents cases, where dense values over the threshold t would be split multiple times with many empty ranges. Figure 6.7 shows the previous index, after a query selected values from the second range.

6.2.6 Mutable Indices and Memory Management

As our system works on immutable containers, which are sub-sets of documents, we did not implement any delete, update, or add operations for our indices. But adding them would be straightforward. For example, adding a new document could be achieved by simultaneously traversing the structure index with the given document, and adding the document reference to each matching node. This would also call the insert function on the content indices, which would have to support inserts. Adding documents to our number index would simply involve adding them to the correct bucket and splitting it, if it grows beyond the threshold, while the string index would simply add the document reference to the fitting nodes. A document could also be removed from the index in a similar fashion. Here, special care would have to be given, when nodes become empty by removing the document. Then the node would have to be removed, which may result in additional deleted nodes further up the tree. Updates of specific documents would be harder to implement, as the complete structure and content could change. Instead of handling this case, we would implement it by removing and inserting the changed document.

As indices are bound to containers, they would only be removed if a container is removed from the system. This may lead to many indices being created and filling the memory. Hence, we implemented a simple memory manager, which keeps track of the sizes of all created indices and when and they have been used last. It can be configured with a maximum memory size, which defaults to all available memory. If a new index has to be created, or an existing index needs to grow, then the memory manager will free up memory by removing existing indices in Least-Recently-Used order, until enough memory is freed.

```
Q0 LOAD TWITTER FROM FILE "twitter.json"
Q1 LOAD TWITTER CHOOSE ISNUMBER('/quoted_status/id')
Q2 LOAD TWITTER CHOOSE EXISTS('/quoted_status/user/location')
Q3 LOAD TWITTER CHOOSE '/quoted_status/user/verified' == true
Q4 LOAD TWITTER CHOOSE TYPE('/quoted_status/user/followers_count')
    == "NUMBER"
```

Listing 6.1: Structure queries on quoted Tweets

6.3 Evaluation

The core version of JODA as well as the adaptive indexing extensions are implemented in C++17. For the evaluation, a Twitter dataset of 109 Gigabyte is used[1]. The dataset contains 30 million JSON documents that represent normal Tweets (∼80%) and some documents represent deleted (∼20%). Deleted Tweets have a completely different document structure with exactly 7 attributes. Some Tweets are Retweets or responses to other Tweets and contain additional objects with information about the original Tweet. The dataset is initially loaded from disk where they occupy 8800 files, each with 9–18 MB. The initial loading is not part of the runtime measures below.

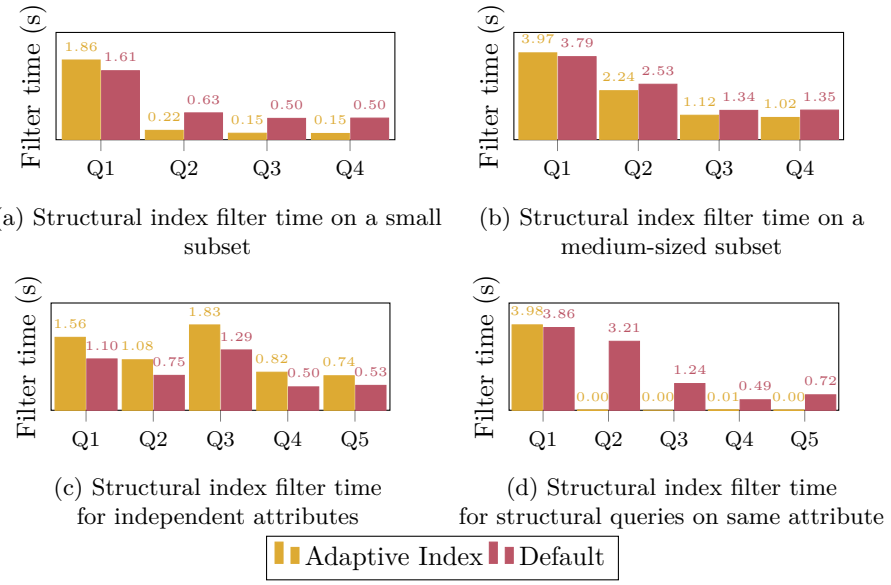
The queries are executed on a server running Ubuntu 16.04.6 LTS with four Intel Xeon E7–4830 v3 CPUs. Each CPU has 12 cores with 24 hyper-threaded cores that run at 2.1 GHz. About 1 TB of RAM, composed of 33 RAM modules at 2400Mhz, is installed.

6.3.1 Structural Index

Figure 6.8 presents the runtime of the query filter step for different queries on different subsets of documents. An example query set using different structural properties to filter the documents is shown in Listing 6.1. Q0 loads the document set into the main memory, from where it will be used by forthcoming queries. Importing the documents takes on average 144.28s. The original data loaded from disk does not contain any index information, hence, no matter if adaptive indexing is enabled or not for the later-on queries, there is no impact on the initial loading time. After the first query to a node, all subsequent queries are executed to descendants of that node. The descendants can use the information about the document structure gathered with the first query. In this case, about 5–10% of the Tweets contain the quote of another Tweet. To only show the effect of the structure, the query predicates always access a different child node, so there is never an already existing union node with organized values for that path. This also means that every query needs to perform some extra work to create the union tree node. None of the queries are supported by an additional content index and will not cause extra work in that regard.

The results, in Figure 6.8a, show the additional time the first query needs to create the union node. All other queries also need this additional time but are faster because they know which 5–10% of documents the need to look at given the `/quoted_status` node. Q2–Q4 are already considerably faster without any

[1] `https://developer.twitter.com/en/docs/labs/sampled-stream`



(a) Structural index filter time on a small subset

(b) Structural index filter time on a medium-sized subset

(c) Structural index filter time for independent attributes

(d) Structural index filter time for structural queries on same attribute

Figure 6.8: Structure index benchmark results

content indices. Note that in general, e.g. because JSON data is already loaded in memory from a previous query, the execution time of the default executor may also vary. The memory usage of the index after the four queries was 227.8MB, which is 0.2% of the total data size.

Figure 6.8b shows similar queries to the first structural test (Listing 6.1) but on a different subset of attributes. The structure of these attributes is nearly identical to the previous queries, but they are contained in over 50% of all documents. Hence, the memory usage of the index is the same as in the previous benchmark. We can see that the structural benefits are not as prominent as in the first benchmark in Figure 6.8a, but the index creation already pays off as soon as a predicate accesses a sub-tree more than once.

Figure 6.8c showcases how much penalty one has to pay to create a new structural index node. All queries in this benchmark access completely different paths that concern different subsets of the documents. That means all queries have to create a new union tree node without any benefits from previous queries. The resulting index is also much larger with 1.18GB ($\approx$ 1% of total data), as each query creates a new subtree. Q3 and Q5 also need additional processing time to initialize an adaptive histogram respective the first level of the adaptive trie. As expected, the default execution method outperforms in every query, as no initial index creation has to be performed. On the other end of the spectrum, Figure 6.8d shows the filter times of five queries, each querying structural information—like existence and type—of the same attribute. For these queries, after the structure index is created, no documents have to be loaded, and the query can be answered with the structure index alone. This results in negligible filter times and a small index with 144.9MB ($\approx$ 0.13% of data).

```
Q1 LOAD TWITTER CHOOSE '/user/screen_name' == "RimaTupick52039"
Q2 LOAD TWITTER CHOOSE '/user/screen_name' != "RimaTupick52039"
Q3 LOAD TWITTER CHOOSE '/user/screen_name' == "J03LP1M3NT3L"
Q4 LOAD TWITTER CHOOSE STARTSWITH('/user/screen_name', "Rima")
Q5 LOAD TWITTER CHOOSE '/user/screen_name' == "
    non_existing_username"
Q6 LOAD TWITTER CHOOSE STARTSWITH('/user/screen_name', "sello")
Q7 LOAD TWITTER CHOOSE STARTSWITH('/user/screen_name', "ma")
Q8 LOAD TWITTER CHOOSE '/user/screen_name' < "a"
```

Listing 6.2: Different string queries on the same attribute

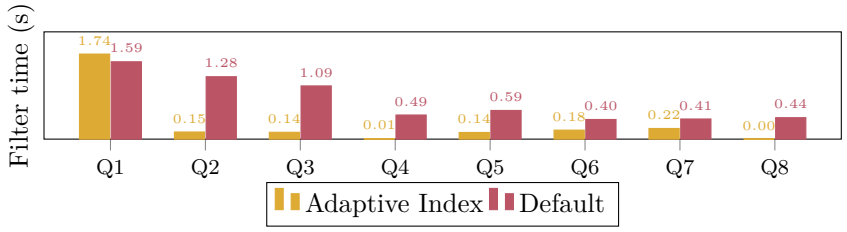


Figure 6.9: Adaptive trie filter time

6.3.2 Content Indices

In the following, we evaluate the performance impact of the content indices in addition to the structure index.

String Index

The benchmark shown in Figure 6.9 tests the performance development when querying an adaptive trie multiple times with different parameters (Listing 6.2). To eliminate other factors, only one path is queried. The first query will create the structure index node and initialize the trie by creating the first level. All subsequent queries are directly forwarded to the trie structure. The queries contain all available trie operations: equals, not-equals, starts with, and lexicographic comparisons.

Q2 is very fast despite the huge result size since Q1 already created all necessary trie paths. In this case, the index creation immediately pays off after the same attribute is queried for the second time, as non-indexed string predicates are relatively slow operations. The same is valid for Q4. Q5 and Q6 both return empty results, but they need to extend the trie in any case. Q7 is a short query to a larger result, which also has to be loaded an extracted in the trie, while Q8 showcases the use of lexicographic comparison. After evaluating all queries the trie index, together with the structural index required for management, requires 339.3MB ($\approx 0.31\%$ of total data) of memory.

Figure 6.10 shows a distribution of the query time for 121 queries to `/text`, an attribute containing the text of the whole tweet. The queries use all supported string predicates, with randomly chosen values extracted from the dataset. As we can see, the first query is approximately 50% slower than the default execu-

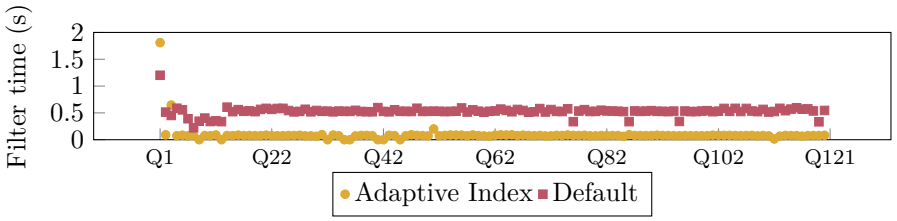


Figure 6.10: Adaptive trie query response time

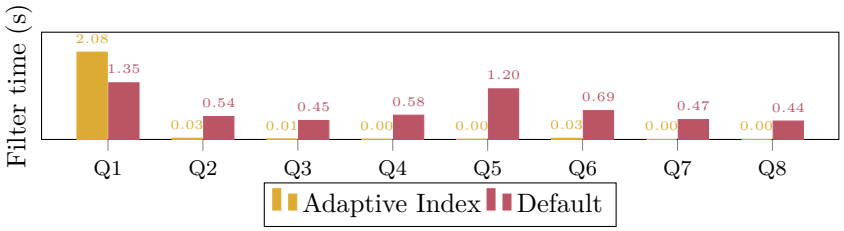


Figure 6.11: Histogram index filter time

tion, which results in massively improved query runtime for nearly all following queries. The trie and structural indices grew to 775.68MB (≈ 29.8% of the attribute data) after these queries.

Number Index

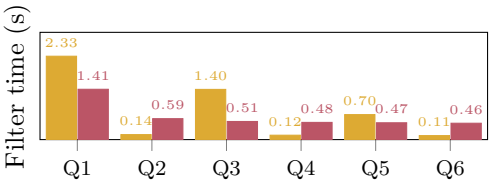
Figure 6.11 shows the performance when querying the same number attribute with and without adaptive number index. All normal tweets contain the chosen path, `/user/friends_count`, so there are next to no structural benefits. The split size for the histogram is 100, so every split creates 100 new buckets. Because the Twitter dataset is skewed to lower numbers of friends and followers, more queries are in the lower range to make it harder for the index and force histogram splits. Queries Q8 and Q7 query values outside of the existing value ranges. The results are clear: Even with very large result sets that will cause histogram splits, e.g. Q2, performance is far ahead of the alternative. Any queries for values outside the minimum and maximum are near instant. However, as number-based predicates are already performant by default, the index is only paying off after querying the same attribute three times. The number index, again together with the structural index required for management, needed 694.9MB (≈ 0.63% of total data) more memory than the default execution. This index generally requires more data, as it has to copy the whole numerical attribute data, together with a document reference. Hence, replacing or optimizing this index for memory usage should be prioritized in future work.

Mixed-Predicate Queries

In the following, we test the robustness of the adaptive approach regarding usage of multiple indices in one query and index usage combined with non-index-supported predicates.

```
Q1 LOAD TWITTER CHOOSE '/user/followers_count' >= 5000 &&
    STARTSWITH('/user/screen_name', "Fel")
Q2 LOAD TWITTER CHOOSE '/user/followers_count' >= 4000 &&
    STARTSWITH('/user/screen_name', "Ma")
Q3 LOAD TWITTER CHOOSE '/user/followers_count' >= 7000 &&
    STARTSWITH('/user/screen_name', "Mon") && '/retweeted_status/
    user/friends_count' == 500
Q4 LOAD TWITTER CHOOSE '/user/followers_count' >= 5500 &&
    STARTSWITH('/user/screen_name', "De") && '/retweeted_status/
    user/friends_count' == 600
Q5 LOAD TWITTER CHOOSE ('/user/followers_count' >= 6000 &&
    STARTSWITH('/user/screen_name', "Ko")) && ('/retweeted_status/
    user/friends_count' > 10000 || '/user/verified' == true)
Q6 LOAD TWITTER CHOOSE ('/user/followers_count' >= 6500 &&
    STARTSWITH('/user/screen_name', "Li")) && ('/retweeted_status/
    user/friends_count' > 9000 || '/user/verified' == true)
```

Listing 6.3: Queries with multiple index-supported predicates



(a) Filter time for multiple index-supported predicates

(b) Complex hybrid query filter time

Figure 6.12: Benchmark results of mixed-predicate queries

```
Q1 LOAD TWITTER CHOOSE '/user/screen_name' == "J03LP1M3NT3L" &&
    SCONTAINS('/user/screen_name', "T3L")
Q2 LOAD TWITTER CHOOSE '/user/screen_name' == "J03LP1M3NT3L" &&
    SCONTAINS('/user/screen_name', "T3L")
Q3 LOAD TWITTER CHOOSE '/user/screen_name' == "non_existing_user"
    && SCONTAINS('/user/screen_name', "T3L")
Q4 LOAD TWITTER CHOOSE '/user/screen_name' == "non_existing_user"
    || SCONTAINS('/user/screen_name', "T3L")
```

Listing 6.4: Complex queries with supported and non-supported predicates

The result of multiple queries (Listing 6.3) containing 2–4 predicates, which are all supported by indices are shown in Figure 6.12a. All predicate values of the queries differ from each other to keep the indices spending time on improving their data structure. Predicate paths stay the same to not construct a new index each time. In Q1, two indices are created, which creates a bigger discrepancy between the adaptive index and the default variant. Q3 and Q5 also create new structure index nodes together with content indices. The index-initializing queries show a expected slower filter time, while all other queries show vastly improved runtimes. In total, all created indices required 1.73GB ($\approx$ 1.59% of total data) after executing all queries.

The queries shown in Listing 6.4 mix a content-index supported operation with a non-supported predicate. As we can see in Figure 6.12b, the first query is of course slower again. But limiting the search-space due to lazy evaluation of the query predicates improves the runtime of queries with logical AND predicates. On the other hand, using a top-level OR predicate, reduces the runtime, as all documents have to be loaded and evaluated anyway. This highlights the currently biggest drawback of our approach. To reduce negative performance impacts, a query analyzer should check if using an index would even make sense for the given query. In total, all created indices required 637.9MB ($\approx$ 0.58% of total data).

6.4 Summary & Potential Extensions

In this chapter, we considered adaptively indexing semi-structured documents in ad-hoc data processing engines that aim at data exploration and wrangling tasks without heavy initial processing. We introduced a structural index that incrementally resembles the structure of a set of documents to limit the search space of future queries accessing attributes in the same sub-tree. This structural index was then augmented with two proof-of-concept content indices, which are adaptively built and improved with string and number attributes in these documents. The experimental evaluation revealed that with moderate initial investment, the runtime of forthcoming queries on the same attributes, even with different predicates, can be vastly reduced. Future improvements could be achieved by introducing a better query analyzer to prevent unfavorable index calls and complementary or alternate adaptive indices to the ones presented in this chapter. Most importantly, these indices would need a better memory footprint than the current, relatively large proof-of-concept indices. Additionally, a multi-query optimizer could be developed which analyzes a known query load to prevent the creation of unnecessary indices.

Chapter 7

User-Defined-Modules

Recent years have witnessed unprecedented competition of data management researchers and companies to provide solutions to handle the vast amount and increasing heterogeneity of data. This has spurred the development of novel data management solutions (aka. NoSQL), tailored to specific data characteristics and query capability demands and the augmentation of traditional relational database management systems to support novel features. Given the plethora of data formats, required functionality, and workloads, a one-size-fits-all solution off-the-shelf is arguably hard to reach and would likely lose grounds to tailored solutions for specific workloads. Data scientists appear reluctant to dive into the plethora of existing solutions but opt for using primitive data preparation utilities and scripted data transformation pipelines. However, many a functionality is provided solely in programming language libraries, most prominently (complex) pre-trained models and toolkit around machine learning tasks (e.g., Wolf *et al.* [79]), and difficult to integrate with existing systems without major effort. In this chapter, we show that by enabling user-defined modules, seemingly simple data processors can combine the advantages of a multitude of systems to enable new pipelines that outperform existing solutions. We implemented our approach by prototypical modularizing JODA, where core parts have been rewritten to accept Python scripts as implementations. While this adds versatility, the high-performance, multi-thread core of JODA, remains untouched. This combination offers blazing-fast parsing and processing of raw JSON data in a scalable fashion, combined with the possibility to write extensions like user-defined functions, custom indices, and support of additional data formats and I/O routines, provided in plain Python.

7.1 Core Architecture and Modules

As mentioned in Chapter 4, JODA is fully decomposed into a set of independent modules—or `tasks`. Each task is defined as a class that can be executed, given an input queue, an output queue, or both. Depending on the query, these tasks are added into a query execution `pipeline` where compatible tasks are connected via I/O queues. The `scheduler` then executes instances of each `task` using all available or a configured number of CPU cores. A `task` may limit the number of instances that can be executed in parallel. For example, a task that reads data from a file system may limit the number of instances to one. A task may stop its execution when the input queue and the task are finished, the output queue is full (stalled), or the input queue is temporarily empty (starved). Stalled and starved tasks are restarted at a later time by the scheduler. Moving from a static query planner calling hard-coded functions to a dynamic query planner based on tasks greatly improved flexibility. Depending on user queries and optimization rules, tasks can now be added, replaced, reordered, or removed.

```
LOAD FROM FILE "File.json",
     FROM SQL "postgres://localhost:..."
CHOOSE LANG('/text') == "en"
AGG (' ':TRAIN(' '));
```

Listing 7.1: Example JODA query using multiple user-defined modules

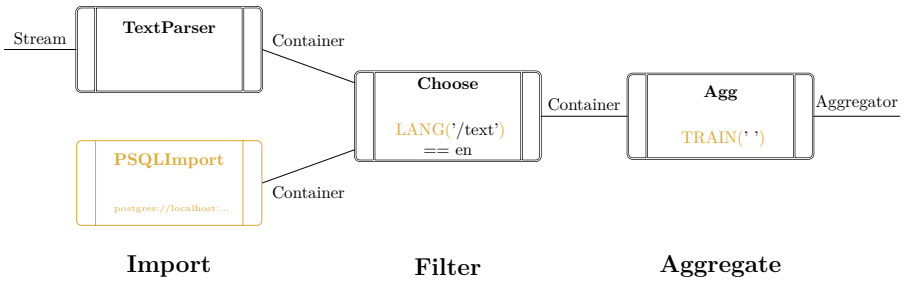


Figure 7.1: A sample JODA pipeline using user-defined modules

While the data being transmitted between tasks can be of different formats, JODA mainly uses `containers` (see Section 4.2.2) as atomic units of data. A container bundles multiple JSON documents and supplementary data structures like (cracking) indices, query caches, and data synopses. This enables an (almost) entirely lock-free execution of queries as the documents are passed through the pipeline without requiring supporting centralized data. Figure 7.1 shows the pipeline of the sample JODA query in Listing 7.1 using both a native JSON import and a customized PostgreSQL import, with subsequent language identification through an ML model via Python, multithreaded filtering in core JODA, and learning of a machine learning model.

Within the tasks that interact with specific parts of the documents, like filtering, transforming, and aggregating, we use `sources` as interfaces to the data. A `source` may be a simple pointer to a part of the document or a function with parameters that use inputs to calculate a new value. All sources in JODA are also implemented as independent classes being loaded and dynamically added to the internal query processing. They define their output type, how many parameters of which type they expect, and provide a function that calculates the result, given a list of parameters. For instance, a source calculating the length of the string defines its return type as `int` and expects one parameter of type `string`. With JODA being decomposed into independent modules that are scheduled and connected based on rules, we can easily extend the system. For instance, can a filter task now be added—and automatically connected—to every other task that returns a `container`. If we implement a new kind of importer, that retrieves `containers` from another system, the query planner can simply add the task to the pipeline and any successive filter task will automatically be connected to it.

This enables the user to add new tasks to the system, such that developers, researchers, and data scientists can add new functionality without understanding or changing the core code. Section 7.2 introduces each module type in detail and gives motivating examples showcasing their application.

7.1.1 User-Defined Modules

`Modules` are our means for the user to extend the system. A `module` is a single script file that can be loaded into JODA to provide some sort of functionality. It has to provide specific functions and variables depending on the feature it implements. To support a wide range of use cases, we implemented the following modules that a user may import:

- **Import:** Connects the processor to new data sources that may previously not be supported.
- **Export:** Exports JSON documents into a user-supplied format or system.
- **Index:** Implements a new index for improving the filter performance of JODA.
- **Agg:** Supplies a new aggregation function that uses a set of data to compute a single result.
- **Source:** A typical user-defined-function as known from other database systems. It provides a single JSON value, given one or multiple parameters.

To support user-supplied functionality, we use a `ModuleRegistry`, and depending on the requirements, a set of `Module`s. When a user supplies a new module, the registry loads the given script and infers the type of the module by the given functions. From then on, the module can be used in all subsequent queries until the user deregisters the module. The registry stores a mapping of module names to the script location such that the query pipeline can find and invoke it. Internally, each module initializes a `task` or `source` class which is also stored in the registry.

During query parsing, the registry is accessed to find the user-supplied module if an aggregation or a source function is referenced but not found in JODA's native function list. An error is raised if the registry is also unable to find the referenced function. Otherwise, the source created from the module is embedded into the internal query representation. During query execution, the module is then invoked to compute the result. Similarly, if the query contains an unknown import or export statement, the registry is checked for an import or export task to be added to the query pipeline. Indices form a special case, as they are not directly referenced in queries but chosen by the query planner. All internal and user-supplied indices are collected during query planning, and the planner then chooses the best index for the query using an estimator given by the index. Given a query, the internal index data, and a `container`, the index estimates how many documents have to be reevaluated with the actual predicate if it is executed.

7.1.2 Connecting Scripts and System

A major aspect of executing user-supplied code is to enable efficient and versatile communication between scripts and the core system itself. In JODA, data is stored internally as a dynamic in-memory JSON data structure. As most languages have some kind of support for JSON data or at least have third-party libraries that provide such support, it would be possible to translate the

internal structure back into a JSON string representation and pass it to the user-supplied scripts. Then a language-specific function can parse and interpret the JSON data. However, translating and parsing such a document would cause significant overhead, in particular, if the query deals with large amounts of documents. Hence, it is preferable to immediately translate the internal data structure into a language-specific one. Before a user function is executed, the system will initialize a variable in the chosen language environment and call the translation function of the implemented language.

As we chose Python as our first supported language in our system, we decided to map JSON values into their Python-native counterpart by traversing the JSON document depth-first. Basic datatypes like `integer`, `float`, `Boolean`, `null`, and `string` are directly initialized in the Python environment. The composite `array` can also be directly converted as JSON as well, as Python supports arrays with heterogeneous datatypes. Finally, every JSON `object` is converted into a Python `dict`. The functions in the user-supplied script are then called with the translated variables as parameters. The potential return values then have to be translated back into the JSON format using the inverse mapping of the previous step. Most language APIs return a handle to the internal result, which is again translated by the system using the language-specific `translate` function. Afterward, the language-specific result is uninitialized, and the translated value is passed on to the internal query engine. If the called function returns an unsupported data type, a runtime error is thrown, and the user function is assumed to have returned `null`.

7.2 Sample Use Cases

We next outline the novel capabilities of JODA through specific use cases, such as allowing interoperability between systems, enabling the improvement of the execution through external approaches, and outlining the perks and benefits of allowing user-defined functions in a system specifically tailored for JSON data. To enable the inclusion of these novel capabilities, we follow two notions:

Ease-of-use: although we enable the users to provide complex functionality in JODA, they just need to provide scripts following straightforward templates to realize it. The internal logic is completely hidden and they do not need to understand it.

Efficiency: the known benefits of JODA are directly applicable since the novel features will be realized on the data that resides within our JSON processor. With that, the users will benefit from fast data processing without transporting the data to any other format or system for performing complex analysis.

Through these capabilities of JODA, we not only outline the guidelines for a more interoperable system but also show that the benefits are double-sided, as both the system and the extending approaches can benefit from the following scenarios.

```
# Returns a single value based on the given parameters
def get_value(arg1, arg2, ...):
  return arg1 + arg2
```

Listing 7.2: Example of a user-defined source function

```
# Initializes the state of the aggregator
# It is passed to every other function
def init_state():
  return 0
# Aggregates a single value into the state
def aggregate(state, arg1):
  return state+arg1
# Merges two states into one
def merge(state, other):
  return state+other
# Finalizes the state into a result
def finalize(state):
  return state
```

Listing 7.3: Example of a user-defined aggregation function

7.2.1 User-Defined Functions

Nowadays, the research in the field of databases constantly introduces novel purposeful structures and algorithms for various new applications and produces improvements over already established aspects of the databases. However, contrary to this research, systems are often expected to have a prespecified set of simple capabilities which are in most cases sufficient for simple operations. Therefore, for requirements that fall outside of the capabilities envisioned for the system, proper modifications of the internal implementation of the system are needed. Thus, to avoid the direct change of the system, often, additional services and applications are required, which rely on the need to extract the data from the data system and process it outside of it, contradicting the initial purpose of the system itself.

The user can extend JODA with a user-defined function that can be called during query execution. To supply the extension, a script following a predefined template needs to be provided. To perform the extension the user only needs to perform a simple API call and provide the implementation of the desired algorithm. This way, the core system will stay entirely hidden and knowledge of them is not required to benefit from the fast processing and We distinguish between two different types of extensions, source functions and aggregation functions. For source functions, the user only needs to implement the method `get_value`, as shown in Listing 7.2. To provide a new aggregation function, the user has to implement the functions `init_state`, `aggregate`, `merge`, and `finalize`, as shown in Listing 7.3. The function `init_state` will be used for initializations needed by the new feature. The function `aggregate` will be executed for every passed JSON document. The merging of the states—that may have been calculated in different threads—will be performed in the `merge` method, and the final result over the merged data will be computed in `finalize`. In the following, we consider example extensions for JODA and detail their realization of the necessary methods.

```
model = fasttext.load_model(PRETRAINED_MODEL_PATH)
def get_value(arg1):
    prediction = model.predict(text)
    return pred.split('__label__')[1]
```

Listing 7.4: Language Identification in Python

Example Extensions

To extend the functionalities of JODA, we implemented **language detection**, **sentiment analysis**, **training and evaluation of a learned model**, and **computation of number statistics** extensions in Python.

To motivate the need for the per source extensions language detection and sentiment analysis, consider a collection of tweets for which the language of origin and the sentiment are unknown but are of interest to the user. The user is interested in identifying all tweets concerning the sports company Adidas that were written in the German language and have a positive sentiment. Such an extension to an already existing system would require modifying the internal code of the system directly to enable this capability. This will naturally be cumbersome for the user since it will require time and the ability of the user to understand the system code. Certain functions and approaches are already quite common and prominent in specific languages, where often, more advanced and ready-to-use libraries exist. By using them, one can avoid recreating the same functionality in the native language of the used system. Examples include the machine learning libraries, such as the language detector or the sentiment analyzer, for which several pre-trained models already exist in Python and can be directly used to extend JODA. Enabling the loading of such a model in JODA will produce the required results will minimal effort.

An example script for extending JODA with language classification by utilizing the *fastText* [80] language classifier is shown in Listing 7.4. Once the language classification is registered, it can be used as a function directly in the query language:

```
LOAD Twitter AS ('/t':'/text'),('/lang':LANG('/t'));
```

As one example of an aggregation extension, we implemented the computation of simple numerical statistics (average, sum, and count) over a given attribute in Python. The script is presented in Listing 7.5. As explained, the `aggregate` method will be performed for each document individually and the `merge` method will connect the temporary results from different partitions. The method `finalize` produces the required result, in this case, an object containing the average, sum, and count.

Adding new capabilities to JODA can also simplify the handling and improve performance of existing code. For example, for training a machine-learning model where the data is too large to be stored in memory, the user will first need to load the data by taking only the relevant parts for the approach and performing the required pre-processing. However, JODA already provides an efficient mechanism to load and process data in batches that can be used, to

```
def aggregate(state, num):
  return [state[0]+num, state[1]+1]
def merge(state, other):
  return [state[0]+other[0], state[1]+other[1]]
def finalize(state):
  return (state[0] / state[1], state[0], state[1])
def init_state():
  return [0, 0]
```

Listing 7.5: Statistics computation in Python

train and use a learned model. In our case, a **Stochastic Gradient Descent (SGD) Classifier**. To train the model, we need to follow the template for an extension working over a collection of records. In the `init_state` method, the classifier model is initialized if it has not been already created. If a model exists, it is loaded for the current JSON partition. The input records are added to the training batch in the `aggregate` method. If the training batch reaches the required predefined size, the training of the model is invoked. The `merge` method is crucial since it can happen that in the previously covered JSON partition, there were remaining documents that were not yet used for training. These documents will be added to the current data batch and used for training once the training batch size is reached. In the `finalize` method, once the remaining documents are used for training, the model will be saved. Once the model is trained, to use it, we need to realize the template for the extension over single records. Hence, in the `get_value` method, we use the trained model for predicting over the input records. Native JODA query language features have been used to filter, clean, and scale the dataset before training the model.

Once the training and evaluation of the model are registered in JODA, they can be called in the same way as the extensions. The time and memory statistics can be obtained directly from the output when calling the new functionalities. For the analysis of the prediction accuracy, the user needs to register one more extension that compares the predicted and actual class for every record.

7.2.2 Replacing and Augmenting Data Processing Modules

Besides the aspect of applying novelty to an existing system through extension, frequently there is a need for improvement by employing novel approaches as a replacement for existing ones. With the research prosperity, new approaches are constantly developed, which produce better results when considering memory footprints or execution time. However, Existing systems rarely progress at the same rate as the most recent scientific advancements and stick to existing techniques. Next to user-defined functions, which are commonly supported in relational database systems, JODA also allows to replace data structures and query processing behavior. Consider the case of replacing existing membership index structures with faster and more efficient ones. Often this would warrant modification of the system code, specifically where such structures are tightly bound with the system. In JODA, one has to provide a replacement script where the implementation logic will be realized through five methods: The

```
def estimate_usage(predicate, state):
    if predicate in state:
        return 0
    return None
def execute_state(predicate, state):
  return (state[predicate], True)
def execute_docs(predicate, docs, state):
  return None
def improve_index(predicate, state, doc_index):
    state[predicate] = doc_index
def init_index():
    return dict()
```

Listing 7.6: Query cache in Python

`init_index` method initializes the index with a persistent state. The method `estimate_usage` gives an estimate on the remaining work after a predicate from the query has been evaluated on the index. Then, for each container the `execute_state` function is called, which may return a filtered document set, using only the state of the index. If this is not possible, and the actual document contents are needed, `None` can be returned and the system will call the `execute_docs` method, which determines which of the given documents fulfill the predicate. Finally, the method `improve_index` updates the index with the final results of the query if they are suitable for the index. As a proof-of-concept for replacements in JODA, we considered and realized a *Bloom filter* and a *query cache*. Both replacements were realized in Python.

The module for replacing an existing index in JODA with a query cache is depicted in Listing 7.6. Once the query cache is registered in JODA, the system will check the output of the `estimate_usage` method to determine the best index for the query. Thus, when the predicates are in the query cache, the results from the replacement index will be used.

This feature can also be used to implement domain-specific indices that are too specific to be implemented in a general-purpose data processor. For example, if the user extensively works on geospatial data, one can implement a spatial index that is optimized for the data set.

7.2.3 Customized Data Import and Export

Consider the case where relevant data is distributed over multiple files or systems, e.g., a company employs a relational database of customers and sales data, a key-value store for online shopping carts, and local CSV files of access logs. Joint data processing over such data sources is a tedious task and would typically require human-involved pre-processing, before the system of choice is able to execute the actual query.

A solution to this problem can be wishful thinking—to hope that the developers of the given tool add support for the required data soon. A more realistic and versatile solution is to have the system provide simple means to extend the data import and export mechanisms of the system, developers can ensure that their tool can be used for highly specialized tasks that integrate many different

```
  # Initializes a state, given a parameter string
  def init(param):
    return <State>
  # Returns the next entry from the input, given the state. If no
      more data is available, returns None.
  def get_next(state):
    to_transform = state.import_next(data)
    return transform(to_transform)
```

Listing 7.7: Data import module interface

```
  # Initializes a state, given a parameter string
  def init(param):
    return <State>
  # Exports a single entry, given the state and the internal data
      representation
  def set_next(state, data):
    to_export = transform(data)
    export(to_export)
  # Is called after the export of all entries is finished to
      perform cleanup tasks
def finalize(state):
  pass
```

Listing 7.8: Data export module interface

sources. Indeed, for many use cases, it is enough to let the user specify how the data has to be transformed, read, and written. Listings 7.7 and 7.8 show our suggestions for simple interfaces supporting user-given import and export tasks. Both modules work with an internal state that can optionally represent a (stateful) resource to be used, like a file handle or an open database connection. Using this state, the import module fetches new entries from the source one-by-one, while the export module transforms the JSON documents into their representation to be stored.

By supporting these interfaces, users can quickly connect their data format or system to the supporting data processor. As proof-of-concept connections, we have implemented a CSV file and a PostgreSQL table reader and writer. Listing 7.9 shows a straightforward example of adding support for reading CSV files into our JSON data processor.

```
import csv
def init(param):
  csvfile = open(param, newline='')
  reader = csv.DictReader(csvfile)
  return reader
def get_next(state):
  return next(state, None)
```

Listing 7.9: Example CSV import implementation

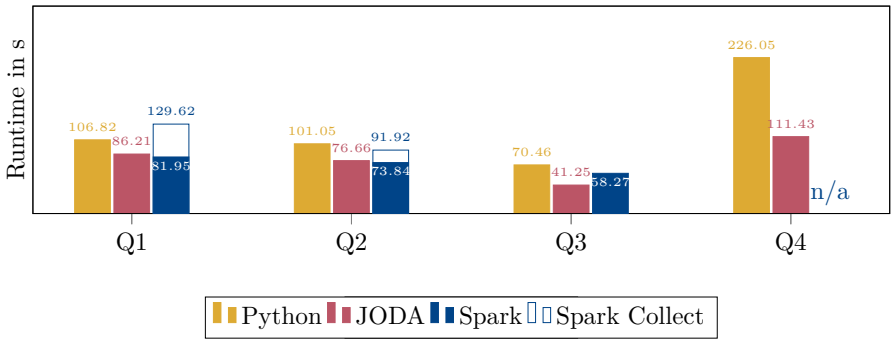


Figure 7.2: Runtime comparison of values

7.3 Evaluation

To provide evidence that the extensions above are not only viable but can also bring large runtime benefits, we implemented at least one user-defined module for each use case. All experiments have been run on an Arch Linux OS using an i7-855U 8x1.8Ghz CPU with 16Gb of RAM. As a dataset, a 5Gb excerpt of the raw Twitter JSON stream[1] has been used. Every evaluation has been run 10 times and the average runtime was computed. The runtime of each query has been compared to the runtime of producing the same result with native Python code. If applicable, we also included the module into Spark and compared the results.

As discussed in Section 7.2.1, we implemented `value modules` performing sentiment analysis, language interference, and training of a custom machine learning model. Additionally, we have implemented a straightforward `aggregation` module that given a numerical attribute, calculates the count, sum, and average values, and returns them in a textual format. Queries Q1, Q2, and Q3 in Listing 7.10 show the usage of these modules in a JODA query.

For the sentiment analysis, we parsed the Twitter dataset and selected only tweets written in English. We then used the user-defined sentiment analysis to return the tweet ID together with the sentiment description for each document. The native Python implementation reads the document set line-by-line and parses the JSON into a native dictionary. It then performs the same operations as JODA. Using the same module, we also implemented a PySpark script, which registers the Python module as a UDF, reads the JSON documents into a data frame, filters it, and applies the UDF to each row. For all systems, the results are written back to a single JSON file. As we can see in Figure 7.2, JODA returns the results the fastest, followed by the native Python implementation. As Python uses a global interpreter lock (GIL), it can only use a single CPU core to process the data, while JODA and Spark distribute their work on multiple cores. But the GIL also restricts JODA to only being able to execute one thread of user-defined Python code simultaneously. This is why, despite working on 8 cores, JODA is only 1.24 times faster than Python. The performance of Spark in this evaluation is slightly better than JODA, if the results are returned per partition.

[1] `https://developer.twitter.com/en/docs/labs/sampled-stream`

```
//Q1 - Sentiment analysis
LOAD FROM FILES "/data/twitter"
CHOOSE EXISTS('/text') && '/lang' == "en"
AS ('/sentiment':SENTIMENT('/text')), ('/id':'/id')
STORE AS FILE "out.json"

//Q2 - Language detection
LOAD FROM FILES "/data/twitter" CHOOSE EXISTS('/text')
AS ('/text':'/text'), ('/lang': LANGUAGE(
    REGEX_REPLACE('/text',"\\n"," ")))
STORE AS FILE "out.json"

//Q3 - Number Statistics
LOAD FROM FILES "/data"
CHOOSE ISNUMBER('/user/followers_count')
AGG ('':STATS('/user/followers_count'))

//Q4 - SQL export
LOAD FROM FILES "/data/twitter" CHOOSE EXISTS('/user')
AS ('/id':'/user/id'),
('/name':REGEX_REPLACE('/user/screen_name',"\\x00",")),
('/location':REGEX_REPLACE('/user/location',"\\x00",""))
('/followers':'/user/followers_count'),
('/friends':'/user/friends_count')
STORE AS SQL_EXPORT "postgresql://postgres:postgres@localhost
    :5455/postgres twitter_users"
```

Listing 7.10: JODA queries using user-defined modules

If a single result document is expected, Spark has the worst performance, due to coalescing the results.

In Q2, we again parsed the Twitter dataset and used the Python third-party library FastText [80] to infer the language of the tweets using a machine learning model. This time, the text had to be preprocessed by removing all newlines with spaces for the model to work properly. This preprocessing is done in each system natively, while the module only performs the language interference on the text. Again a native Python script reads all documents line-by-line and applies the module to each tweet, while PySpark registers the module as a UDF. The results are written back to a single JSON file. The runtimes are similar to the previous experiment. Only the cost of coalescing the data has less of an impact on the Spark implementation.

Q3 evaluates the performance of a straightforward `aggregation` module. The Python implementation again uses the module functions on a line-by-line parsing of the JSON data set, while JODA registers the module and uses its aggregation framework. As PySpark does not support user-defined aggregation functions for data frames in Python, we adapted the Spark script to read the JSON files into an RDD and use its aggregation function with the module. Before aggregating, each system natively checks if the nested attribute exists and has a numerical type. The aggregation result is then returned to the user. As we can see in Figure 7.2, JODA returns the results the fastest, followed by Spark, and with the native Python implementation the slowest.

In the last experiment, we wanted to test the performance of a custom data export module. For this experiment, we implemented a module that connects to a PostgreSQL database and exports the given dictionary/JSON document set into a table. Q4 first checks for the existence of a user attribute and then transforms the document into a simplified representation of the user. This reduced user object is then stored in a PostgreSQL table. The user-defined module uses the Python `psycopg2` library to connect to the database and execute the query. The native Python implementation again parses the documents, transforms them, and finally calls the module to insert each tuple into the database. While Spark natively supports the export to databases, it does not support user-defined data readers or writers. In this section, we want to evaluate the performance of user-supplied modules. Hence we omitted the Spark implementation. But the evaluation shows that the multi-threaded nature of JODA allows for much faster filtering and transformation of the data, such that JODA performs the task in 49.2% of the native Python implementation.

We also briefly evaluated the potential performance benefits of using a user-defined `index` module. JODA implements a straightforward predicate cache index, in which the result of a filter step is stored in a hash map, with the predicate as the key. For this experiment, we implemented a naive version of such an index in Python, which stores the result of a filter step in a dictionary. To evaluate this index we again parsed the Twitter dataset and executed a query with a relatively costly filter predicate using regex matching on the text attribute. Then we executed the same query again, but this time the query-planner will use the index to evaluate the index step. This procedure has been repeated once for JODA configured to (a) not use any index at all, (b) use the native predicate cache, and (c) use the user-defined Python index. For configuration (a), the first query took $0.35s$ while the successive query took $0.23s$ for a total of $0.58s$. Configuration (b) took $0.4s$ and $0.01s$ for a total of $0.41s$. As we can see, filling the cache incurs some minor overhead, but the subsequent queries are much faster. Finally, configuration (c) took $0.36s$ and $0.09s$ for a total of $0.45s$. The filling of the index has less of an impact on the user-defined implementation, as the implemented index is far simpler than the native one. Furthermore, is the second query much slower than the native index, but still requires less than 1/3 of the time than without the index. This illustrates that it is possible to implement user-defined indices that can be used by the query planner to speed up the execution of queries.

In **summary**, we found that integrating user-defined Python modules into JODA is a very powerful tool for data analysis. The performance, while being restricted by the GIL, is still significantly better than the native Python implementation. But there is still room for improvement, as a distributed Spark job can outperform JODA with Python modules in some tasks.

7.4 Summary & Potential Extensions

In this chapter, we described how a modularized data processor can be easily extended to support user-defined external modules. We implemented the described changes in the JODA data processor as a proof of concept. For every supported module, we provided use cases and examples. We further showed

that the performance of this system can be better than writing specialized code in the programming language of choice or using a general data processor with programming support. By supplying modules that integrate with external tools we also showed that we can fit the system into a broader ecosystem of tools and hence, create a powerful data processing pipeline.

While the described changes are already usable and performant, there are still some improvements that can be made. Because of the Global Interpreter Lock of Python, only one thread can execute Python code at a time. In an inherently multithreaded system like JODA, this can be a bottleneck for complex user-defined functions. To overcome this, JODA could spawn multiple Python worker processes and distribute the workload among them instead of using a single Python interpreter. This would make the implementation more complex, but would also allow the user to use the full power of the CPU cores.

Another improvement would be to support more programming languages. Currently, JODA only supports Python, but it would be possible to add support for other languages. Interpreted languages are the easiest to implement, as they can be executed in the same process as the JODA core by a simple library call. For example, Ruby, Lua, or JavaScript could be supported in this way.

By implementing a more complex compiling and dynamic linking architecture, it would be possible to support compiled languages like C or C++. This change could speed up the execution of the user-defined functions and would enable support for more languages.

While currently the most important party of JODA have been modularized and can be extended, there are still some parts missing. For example, could JODA support user-defined tasks, that supply their own query statements to be used in the JODA query language. The tasks would then be integrated into the execution pipeline and could perform arbitrary computations on the data.

Chapter 8

BETZE: A Novel Benchmark for Interactive Exploration

Interactive data exploration [65–67,81–83] 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 [84–87].

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 she can obtain the tweets by demanding the existence of an attribute 'user' (cf. Figure 8.1), 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 chapter, we present our benchmark generator BETZE [5], which can be used 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 [65] specifically addresses research challenges around assisting users on their way to find hidden insights through multiple, iterative steps, often via visual support [88–90], 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 the idle time of users and to keep them focused [91, 92].

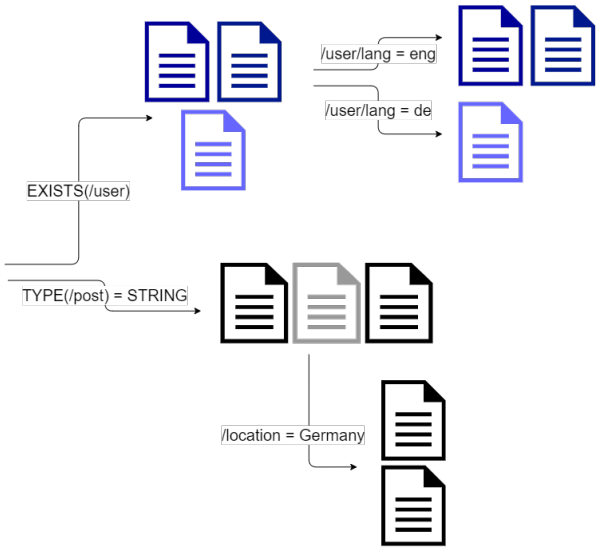


Figure 8.1: Data exploration example

Existing benchmarks for semi-structured documents, like NoBench [61], 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 [83]. 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 [93], 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 our JODA systems to extract the insights using the `ATTSTAT` aggregation function.

In this chapter, 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.
- We present the results of a first experimental evaluation using the proposed benchmark and four competitors.

The remainder of this chapter is organized as follows. Section 8.1 introduces the random explorer model and which types of queries are supported as of now. Section 8.2 discusses how query generation is conducted. Section 8.3 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 8.4 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 8.5 summarizes the chapter and discusses potential extensions.

8.1 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 [93]. 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 8.2 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 8.2)

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

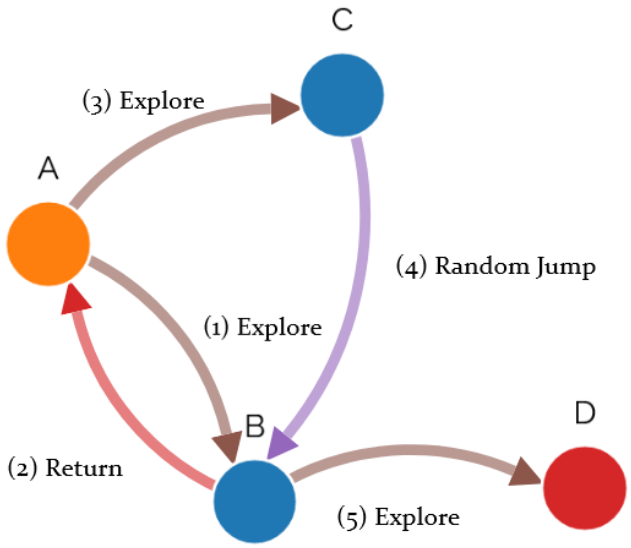


Figure 8.2: Possible exploration session in the random explorer model

	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 8.1: Default user configurations

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 8.1, that cause heavy, intermediate, and low load according to the aforementioned rationales—as the experiments confirm, e.g. in Figure 8.5. 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 [84–87].

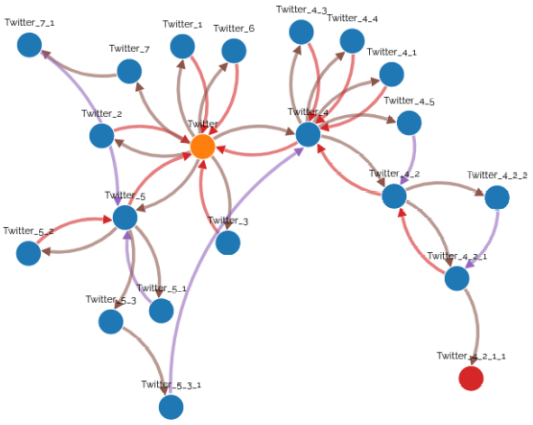
- A **novice user** does not have any knowledge about the tools and no intuition about the dataset and how to achieve their goal. Such a user will issue more queries that backtrack and often jump to another random dataset when noticing that the current path probably will not yield the desired result.
- 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 8.3 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).

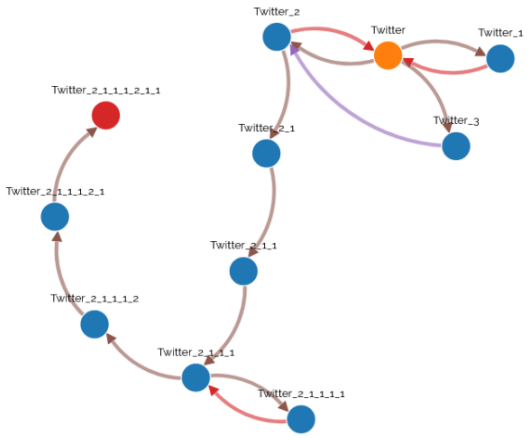
8.1.1 Query Support

Our initial implementation of BETZE is able to generate queries for JODA [2], MongoDB [12], jq [9], and PostgreSQL [10]. As mentioned before, support for additional systems can be added easily with a few lines of code. We give more details on how this can be done in Section 8.2.4. 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:

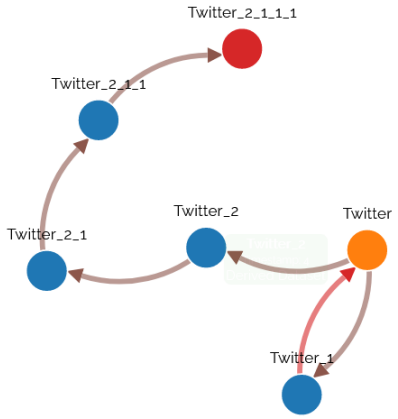
- Loading datasets
- Filtering datasets
  a) Using simple predicates
  b) Combined only with logical binary **AND** and **OR** operators
- Outputting:
  a) the whole content of the selected documents
  b) an aggregation of the documents (grouped by an attribute)



(a) Example novice user session



(b) Example intermediate user session



(c) Example expert user session

Figure 8.3: Example user sessions

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 8.1 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.

8.2 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.

8.2.1 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 8.2, 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

```
# 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 8.1: Example queries for each of the supported languages

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.

8.2.2 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 8.1.1) are checked for their validity to be applied to this path. If no predicate is applicable to the given

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

Listing 8.2: Example analysis file

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 8.2 and 8.3). 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.

8.2.3 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 8.1 and shown in Figure 8.2, 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 8.2 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 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 parent's 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.

8.2.4 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 8.3 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, prefix matching) as leaf nodes. For each implemented language interface, the benchmark generator will translate the internal query representations into a system-specific script using the `Translate` function. Listings 8.4 and 8.5 show excerpts of the PostgreSQL and MongoDB implementations of the language interface. They illustrate the ease of implementation, as the translation of the internal query representation into the system-specific query language is mostly a matter of string concatenation.

```
type Language interface {
  // Display the 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 a preface to
      the system-specific file
  Header() string
  // Returns the delimiting symbol/string to terminate a query
  QueryDelimiter() string
}
```

Listing 8.3: Language interface

```
  func (Postgres) Name() string { return "Postgres" }
  func (Postgres) ShortName() string {r eturn "postgres" }
  func (Postgres) Comment(comment string) string{ return "-- " +
      comment }
  func (Postgres) Header() string { return "" }
  func (Postgres) QueryDelimiter() string { return ";" }
  func (Postgres) Translate(query query.Query) string{
    ...
    query_string += "WHERE " + translate_predicated(query.
        predicate)
    ...
  }
  func translate_predicate(predicate query.Predicate) string{
    ...
    switch pred := predicate.(type) {
    ...
    return fmt.Sprintf("jsonb_path_exists(doc,'%s ? (@ == %s)')",
         pred.Path, pred.Str)
    }
  }
```

Listing 8.4: Excerp of Postgres implementation

```
func (MongoDB) Name() string { return "MongoDB"}
func (MongoDB) ShortName() string { return "mongo" }
func (MongoDB) Comment(comment string) string{ return "// " +
    comment }
func (MongoDB) Header() string { return ""}
func (MongoDB) QueryDelimiter() string { return ";" }
func (MongoDB) Translate(query query.Query) string{
  ...
  query_string = fmt.Sprintf("%s.count(%s)", query.dataset,
      translate_predicate(query.predicate))
  ...
}
...
func translate_predicate(predicate query.Predicate) string{
  ...
  switch pred := predicate.(type) {
  ...
  return fmt.Sprintf("{\"%s\" : \"%s\"}", pred.Path, pred.Str)
  }
  ...
}
```

Listing 8.5: Excerp of MongoDB implementation

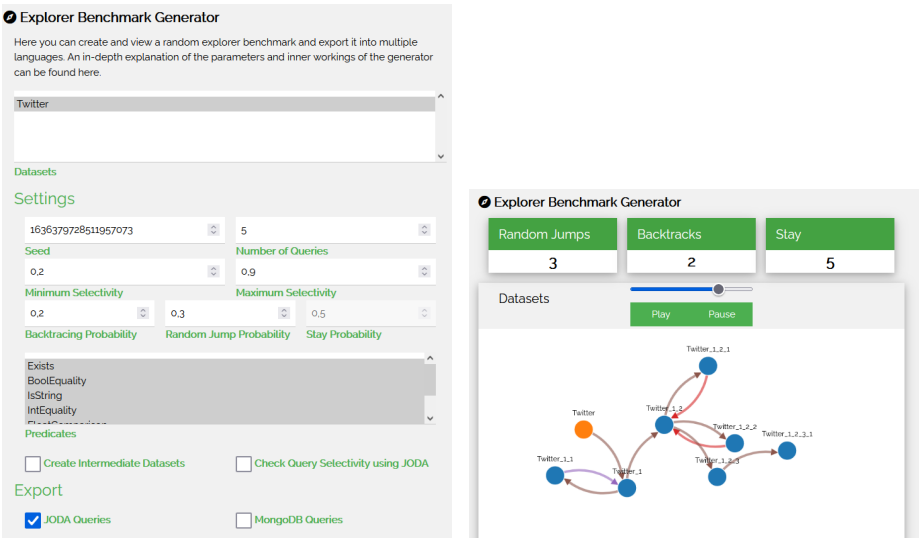
It is also possible to easily implement new predicates and aggregation functions in the system. For each function, one `Factory` class has to be implemented with two functions. First, given a specific path of the analyzed dataset, the factory has to decide whether the function can be generated for the 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 the desired selectivity.

In Section 8.2.1 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.

```
# ./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 8.6: CLI commands to generate a session and benchmark it with all supported systems.



(a) Configuration page (b) Interactive result page

Figure 8.4: The web interface of BETZE

8.3 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. We also implemented a simple command-line interface (CLI) wrapper around the library. With this, all settings introduced in this chapter 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 8.6. 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 8.4a 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 8.4b, 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[1].

8.4 Evaluation

To evaluate our approach, we employ **Twitter** dataset, consisting of a 109 GB file containing a sample of the raw Twitter JSON stream[2]. 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.

All experiments are executed in a dockerized environment to facilitate the reproducibility of the experiments. 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 also measure the query execution time using the capabilities of the systems themselves. The sum of the query execution times without data import is noted as **w/o import**. If

[1] `https://youtu.be/U0rJNEP78vY`
[2] `https://developer.twitter.com/en/docs/labs/sampled-stream`

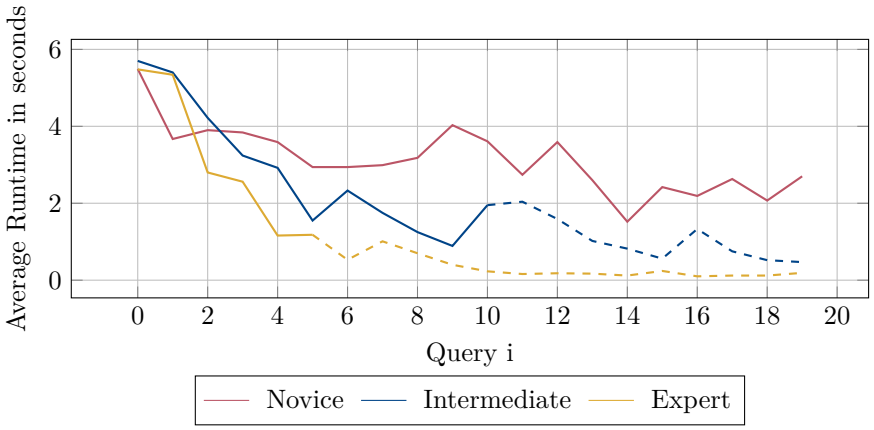


Figure 8.5: Trends in execution time for each user preset (20 queries for all users)

not noted otherwise we use the **w/o import** time by default as we are more interested in the systems capabilities as in the data I/O.

8.4.1 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 8.5 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 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 novice, intermediate user, and expert user, 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 aver-

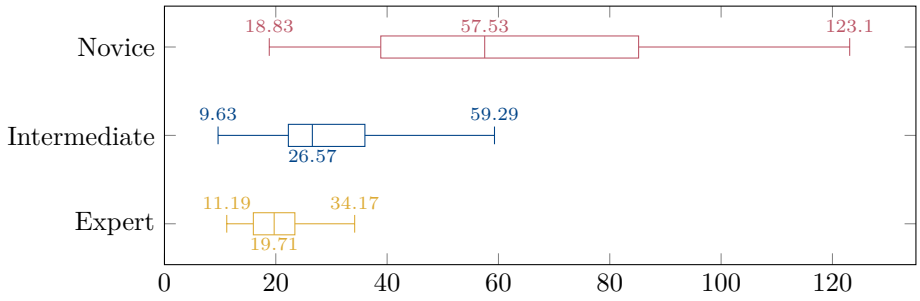


Figure 8.6: Execution time (in seconds) of 30 sessions per user configuration

age, 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 8.6 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 8.5. 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 total execution time of the entire session. Figure 8.7 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

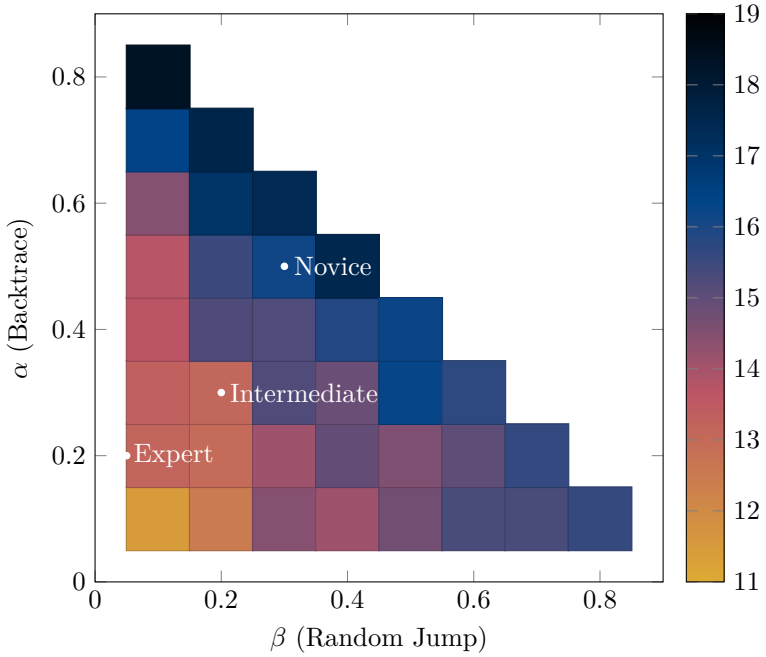


Figure 8.7: Aggregated execution times (in seconds) of $n = 10$ queries with varying jump and backtrace probabilities

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.

8.4.2 Query Skew

Queries of real users will be skewed towards the most interesting attributes. In Section 8.2.3 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 8.2 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.

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

8.5 Summary

In this chapter, 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.

8.5.1 Possible Extensions

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 thesis. 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.

Chapter 9

Evaluation

In this chapter, we evaluate the performance of JODA against the chosen competitor systems introduced in Chapter 2.3. In contrast to the previous evaluations, where we focused on one specific aspect or feature of JODA, we now evaluate the overall performance of JODA and its competitors. We evaluate each system using its default out-of-the-box configuration.

9.1 Setup

All of the following experiments are performed on a machine with 4 Xeon E7-4830 CPUs, each having 12 cores—and 24 threads—with 2.1 GHz, and 33 RAM-Kits, each having 32 GB of memory at 2400 MHz, providing the server with around 1 TB of RAM. The data is stored initially on a RAM disk to reduce the impact of the I/O on the performance. Ubuntu 16.04.3 LTS is used as the underlying operating system.

Every system is installed as a Docker container. If available, the official Docker images are used. If not stated otherwise, the default configuration of the system and Docker is used. For scaling experiments, Docker functionality is used to restrict the number of CPU cores and memory available to the container.

For server systems like Spark and MongoDB, the server is first started. Then the timing of the query starts after the server is ready and the first query is sent to the server. Timeouts are realized by externally measuring the wallclock time and killing the container.

9.2 Datasets

In this evaluation, we use three datasets with different characteristics to evaluate the performance of the systems. Each dataset is structured as a single line-delimited JSON file. The file has a size of 10 GB in its uncompressed textual form.

The **Twitter** dataset consists of a sample of the raw Twitter JSON stream[1]. The file contains 2 700 000 JSON documents, where each document has between 7 and 348 attributes, containing every JSON type. The documents are split into two major groups. Around 80% of documents are normal tweets, retweets, and replies, while around 20% of documents are deletion instructions. The tweets have a varying number of attributes, depending on their status, e.g. retweets and favorites, while the deletion documents consist of seven attributes.

[1] `https://developer.twitter.com/en/docs/labs/sampled-stream`

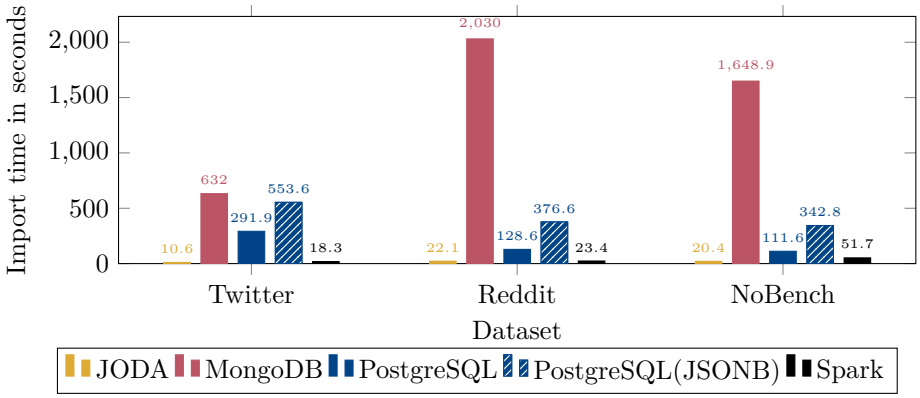


Figure 9.1: Runtime of import step in seconds

Additionally, we use **Reddit**[2] as another real-world dataset. It consists of comments on the social media platform. The 10 GB file contains 18 100 000 documents. Each document has a fixed schema with 20 attributes and no nesting.

Finally, the **NoBench** [61] dataset is a synthetic sparse dataset. Each of the 16 700 000 documents has exactly 21 attributes of all JSON types—except for null—with only minor nesting. Eleven attributes are always present, while the remaining ten attributes are randomly chosen from a set of 1 000 sparse attributes.

9.3 General Performance

As a first performance benchmark, we compare the general performance of the systems. To this end, we compare the import, filtering, and aggregation performance using simple queries.

9.3.1 Data Import

The first step for most of the systems is to import the data. Given the location of a JSON file, the systems import the data into their internal data structures. Only jq does not require an import step, as it operates directly on the JSON file. In this experiment, we measure the time it takes to import different datasets into the systems. But for all future experiments, we excluded the import time from the query time, as it is not part of the query execution and can be done once for a dataset.

To measure the import time, we use the following queries. For JODA a single import query in the form of `LOAD X FROM FILE "..." LINESEPARATED` is used. This query parses the whole dataset and stores it in-memory using an internal representation. For Spark we use the `spark.read.json` function to read the dataset into a `DataFrame`. In PostgreSQL we first create an unlogged table with the `json` column type and then import the data using the `COPY` command. MongoDB uses the `mongoimport` command to import the data into a collection.

[2] `https://files.pushshift.io/reddit/comments/`

Figure 9.1 shows the import times for the different datasets in all systems except jq. As we can see JODA is consistently the fastest system, with Spark following closely behind. PostgreSQL with the `json` column type is significantly slower than either JODA or Spark. With the JSONB column type, this difference becomes even more pronounced, as the data is additionally decomposed into the binary format. The slowest system is MongoDB. For the Twitter dataset, the difference between MongoDB and PostgreSQL with the JSONB format is not significant. But for the other datasets, MongoDB is up to 5 times slower.

While all three datasets have the same size on disk, the Twitter dataset is significantly smaller than the other two datasets when counting the number of documents. For most systems this difference results in approximately the twice the import time. But MongoDB is significantly slower, which suggests a high dependency on the number of documents for its runtime. Interestingly, as only system, the import time of PostgreSQL reduces for both column types for the larger datasets. While Reddit is a dataset with fixed schema, and NoBench has a sparse schema with low nesting, Twitter is a deeply nested dataset with a dynamic schema. PostgreSQL seems to struggle with this kind of data.

Generally, the evaluated systems can be clearly categorized into two groups. On one hand, there is JODA and Spark which are data processors without persistent database capabilities. On the other hand, there is PostgreSQL and MongoDB which are full-fledged databases. The performance of both groups is very different, as the database systems are optimized for different use cases.

9.3.2 Filter & Export

After evaluating the import performance, we evaluate basic filter and export queries. We create a straightforward filter query for each dataset and let the system export the result back to a JSON file. For the Twitter dataset, we keep only tweets with an even ID. We do the same for the Reddit and NoBench datasets, but instead of using an ID, we use the `retrieved_on` field and `num` fields, respectively. The query results are written back to a file on a mounted RAM disk to reduce the impact of disk I/O on the results. The RAM disk is directly mounted into the docker container. For JODA, we use the `STORE AS FILE` command to write the result to a file, while jq pipes the result to a file. We use the `mongoexport` command with the `-q` flag to export the result of the MongoDB query. For Spark, we use the `write.json` function to write the result. In this case, the result is not written to a single file but to a directory containing multiple files. For PostgreSQL, we use the `COPY` command.

Figure 9.2 shows the runtime of the filter and export queries for all systems, excluding the time required to import the data. As we can see, JODA is the fastest system, closely followed by Spark. Then follows PostgreSQL with the JSON datatype, which is significantly slower than JODA. With the JSONB datatype, the query needs even more time. Then follow MongoDB and jq, which are significantly slower than the other systems. But comparing the results jq with the other systems is not fair, as jq always includes the import time in its runtime. Therefore, we also include the import time of the other systems in Figure 9.3. With import included, MongoDB is by far the slowest of all evaluated systems. The execution time of PostgreSQL with the JSONB column

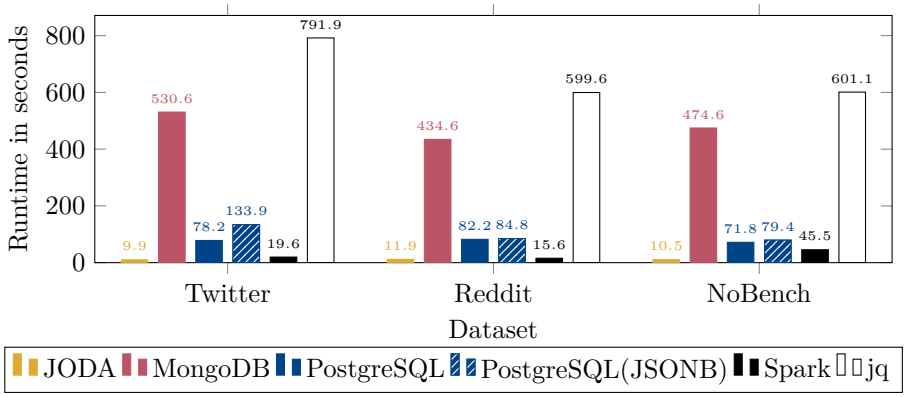


Figure 9.2: Runtime of filter and export queries (excluding import) in seconds

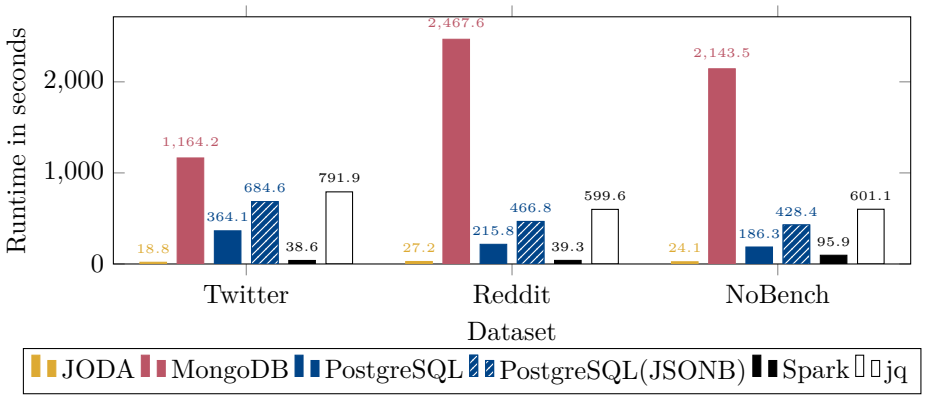


Figure 9.3: Runtime of filter and export queries (including import) in seconds

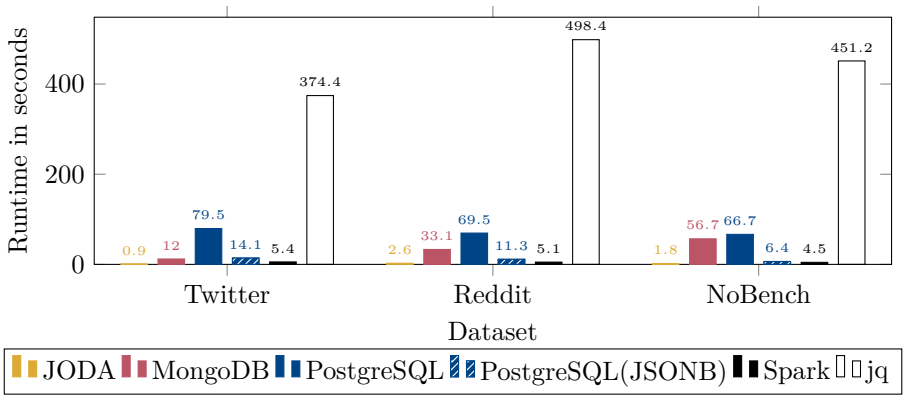


Figure 9.4: Runtime of aggregation (excluding import) in seconds

is now closer to the runtime of jq. Of course, if more than a single query is executed, the import time is amortized and the difference between jq and the other systems will be more significant. JODA and Spark remain the fastest systems, with JODA being slightly faster than Spark. The difference between these two systems is even more significant for the NoBench dataset. The sparse nature of the dataset could be the reason for this difference, as Spark converted each document to the same schema, which now needs to be evaluated for null values at export. In contrast, JODA exports the documents as they are.

Generally, our experiments show that the query performance of the JSONB datatype is much faster than the JSON datatype. In this experiment, the worse performance of the JSON datatype is due to the fact that the data is exported, and the documents have to be reconstructed from the decomposed JSONB column. The JSON column type is already stored in textual representation and can simply be written.

9.3.3 Aggregation

To test the performance of the aggregation, we use the previously imported datasets and perform a straightforward aggregation on the data. For the Twitter dataset, we calculate the average of the `/user/friends_count` attribute, which exists in $\sim$ 80% of all documents and is of type `int`. For NoBench we calculate the count of the `/sparse_100` attribute, which is one of the sparse columns and only exists in 1% of all documents. Lastly, for the Reddit dataset, we calculate the minimum of the `/retrieved_on` attribute, which exists in all documents and is an integer timestamp.

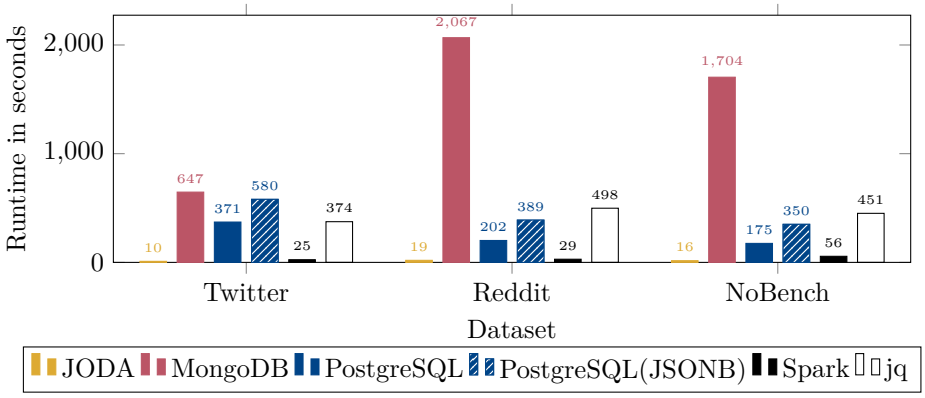


Figure 9.5: Runtime of aggregation (including import) in seconds

The results are shown in Figure 9.4. As we can see, JODA is again the fastest system for all three datasets, closely followed by Spark. For the Reddit and NoBench datasets, PostgreSQL with the JSONB datatype is only slightly slower than Spark, while the remaining systems are significantly slower. But for the Twitter dataset, MongoDB is faster than PostgreSQL with the JSONB datatype, while still taking more than twice as long as Spark. PostgreSQL without the JSONB datatype is significantly slower than the other systems, except for jq, which required by far the longest time. As in the experiments before, we also include the import time in Figure 9.5. Here we can see, that for one-off queries, jq may actually be better suited than MongoDB, and in the case of the Twitter dataset even PostgreSQL with the JSONB datatype. With import included, MongoDB is the slowest system for all datasets. We can clearly see that for MongoDB the number of documents to parse, evaluate, and aggregate has a significant impact on the runtime. The Twitter dataset has only 2.7 million documents, of which 2.2 million are aggregated, while the NoBench dataset has 16.7 million documents, of which only 167 000 are aggregated and the Reddit dataset has 18.1 million documents, of which all are aggregated. This is reflected in the runtime of most systems, but the impact is especially significant for MongoDB.

9.4 Explorative Workloads

In this section, we use the BETZE benchmark generator (see Chapter 8) to generate explorative query workloads to evaluate the performance of the systems. This gives us a better understanding of the performance of the systems in a more realistic exploratory scenario.

9.4.1 Scalability

To test the scalability of the systems in regards to number of CPU threads and dataset size we generated one fixed BETZE session and only modify the input. In these experiments, we only use PostgreSQL with the JSONB column, as the previous experiments showed that the query evaluation time is the fastest of the two PostgreSQL options. We used the default (intermediate) preset, with a

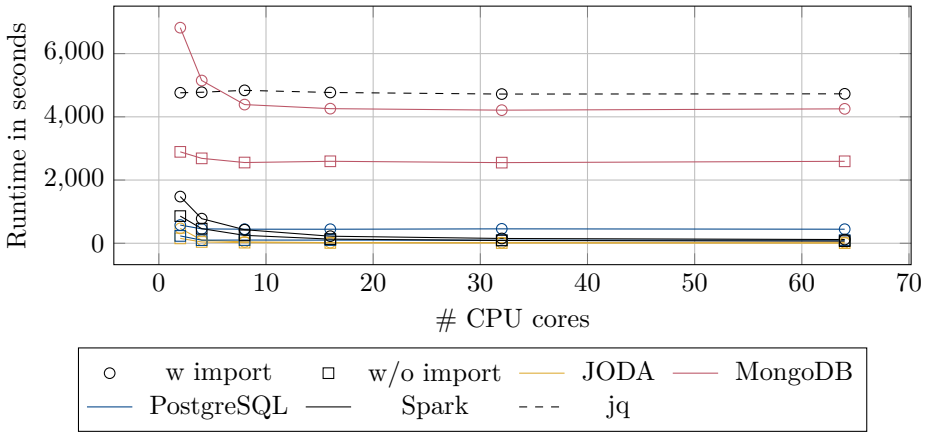


Figure 9.6: Runtime of different systems (in seconds) depending on usable CPU threads

seed of 1 and default settings, to generate one benchmark session. The JODA queries of the session can be found in Appendix C. We double the number of CPU threads between each experiment and calculated the average session runtime of five executions. Figure 9.6 shows the results of these experiments. As the difference between the runtimes of jq and MongoDB in comparison to the other systems is too large, we also show the results without these systems in Figure 9.7.

jq is completely stable and does not depend on the number of CPU cores, as it only uses one thread. PostgreSQL is also very stable after at least 4 cores. For MongoDB, the query evaluation is not affected by the number of CPU cores, but the import step is. Import times reduce significantly up to 16 cores, but do not improve further afterward. Spark and JODA are impacted the most by the number of CPU cores, as both systems make use of all available cores by design. The general order of the systems is the same as in the previous preliminary experiments, where MongoDB and jq are the slowest systems by far. For a larger number of CPU cores, JODA is the fastest system, followed by Spark and PostgreSQL. But for less than 8 cores, PostgreSQL is even faster than Spark.

To evaluate the scalability of the systems in regard to the dataset size, we generate multiple *NoBench* datasets. Figure 9.8 shows the performance of a default session with seed 123 when executed on a NoBench dataset with x documents. Again, Figure 9.9 shows the same results without MongoDB and jq. The shown document counts correspond to approximate dataset sizes of 5.5MB, 55MB, 550MB, and 5.5GB respectively.

As we can see, JODA is the most stable system and generally the fastest for increasing dataset sizes. Again, jq and MongoDB are the slowest systems, but in this experiment jq is slightly faster than MongoDB with the import step included. For PostgreSQL, the import of the JSON documents takes multiple times longer than the evaluation of the whole session. The overall order of the performance of the system remains unchanged from the previous experiments.

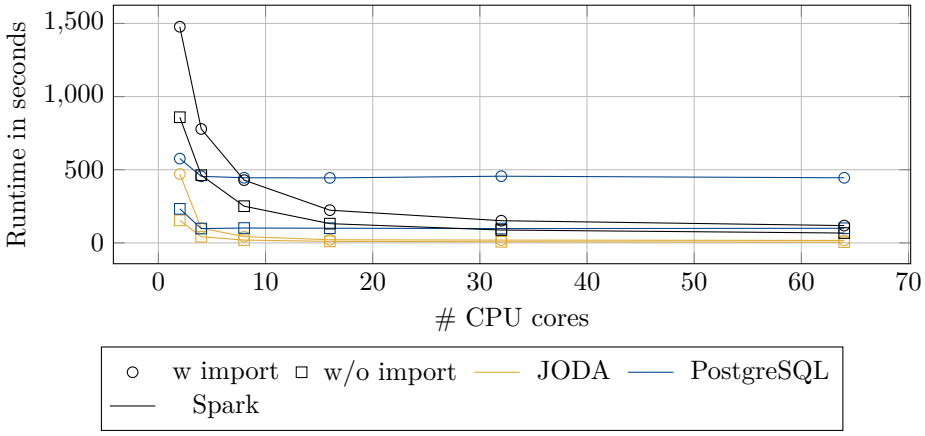


Figure 9.7: Runtime of different systems (in seconds) depending on usable CPU threads. MongoDB and jq excluded.

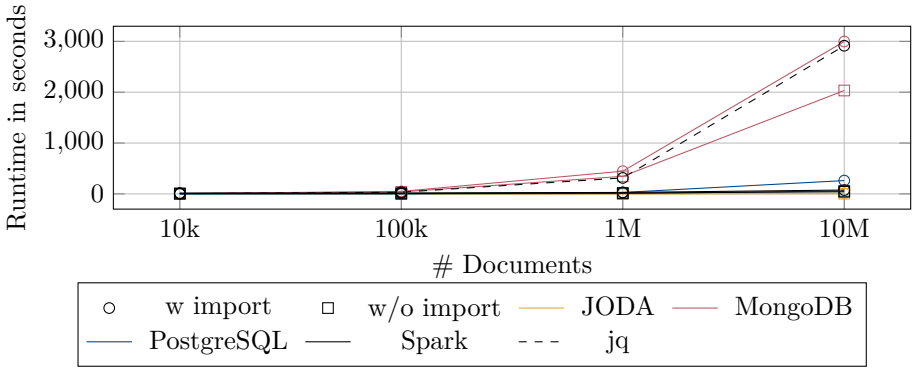


Figure 9.8: Runtime of different systems (in seconds) depending on document count, using the NoBench dataset

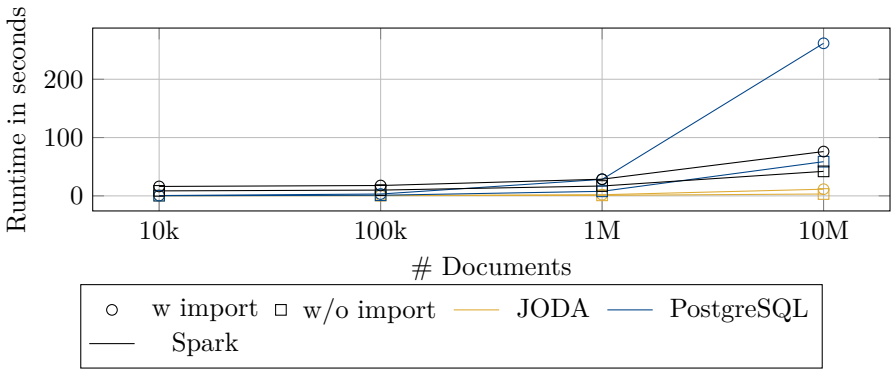


Figure 9.9: Runtime of different systems (in seconds) depending on document count, using the NoBench dataset. MongoDB and jq excluded

Config	Twitter N	Twitter I	Twitter E	NoBench N	NoBench I	NoBench E	Reddit N	Reddit I	Reddit E
JODA	12 s	11 s	10 s	21 s	18 s	18 s	32 s	24 s	21 s
Spark	3 m	1 m	59 s	3 m	2 m	1 m	2 m	1 m	50 s
PSQL	20 m	13 m	13 m	10 m	7 m	73 m	12 m	9 m	7 m
Mongo	82 m	39 m	28 m	-	85 m	70 m	-	-	111 m
JQ	-	84 m	47 m	-	81 m	43 m	-	92 m	46 m

Table 9.1: Session execution time with import of data included for multiple presets and configurations with seed 1

9.4.2 Exploration

To further compare all currently supported systems, we created one session for each preset with a seed of 1 for each of the three datasets. To reduce the impact of disk operations on the benchmark, we did not display or export the result documents, but only executed the query and counted the number of results. We tried to prevent the usage of included aggregation functionality of the systems as we want to compare the performance of the query evaluation only. For JODA, we used the log to extract the number of result documents. The output of jq was piped to the `wc -l` command to count the number of lines. For MongoDB, Spark, and PostgreSQL, we used the counting functionality of the cursor returned by the query.

Table 9.1 shows the execution times of the complete sessions for the three datasets. The configurations *N*, *I*, and *E* correspond to the *novice*, *intermediate*, and *expert* presets of BETZE. As we can see, for the novice preset, which consists of 20 queries, jq did never finish the execution within the timeout of two hours. The same applies for MongoDB with the NoBench and Reddit datasets. Interestingly, for Reddit dataset, MongoDB did also not finish the execution of the intermediate preset within the timeout, while jq did. The Reddit dataset is the largest of the three measured by number of documents. As noted in earlier experiments, the runtime of MongoDB seems highly dependent on the dataset size. Generally, the ranking of the systems is the same as in the previous experiments, with JODA being the fastest system, followed closely by Spark. Then comes PostgreSQL and MongoDB, with jq being the slowest system. Only for the Reddit dataset, the positions of PostgreSQL and MongoDB are swapped.

For every exploration session, JODA finishes the execution in less than one minute. The average waiting time for a single query within the exploration sessions is $\sim$ 2 seconds. This is a significant improvement over the waiting times of the other systems, which range between 12 seconds and 22 minutes.

Overall, as we can see, using jq or MongoDB 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 and Spark can produce tolerable waiting times for interactive data exploration on semi-structured data. But the waiting times of JODA are still significantly lower (up to 12 times) than those of Spark.

Chapter 10

Conclusion & Outlook

In this dissertation, we presented JODA, a novel data processor for JSON data. It is designed to be fast, flexible, and easy to use. We explained the design decisions and the architecture of JODA and gave an overview of the implemented features. An extensive introduction to the straightforward query language and the query execution engine was given. We also showed how the CLI, the web API, and further supporting tools can be used to interact with JODA.

Delta trees were introduced as a novel data structure for storing the results of an iterative query. We explained how the deltas of a transformation step can be efficiently stored for a single document and further optimized the approach with virtual object indices. A close evaluation showed that the performance improvements of delta trees are significant for iterative queries.

To further improve the performance of JODA we introduced adaptive indexing. With each query on a collection, JODA builds structure and content indices that iteratively improve the performance of the filter step of subsequent queries. We showed that the structure index, as well as the content indices for string and numerical values, are very effective for suitable query loads, while only having a small overhead for other query loads.

After improving the performance of JODA we extended the system and query language with support for user-defined functions by decomposing JODA into modules. Support for user-defined import and export modules was added to allow the user to integrate JODA into their custom workflow. User-defined functions allow the user to easily extend the query language with custom functionality. We also added support for user-defined indices that can be specialized for specific workloads or to quickly prototype new ideas. The performance of these modules was on par if not better than a complete implementation in the module programming language, thanks to the architecture of JODA.

To better understand the performance of JODA and other data processors in exploratory workloads, we developed BETZE, a benchmark generator for JSON data processors. It analyses a JSON dataset and creates a query workload that is representative of a data scientist exploring this dataset for the first time.

In a comprehensive evaluation, we compared JODA to other data processors for JSON data. We showed that JODA is significantly faster than other data processors while also being easy to deploy and use. Overall JODA became a very versatile tool for data scientists and developers to explore, process, and analyze JSON data. It fills the noted gap of existing systems for explorative workloads and is a valuable addition to the data processing ecosystem.

10.1 Outlook

During the development of JODA great care has been taken to optimally support vertical scaling to all sizes of computing resources. However, horizontal scaling is an important topic for cloud- and server-based data processors. The multithreaded architecture of JODA and usage of independent containers for query evaluation lend themselves well to horizontal scaling. In a future version, we plan to support horizontal scaling by registering external instances to a central JODA instance. This main instance will then distribute the queries to all registered instances, collect the results, and return the aggregated result set. Aggregating queries can also be easily implemented, as currently JODA already evaluates aggregations in parallel and merges the result in a last post-processing step. The sub-instances only have to return the intermediate results of the aggregation. JODA already supports running in server mode with an integrated web API. This web API would need to be extended to support inter-server communication like registering instances, distributing queries, and returning intermediate aggregation results. Each instance would then execute the query on their local data. In another iteration, automatic data distribution between instances depending on load could be implemented. This would allow to automatically distribute the data to the instances with the least load. This would also allow us to scale the instances up and down depending on the expected workload.

Appendices

Appendix A

List of Functions

In the following, we list all currently implemented functions that can be used in the JODA query language. They are grouped by their functionality and give a brief description of their purpose. Additionally, the online documentation[1] also contains an up-to-date list of all functions with examples.

Mathematical

ABS (<x>) Calculates the absolute value of a number
ACOS (<x>) Calculates the arccosine of the given number
ASIN (<x>) Calculates the arcsine of the given number
ATAN (<x>) Calculates the arctangent of the given number
ATAN2 (<x>, <y>) Calculates the ATAN2 of the given number
CEIL (<x>) Calculates the ceiling of the given (floating point) number
COS (<x>) Calculates the cosine of the given number
DEGREES (<x>) Converts the given radians to degrees
DIV (<x>, <y>) Divides x by y
FLOOR (<x>) Calculates the floor of the given (floating point) number
MOD (<x>, <y>) Returns the remainder of the division of ‘x‘ by ‘y‘
PI () Constant of the number pi.
POW (<x>, <y>) Calculates ‘x‘ to the power of ‘y‘
PROD (<x>, <y>) Multiplies ‘x‘ with ‘y‘
RADIANS (<x>) Converts the given degrees to radians
ROUND (<x>) Rounds the given (floating point) number to the nearest integer number
SIN (<x>) Calculates the sine of the given number
SQRT (<x>) Calculates the square root of ‘x‘
SUB (<x>, <y>) Subtracts ‘x‘ from ‘y‘
SUM (<x>, <y>) Calculates the sum of ‘x‘ and ‘y‘
TAN (<x>) Calculates the tangent of x
TRUNC (<x>) Truncates the floating point number

Cast

FLOAT (<x>) Casts the given value to floating point
INT (<x>) Casts/Parses the given value to integer
STRING (<x>, (<jsonify>)) Converts an atomic value to their string representation

[1] `https://joda-explore.github.io/JODA/functions/`

Iterator

ALL (<iteratable>, <predicate>) Checks if the second parameter is true for all children of the first parameter.

ANY (<iteratable>, <predicate>) Checks if the second parameter is true for any children of the first parameter.

FILTER (<iteratable>, <predicate>) Filters an array with a given predicate.

MAP (<iteratable>, <map function>) Maps array children into another value

Boolean Algebra

AND (<lhs>, <rhs>) Combines the two parameters with the Boolean AND operation.

IMPLICATION (<lhs>, <rhs>) Combines the two parameters with the Boolean implication (->) operation.

NOT (<x>) Negates the Boolean value

OR (<lhs>, <rhs>) Combines the two parameters with the Boolean OR operation.

XOR (<lhs>, <rhs>) Combines the two parameters with the Boolean exclusive or (XOR) operation.

String

CONCAT (<str1>, <str2>) Concatenates two string values

FINDSTR (<str>, <substr>) Returns the position of the second string in the first string, or -1 if it is not contained.

LEN (<str>) Returns the length of the passed string

LOWER (<str>) Converts the given string to lower case

LTRIM (<str>) Trims all whitespace to the left of the string

REGEX (<str>, <regexp>) Checks whether ‘str‘ matches the regular expression ‘regexp‘

REGEX_EXTRACT (<str>, <regexp>) Matches all ‘regexp‘ in ‘str‘ and returns them.

REGEX_EXTRACT_FIRST (<str>, <regexp>) Matches first ‘regexp‘ in ‘str‘ and returns it.

REGEX_REPLACE (<str>, <regexp>, <replace>) Replaces all matches of ‘regexp‘ in ‘str‘ with ‘replace‘

RTRIM (<str>) Trims all whitespace to the right of the string

SCONTAINS (<str>, <substr>) Checks whether ‘str‘ contains ‘substr‘

SPLIT (<str>, <delimiter>) Splits a given string by the supplied delimiter

STARTSWITH (<str>, <substr>) Checks whether ‘str‘ starts with ‘substr‘

SUBSTR (<str>, <start>, (<length>)) Returns a substring of ‘str‘ from ‘start‘ with ‘end‘ characters.

UPPER (<str>) Converts the given string to upper case

Comparison

EQUAL (<lhs>, <rhs>) Checks if the given parameters have the same values.

GREATER (<lhs>, <rhs>) Checks if ‘lhs‘ is greater than ‘rhs‘.

GREATEREQ (<lhs>, <rhs>) Checks if ‘lhs‘ is greater or equal than ‘rhs‘.

LESS (<lhs>, <rhs>) Checks if ‘lhs‘ is less than ‘rhs‘.

LESSEQ (<lhs>, <rhs>) Checks if ‘lhs‘ is less or equal than ‘rhs‘.

UNEQUAL (<lhs>, <rhs>) Checks if the given parameters do not have the same values.

Type

EXISTS (<x>) Checks if the given attribute exists.

FALSY (<x>) Checks whether the given value is falsy.

ISARRAY (<x>) Checks whether the attribute is of type array

ISBOOL (<x>) Checks whether the attribute is of type Bool

ISNULL (<x>) Checks whether the attribute is of type null

ISNUMBER (<x>) Checks whether the attribute is of numerical type

ISOBJECT (<x>) Checks whether the attribute is of type object

ISSTRING (<x>) Checks whether the attribute is of type string

TRUTHY (<x>) Checks whether the given value is truthy.

TYPE () Returns the type of the given attribute

Metadata

FILENAME () Returns the filename (with full path), or "[PROJECTION]" if the document was projected

FILEPOSEND () Returns the end position of the document within the file.

FILEPOSSTART () Returns the starting position of the document within the file.

ID () Returns the unique internal ID of the document.

NOW () Returns the current UNIX timestamp in milliseconds

Misc

HASH (<x>) Computes a hash value of the given value

SEQNUM () Returns a sequential number for every document in the collection

Array

IN (<element>, <array>) Checks whether the ‘element‘ is contained in the ‘array‘

SIZE (<arr>) Returns the size of the array

Object

LISTATTRIBUTES (<obj>) Returns a list of all member names in the given object

MEMCOUNT (<obj>) Returns the number of members in the object

Appendix B

List of Tasks

In the following, we list all tasks that are currently implemented in JODA. They are grouped by their functionality and give a brief description of their purpose.

Parsing

ListFile *Output:* File; *Single threaded*;
Checks if a given file exists and passes on a system path. Also exists as a version for line-separated files.

ListFiles *Output:* File; *Single threaded*;
Iterates all files in a given directory and passes on the system paths. Also exists as a version for line-separated files.

FileOpener *Input:* File; *Output:* Stream; *Single threaded*;
Opens a file and passes on a character stream. Also exists as a version for line-separated files.

URLStream *Output:* Stream; *Synchronous*;
Requests a URL and passes on the response as a character stream.

InStream *Output:* Stream; *Synchronous*;
Passes on the input character stream of JODA (e.g.: piped JSON text).

LSFileMapper *Input:* File; *Output:* String; *Single threaded*;
Uses `mmap` to map a file into memory, scan for the newline character, split lines into strings, and pass them on.

LSStreamReader *Input:* Stream; *Output:* String; *Single threaded*;
Reads character streams and scans for the newline character. Splits lines into strings and passes them on.

StreamParser *Input:* Stream; *Output:* Container; *Multi threaded*;
Reads and parses character streams to JSON documents. The documents are packed into containers and passed on.

StringParser *Input:* String; *Output:* Container; *Multi threaded*;
Parses strings to JSON documents. The documents are packed into containers and passed on.

Storage

StorageSender *Output:* Container; *Synchronous*;
Sends all containers in a collection.

StorageReceiver *Input:* Container; *Synchronous*;
Stores all received containers in a collection.

StorageBuffer *Input:* Container; *Output:* Container; *Synchronous*;
First sends all containers in a collection. Then stores all received containers in the same collection and also passes them on.

Processing

Choose *Input:* Container; *Output:* Container; *Multi threaded*;
Filters containers with a given `CHOOSE` predicate. If the filtered container is empty it is discarded.

As *Input:* Container; *Output:* Container; *Multi threaded*;
Transforms containers with the given `AS` tuples.

Agg *Input:* Container; *Output:* Aggregator; *Multi threaded*;
Aggregates containers with the given `AGG` aggregation functions. Sends the intermediate results after all containers have been processed.

AggMerge *Input:* Aggregator; *Output:* Container; *Single threaded*;
Merges all intermediate aggregation results into a single document and passes it on in a new container.

WindowAgg *Input:* Container; *Output:* Container; *Single threaded*;
Special version of `Agg`. Given an additional window expression, it aggregates the containers in the window and passes on the aggregation result in a new container.

Joins & Groups

LeftJoin *Input:* Container; *Synchronous*;
Stores all received containers in an internal list for later join processing.

RightJoin *Input:* Container; *Output:* Container; *Multi Threaded*;
Joins every received container with every stored container of the previous `LeftJoin` task and returns the result as a new container.

GroupStore *Input:* Container; *Synchronous*;
Groups all documents depending on a group predicate and stores the result in an internal representation either in-memory or as files.

MemoryGroup *Output:* Container; *Single threaded*;
Fetches the in-memory grouped documents, passes them on as containers. The internal group representation is then removed.

FileGroupList *Output:* File; *Single threaded*;
Iterates all group files belonging to the current grouping operation and passes them on.

FileGroupParse *Input:* File; *Output:* Container; *Multi threaded*;
Reads and parses the grouping files and sends the resulting documents as containers.

Optimization

ChooseAs *Input:* Container; *Output:* Container; *Multi Threaded*;
Combination of Choose and As tasks in a tight loop.

ChooseAgg *Input:* Container; *Output:* Aggregator; *Multi Threaded*;
Combination of Choose and Agg tasks in a tight loop.

ChooseAsAgg *Input:* Container; *Output:* Aggregator; *Multi Threaded*;
Combination of Choose, As, and Agg tasks in a tight loop.

Extension

PythonImport *Output:* Container; *Single threaded*;
Executes a given Python import module, packs the result documents into containers, and passes them on.

PythonExport *Input:* Container; *Single threaded*;
Iterates over each document in the passed containers and passes them to the given Python export module.

Appendix C

BETZE Query Session for Scalability

```
LOAD Data
CHOOSE (((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false)

LOAD Data
CHOOSE ((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('/
    sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/sparse_484
    ')) || '/nested_obj/num' <= 1980582.508197))

LOAD Data
CHOOSE (((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('/
    sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/sparse_484
    ')) || '/nested_obj/num' <= 1980582.508197)) && (((ISSTRING('/
    sparse_387') || EXISTS('/sparse_060')) || EXISTS('/sparse_980')
    ) || '/dyn1' <= 7586826.435617))

LOAD Data
CHOOSE (((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('/
    sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/sparse_484
    ')) || '/nested_obj/num' <= 1980582.508197)) && (EXISTS('/
    sparse_884') || ISSTRING('/dyn2')))

LOAD Data
CHOOSE ((((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('/
    sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/sparse_484
    ')) || '/nested_obj/num' <= 1980582.508197)) && (EXISTS('/
    sparse_884') || ISSTRING('/dyn2'))) && (((ISSTRING('/sparse_483
    ') || ISSTRING('/sparse_242')) || ISSTRING('/sparse_540')) ||
    EXISTS('/sparse_143')))

LOAD Data
CHOOSE (((((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('/
    sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/sparse_484
    ')) || '/nested_obj/num' <= 1980582.508197)) && (EXISTS('/
    sparse_884') || ISSTRING('/dyn2'))) && (((ISSTRING('/sparse_483
    ') || ISSTRING('/sparse_242')) || ISSTRING('/sparse_540')) ||
    EXISTS('/sparse_143'))) && EXISTS('/sparse_248'))

LOAD Data
CHOOSE (((((((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('/
    sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/sparse_484
    ')) || '/nested_obj/num' <= 1980582.508197)) && (EXISTS('/
    sparse_884') || ISSTRING('/dyn2'))) && (((ISSTRING('/sparse_483
    ') || ISSTRING('/sparse_242')) || ISSTRING('/sparse_540')) ||
    EXISTS('/sparse_143'))) && EXISTS('/sparse_248')) && '/
    nested_obj/num' <= 3031986.344638)
```

```
LOAD Data
CHOOSE (((((((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
    ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('/
    sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/sparse_484
    ')) || '/nested_obj/num' <= 1980582.508197)) && (EXISTS('/
    sparse_884') || ISSTRING('/dyn2'))) && (((ISSTRING('/sparse_483
    ') || ISSTRING('/sparse_242')) || ISSTRING('/sparse_540')) ||
    EXISTS('/sparse_143'))) && EXISTS('/sparse_248')) && '/
    nested_obj/num' <= 3031986.344638) && ((('/nested_obj/num' ==
    242063 || '/dyn1' == 7510913) || '/nested_obj/num' == 1858662)
    || '/thousandth' <= 746.269983))

LOAD Data
CHOOSE ((((((((((EXISTS('/sparse_462') || EXISTS('/sparse_037')) ||
     ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING('
    /sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/
    sparse_484')) || '/nested_obj/num' <= 1980582.508197)) && (
    EXISTS('/sparse_884') || ISSTRING('/dyn2'))) && (((ISSTRING('/
    sparse_483') || ISSTRING('/sparse_242')) || ISSTRING('/
    sparse_540')) || EXISTS('/sparse_143'))) && EXISTS('/sparse_248
    ')) && '/nested_obj/num' <= 3031986.344638) && ((('/nested_obj/
    num' == 242063 || '/dyn1' == 7510913) || '/nested_obj/num' ==
    1858662) || '/thousandth' <= 746.269983)) && ('/nested_obj/num'
     == 2308517 || '/thousandth' >= 490.820768))

LOAD Data
CHOOSE ((((((((((EXISTS('/sparse_462') || EXISTS('/sparse_037'))
    || ISSTRING('/sparse_776')) || '/bool' == false) && (((ISSTRING
    ('/sparse_248') || ISSTRING('/sparse_149')) || EXISTS('/
    sparse_484')) || '/nested_obj/num' <= 1980582.508197)) && (
    EXISTS('/sparse_884') || ISSTRING('/dyn2'))) && (((ISSTRING('/
    sparse_483') || ISSTRING('/sparse_242')) || ISSTRING('/
    sparse_540')) || EXISTS('/sparse_143'))) && EXISTS('/sparse_248
    ')) && '/nested_obj/num' <= 3031986.344638) && ((('/nested_obj/
    num' == 242063 || '/dyn1' == 7510913) || '/nested_obj/num' ==
    1858662) || '/thousandth' <= 746.269983)) && ('/nested_obj/num'
     == 2308517 || '/thousandth' >= 490.820768)) && (('/num' ==
    6121896 || '/dyn1' <= 5211996.845157) || '/thousandth' <=
    701.459958) && '/nested_obj/num' <= 1726982.896982))
```

Listing C.1: BETZE exploration session used in scalability experiments in Chapter 9.4.1

List of Figures

List of Algorithms

List of Tables

Listings

Bibliography

[1] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8259

[2] 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

[3] N. Schäfer and S. Michel, "Partially materializable delta trees for efficient data wrangling of semi-structured contents," in *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, Eds. OpenProceedings.org, 2020, pp. 399–402. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.42

[4] 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

[5] N. Schäfer and S. Michel, "BETZE: benchmarking data exploration tools with (almost) zero effort," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 2385–2398. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00224

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML)," *World Wide Web J.*, vol. 2, no. 4, pp. 27–66, 1997. [Online]. Available: http://www.w3.org/TR/WD-xml-970807

[7] P. Bryan, K. Zyp, and M. Nottingham, "Javascript object notation (json) pointer," Internet Requests for Comments, RFC Editor, RFC 6901, April 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6901.txt

[8] Tencent Company and M. Yip. (2023) RapidJSON a fast JSON parser/generator for c++ with both SAX/DOM style API. [Online]. Available: http://rapidjson.org/

[9] (2023, jan) jq project website. [Online]. Available: https://stedolan.github.io/jq/

[10] (2023, jan) Postgresql project website. [Online]. Available: https://www.postgresql.org/

[11] Oracle Corporation. (2023, jan) Mysql. [Online]. Available: https://www.mysql.com/

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

[13] A. S. Foundation. (2023) Apache Spark - unified engine for large-scale data analytics. [Online]. Available: https://spark.apache.org/

[14] D. Durner, V. Leis, and T. Neumann, "JSON tiles: Fast analytics on semi-structured 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. 445–458. [Online]. Available: https://doi.org/10.1145/3448016.3452809

[15] D. Tahara, T. Diamond, and D. J. Abadi, "Sinew: a SQL system for multi-structured data," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 815–826. [Online]. Available: https://doi.org/10.1145/2588555.2612183

[16] D. Pritchett, "BASE: an acid alternative," *ACM Queue*, vol. 6, no. 3, pp. 48–55, 2008. [Online]. Available: https://doi.org/10.1145/1394127.1394128

[17] BSON. BSON - binary JSON format. [Online]. Available: http://bsonspec.org/

[18] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 241–252. [Online]. Available: https://doi.org/10.1145/2213836.2213864

[19] K. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, pp. 422–433, 1979. [Online]. Available: https://doi.org/10.1145/322139.322143

[20] P. Bille, "A survey on tree edit distance and related problems," *Theor. Comput. Sci.*, vol. 337, no. 1-3, pp. 217–239, 2005. [Online]. Available: https://doi.org/10.1016/j.tcs.2004.12.030

[21] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and I. S. Mumick, Eds. ACM Press, 1996, pp. 493–504. [Online]. Available: https://doi.org/10.1145/233269.233366

[22] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet, "Change-centric management of versions in an XML warehouse," in *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, Eds. Morgan Kaufmann, 2001, pp. 581–590. [Online]. Available: http://www.vldb.org/conf/2001/P581.pdf

[23] H. Su, D. Kramer, L. Chen, K. T. Claypool, and E. A. Rundensteiner, "XEM: managing the evolution of XML documents," in *Eleventh International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications, Heidelberg, Germany, 1-2 April 2001*, K. Aberer and L. Liu,

Eds. IEEE Computer Society, 2001, pp. 103–110. [Online]. Available: https://doi.org/10.1109/RIDE.2001.916497

[24] Y. Wang, D. J. DeWitt, and J. Cai, "X-diff: An effective change detection algorithm for XML documents," in *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, U. Dayal, K. Ramamritham, and T. M. Vijayaraman, Eds. IEEE Computer Society, 2003, pp. 519–530. [Online]. Available: https://doi.org/10.1109/ICDE.2003.1260818

[25] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *J. Parallel Distrib. Comput.*, vol. 111, pp. 162–173, 2018. [Online]. Available: https://doi.org/10.1016/j.jpdc.2017.08.003

[26] L. Torvalds and J. Hamano. (2010) Git: Fast version control system. [Online]. Available: https://git-scm.com/

[27] A. Subversion. (2011) Enterprise-class centralized version control for the masses. [Online]. Available: https://subversion.apache.org/

[28] GNU. (1998) Cvs - concurrent versions system. [Online]. Available: https://www.nongnu.org/cvs/

[29] J. W. Hunt and M. D. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[30] D. Korn, J. MacDonald, J. Mogul, and K. Vo, "The vcdiff generic differencing and compression data format," Internet Requests for Comments, RFC Editor, RFC 3284, June 2002.

[31] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden, "Materialization strategies in a column-oriented DBMS," in *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, Eds. IEEE Computer Society, 2007, pp. 466–475. [Online]. Available: https://doi.org/10.1109/ICDE.2007.367892

[32] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan, "On the integration of structure indexes and inverted lists," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, G. Weikum, A. C. König, and S. Deßloch, Eds. ACM, 2004, pp. 779–790. [Online]. Available: https://doi.org/10.1145/1007568.1007656

[33] C. Chung, J. Min, and K. Shim, "APEX: an adaptive path index for XML data," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, M. J. Franklin, B. Moon, and A. Ailamaki, Eds. ACM, 2002, pp. 121–132. [Online]. Available: https://doi.org/10.1145/564691.564706

[34] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. J. Levandoski, and D. B. Lomet, "Schema-agnostic indexing with azure documentdb," *Proc.*

VLDB Endow., vol. 8, no. 12, pp. 1668–1679, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1668-shukla.pdf

[35] T. Kissinger, H. Voigt, and W. Lehner, "SMIX live - A self-managing index infrastructure for dynamic workloads," in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, A. Kementsietsidis and M. A. V. Salles, Eds. IEEE Computer Society, 2012, pp. 1225–1228. [Online]. Available: https://doi.org/10.1109/ICDE.2012.9

[36] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri, "Automatically indexing millions of databases in microsoft azure SQL database," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 666–679. [Online]. Available: https://doi.org/10.1145/3299869.3314035

[37] J. Arulraj, R. Xian, L. Ma, and A. Pavlo, "Predictive indexing," *CoRR*, vol. abs/1901.07064, 2019. [Online]. Available: http://arxiv.org/abs/1901.07064

[38] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 2007, pp. 68–78. [Online]. Available: http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[39] G. Graefe and H. A. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, ser. ACM International Conference Proceeding Series, I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, Eds., vol. 426. ACM, 2010, pp. 371–381. [Online]. Available: https://doi.org/10.1145/1739041.1739087

[40] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe, "Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 585–597, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p586-idreos.pdf

[41] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt, "Progressive indexes: Indexing for interactive data analysis," *Proc. VLDB Endow.*, vol. 12, no. 13, pp. 2366–2378, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2366-holanda.pdf

[42] S. Gupta and K. Ramachandra, "Procedural extensions of SQL: understanding their usage in the wild," *Proc. VLDB Endow.*, vol. 14, no. 8, pp. 1378–1391, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1378-ramachandra.pdf

[43] E. Friedman, P. M. Pawlowski, and J. Cieslewicz, "Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-

defined functions," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1402–1413, 2009. [Online]. Available: http://www.vldb.org/pvldb/vol2/vldb09-464.pdf

[44] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1466–1477, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1466-crotty.pdf

[45] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The madlib analytics library or MAD skills, the SQL," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1700–1711, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p1700_joehellerstein_vldb2012.pdf

[46] L. Passing, M. Then, N. C. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, "SQL- and operator-centric data analytics in relational main-memory databases," in *EDBT*. OpenProceedings.org, 2017, pp. 84–95.

[47] M. E. Schüle, J. Huber, A. Kemper, and T. Neumann, "Freedom for the sql-lambda: Just-in-time-compiling user-injected functions in postgresql," in *SSDBM*. ACM, 2020, pp. 6:1–6:12.

[48] C. Duta, D. Hirn, and T. Grust, "Compiling PL/SQL away," in *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p1-duta-cidr20.pdf

[49] D. Hirn and T. Grust, "One WITH RECURSIVE is worth many gotos," 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. 723–735. [Online]. Available: https://doi.org/10.1145/3448016.3457272

[50] M. Sichert and T. Neumann, "User-defined operators: Efficiently integrating custom algorithms into modern databases," *Proc. VLDB Endow.*, vol. 15, no. 5, pp. 1119–1131, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p1119-sichert.pdf

[51] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. N. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqui, and S. B. Wrede, "Systemds: A declarative machine learning system for the end-to-end data science lifecycle," in *CIDR*. www.cidrdb.org, 2020.

[52] M. E. Schüle, L. Scalerandi, A. Kemper, and T. Neumann, "Blue elephants inspecting pandas: Inspection and execution of machine learning pipelines in SQL," in *EDBT*. OpenProceedings.org, 2023, pp. 40–52.

[53] (2023, jan) Tpc project website. [Online]. Available: http://www.tpc.org/

[54] 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

[55] 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

[56] 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

[57] 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

[58] 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

[59] (2021, nov) Trec conference website. [Online]. Available: https://trec.nist.gov

[60] 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.

[61] 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

[62] 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

[63] 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

[64] 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

[65] 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

[66] 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

[67] 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

[68] 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

[69] 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

[70] 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

[71] G. Nicol, M. Champion, J. Robie, A. L. Hors, L. Wood, R. S. Sutor, S. Isaacson, C. Wilson, S. B. Byrne, and I. Jacobs, "Document object model (DOM) level 1," W3C, W3C Recommendation, Oct. 1998, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.

[72] M. Dyck, J. Spiegel, and J. Robie, "XML path language (XPath) 3.1," W3C, W3C Recommendation, Mar. 2017, https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

[73] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[74] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb in action: Adaptive query processing on raw data," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1942–1945, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p1942_ioannisalagiannis_vldb2012.pdf

[75] F. M. Schuhknecht, J. Dittrich, and L. Linden, "Adaptive adaptive indexing," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 665–676. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00066

[76] S. Lang, "Adaptive indexing of semi-structured data," 2020.

[77] D. R. Morrison, "PATRICIA - practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968. [Online]. Available: https://doi.org/10.1145/321479.321481

[78] S. Sprenger, P. Schäfer, and U. Leser, "Bb-tree: A main-memory index structure for multidimensional range queries," in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 1566–1569. [Online]. Available: https://doi.org/10.1109/ICDE.2019.00143

[79] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-Art Natural Language Processing." Association for Computational Linguistics, 10 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[80] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[81] 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

[82] 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

[83] G. Doniparthi, T. Mühlhaus, and S. Deßloch, "A hybrid data model and flexible indexing for interactive exploration of large-scale bioscience 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

[84] 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

[85] 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

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

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

[88] 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

[89] 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

[90] 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

[91] 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

[92] 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

[93] 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

Nico Schäfer

Master of Science in Computer Science

Education

2018–Today **Doctoral Studies**, *RPTU Kaiserslautern-Landau*
Topic: On Enabling Efficient and Scalable Processing of Semi-Structured Data

2015–2018 **MSc. Informatik**, *TU Kaiserslautern*
Thesis: Vertically-Scaled JSON On-Demand Processing With JODA
Major: Information systems

2011–2015 **BSc. Informatik**, *TU Kaiserslautern*
Thesis: An SQL-based Intermediate Layer for Table Transformations on Wide-Column Stores
Major: Software-Engineering

2002–2011 **Abitur**, *IGS Kurt Schumacher*, Ingelheim am Rhein

Work Experience

2015–Today **Research associate**, *TU Kaiserslautern / RPTU Kaiserslautern-Landau*
Employment within the framework of the doctorate.
Organization of exercises, seminars, supervision of theses.
DigiVine project

2012–2018 **Student assistant**, *TU Kaiserslautern*
Various student assistant activities.
IT-Service FB WiWi
Tutoring Information systems

nschaefer@cs.uni-kl.de