JODA: A Vertically Scalable, Lightweight JSON Processor for Big Data Transformations

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

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

***Abstract*—We describe the demonstration of JODA (Json On Demand Analytics), an approach to handling large amounts of JSON documents in a vertically scalable manner. With JODA, the user can import, filter, transform, aggregate, group, and export documents with a simple PIG-style query language, offering fast execution speed. This is achieved by utilizing a multithreaded architecture over disjoint, read-only containers of data that are processed in parallel, similar to what RDDs are to Spark. Containers are augmented with auxiliary information like Bloom filters and adaptive indices and all containers are processed in parallel by individual threads. By avoiding locks, latches, and synchronization beyond simple thread pooling, we do not risk contention and therefore maximize resource utilization. The demonstration scenarios aim at engaging visitors with several data analytics tasks around large, real-world datasets that are to be solved with the help of JODA, and further gives insights on system internals and the installation/configuration process.**

***Index Terms*—in-memory, json, semi-structured**

I. Introduction

Data scientists and practitioners are commonly faced with large amount of structured or semi-structured data, often in form of CSV or JSON files, which need to be processed, analyzed, stored, or published. For this task, tools have to be used that are ideally designed for these use cases or provide the performance needed to explore millions of documents in a timely fashion. Many traditional relational database management systems (RDBMS) were augmented 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 mandatory for simple exploration tasks. NoSQL document stores, on the other hand, are designed specifically for query evaluation of semi-structured documents, without explicit support of transactions or rich query languages. Such NoSQL stores are optimized for horizontal scaling and seemingly fall short in scaling vertically, that is, they do not fully exploit 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, leading to a slow import step, which is not in the spirit of "NoDB" [1] solutions that would avoid heavy indexing at the first place. 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 skills that may not be available.

JODA is designed as a very lean, seemingly simple tool, optimized for ad-hoc data transformations and aggregations of raw JSON data on a single, multi-core system. One can think of it as a command line tool for JSON data manipulation and transformation, as an enabling, filter and aggregation, step to further processing or publication. The key features of JODA are:

- Fast execution times by fully utilizing available system resources and a clear focus on core features set.
- Simple query language for selection, projection, and aggregation.
- Automated dynamic indexing, depending on data and query load.
- Adaptive in-memory or file-based evaluation, depending on available resources.
- Easy to deploy and use, but highly configurable if required.

With the proposed demonstration, the ICDE audience is invited to handcraft queries over tens to hundreds of gigabyte data in a JODA instances deployed on a local 1TB RAM, 96 cores server as well as on a off-the-shelf consumer laptop we will bring along. The demonstration scenarios allow investigating internal features of JODA as well as hands-on solving data analytics and transformation tasks over large amounts of Twitter tweets and other benchmark datasets.

II. Related Work

The work on JODA is related to three recent research directions in data management. First, work on processing raw data without heavy, upfront indexing, like the NoDB approach by Alagiannis et al. [1] and more recent related approaches [2],

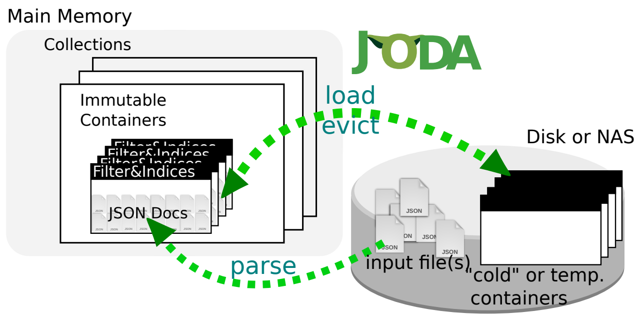


Fig. 1: High-level architecture of JODA. Documents are parsed and stored in containers. Collections are named sets of containers, representing individual datasets.

[3]; respectively approaches aiming at fast integration/loading of raw files into database systems [4]. Second, systems and foundations around the NoSQL "movement" with application-tailored approaches and less ACID, with approaches like Amazon Dynamo, the Apache Spark/SQL stack, and MongoDB. Third, there is research on in-memory databases that exploit modern multi-core machines [5], [6]. JODA is not meant to be a database or data store, it is designed as an extremely lightweight data processing tool, for data transformations and aggregations. It is in strong contrast to NoSQL document stores or data processing using MapReduce or Spark, but borrows provenly beneficial concepts from various approaches, like the RDD-style containers, the PIG-style query language, query caching, and adaptive indexing. Parsing JSON documents is a relatively costly task and there is recent advances [7] on novel parsers tailored to the peculiarities of JSON, which could be integrated into JODA. The need for simple JSON processors have been partially addressed with tools like `jq`, which provides command-line JSON processing capabilities and is available within 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 command line. However, the applicability of such tools is often confined to single threads or other restrictions, rendering them inapplicable for handling large(r) amounts of data in acceptable response times.

III. The JODA Approach

The internals of JODA are centered around containers, as shown in Fig. 1, a simple abstraction that represents a set of JSON documents and, if any, stores associated indices that describe the documents of the container. Collections are named sets of containers, representing individual datasets. There is no global directory to, for instance, map JSON paths to containers, also there are no global indices on numeric or textual attributes of documents. Each query is evaluated on a specific collection of documents, which may first have to be imported, from raw input files, or from previously computed and stored collections. The user may choose to filter the set of documents, by providing a filter predicate, which is evaluated against every document in the set. This predicate consists of functions and can access all values contained in a document. All selected documents may then be transformed into one or multiple output documents, by creating and modifying attribute names and values. Optionally, it is possible to aggregate these documents into a single result document. The result set can then be stored in the system or be output to the user.

```
#import entire directory containing JSON files
LOAD twitter FROM FILES "twitter/";
#or reading single file
LOAD twitter FROM FILE "twitter.json";
```

Listing 1: JODA import commands

A. Parsing

To enable the parsing of different file formats and sources (cf., Listing 1), JODA is able to use multiple readers at the same time. Each reader is responsible to transform sources into one or multiple strings, each containing one document. JODA uses RapidJSON[1] to parse the document into a DOM-tree. The parsed documents are stored in containers and are associated with a specific collection name. Containers and collections are read only. Through queries, collections are transformed into new collections, e.g., by selecting all tweets of a certain geolocation and to project the attributes on author, name, and language. Ideally, the amount of available memory is large enough to accommodate all containers of all collections that are currently relevant to a user, hence, have been loaded. If this is not the case, containers can be serialized to disk, and loaded on demand. This serialization lowers the query performance, but enables JODA to evaluate large data sets also on commodity hardware.

B. Per-Container Bloom Filter and Indices

Auxiliary data structures to speed up processing can be stored together with the documents inside containers. JODA uses Bloom filter on JSON schema paths for skipping whole containers and database cracking [8], [9] for comparable data types. This index extracts the numerical attributes of multiple documents and stores them in a cracking column. The column is then iteratively cracked into smaller parts, depending on the query, and an index for fast access is built. Internally, this index is implemented as an AVL tree, where the nodes are pivot elements and the leaves are sets of document ids.

In order to allow skipping entire containers at query time, Bloom filter [10] are computed per container. This index is created at parsing time by adding all attribute-paths, contained in all documents. As containers are finalized after parsing, the filter does not need to be adjusted after its initialization. Before evaluating the filter step, the Bloom filter is queried against all

[1]https://github.com/Tencent/rapidjson

```
LOAD twitter FROM FILES "/twitter"
CHOOSE '/lang' == "EN"
AS ('/user':'/user'),('/text':'/text'),
   ('/lang':"English")
STORE AS FILE "english_tweets.json";
```

Listing 2: Sample query

mandatory paths in the predicate. Mandatory paths, are paths referenced in parts of the predicate that are required (i.e., all paths that are not within `OR` expressions). The container is skipped, if at least one such path is certainly not contained in any document of the container.

Additionally, a simple query cache is implemented to rapidly answer queries or subsets of queries that were posted before, without touching the documents. For each query, the filter predicate is stored, together with a set of document ids within the container. If the same filter predicate is used at a later time, the result can be returned without accessing the documents.

C. Memory Management

Most of the operations in JODA are performed in-memory. The documents themselves are stored in DOM-trees that may be changed dynamically. Each query may have to copy the whole tree or parts of it multiple times for each document to build the result documents. Each of these operations requires memory (de-)allocations, which are relatively slow.

JODA uses memory-pool allocators to reduce the amount of required OS (de-)allocation instructions. All documents within a container share the same memory-pool. Each document related allocation uses this pool to receive a range of pre-allocated memory in a fraction of the time it would require to allocate the memory on the heap. As no documents in a container can be removed or changed, only allocations are allowed. This makes keeping track of pointers and implementing deallocation procedures unnecessary. To remove the documents from memory, all documents in a container have to be removed and the memory-pool has to be deallocated completely.

D. Query Language

We implemented a simple PIG-style query language in JODA. The queries consist of up to six commands, enabling the user to import (`LOAD`), filter (`CHOOSE`), transform (`AS`), aggregate (`AGG`), export (`STORE`), and delete (`DELETE`) data.

Listing 2 shows a sample query that is loading twitter data from a directory containing JSON files. Subsequently, the query is then filtering out non-English tweets, simplifies these tweets, and finally exports the result into a file on the file system.

A variety of functions, like simple arithmetics, string manipulation, statistics, and meta-data retrieval, are implemented to interact with the documents. These functions can be used during the filter, transformation and aggregation step. Furthermore, JODA provides basic capabilities for grouping documents by value.

E. Query Evaluation

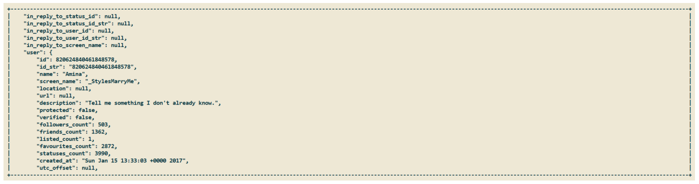
The filter, transform, and aggregation phases, are executed parallel on multiple containers within the chosen collection. For this, a dynamic pool of query threads is created. Each of these threads fetches a container, evaluates the query on it and potentially outputs a result container, which is put into the result collection in return.

Before executing the query, it is optimized using a combination of techniques. Filter predicates may contain sub-predicates that could be evaluated without documents. For example, is it possible to calculate `SUM(1,1)` or answer `SCONTAINS("Hello World","Hello")` without accessing any attributes of documents. After parsing, the system will try to find these occurrences, by traversing the predicate tree, a representation of the predicate, where each function and constant value is a node. Starting at the bottom, each of these functions is replaced by the static result value, until no more functions can be replaced. If the whole filter predicates is replaced by either a `TRUE` or `FALSE` constant value, the filter step is executed without accessing documents or indices.

The query may also contain transformation functions that return more than one document. For example, is a normal copy transformation (e.g., copy JSON pointer `'/text'` to destination pointer `'/newtext'`) executed once per document. Multi-document transformations (e.g.: splitting up an array and copying each element into a new document) generate multiple new documents from a single source document. The order of these transformations has a large impact on how many copy operations are executed. To reduce the amount of these operations, JODA reorders the transformation functions. First, all single-document functions are executed to create a single template document. This document is then passed to the multi-document transformation functions, which will create multiple result documents. This potentially reduces the number of copy operations from $n * K$ to $n + K$, where n is the number of single-document functions and K is the number of documents created by the multi-document functions.

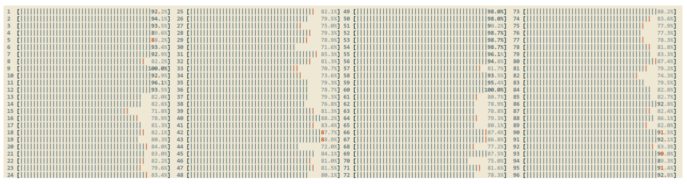
IV. Description of Demonstration

Attendees of our demo can experience the entire workflow of installing and configuring JODA and start exploring and processing large JSON datasets with it. We will provide several real-world data sets and benchmarks like NoBench. For instance, several GB of Twitter tweets, in the native JSON syntax obtained through the Twitter API, can be explored by users during an interactive session. Queries can be executed locally on the provided laptop or remotely on our 96 core, 1TB RAM server to witness the versatility of the program in regards to system configurations. We will also provide USB sticks and download links for a tarball, respectively Debian package, to allow users installing the software on their own laptops. A demonstration video of JODA is available on YouTube under https://www.youtube.com/watch?v=xjv8yDw8Z5I.



```
+-------------------------------------------------------------------------------+
|   "in_reply_to_status_id": null,                                              |
|   "in_reply_to_status_id_str": null,                                          |
|   "in_reply_to_user_id": null,                                                |
|   "in_reply_to_user_id_str": null,                                            |
|   "in_reply_to_screen_name": null,                                            |
|   "user": {                                                                   |
|       "id": 820624840461848578,                                               |
|       "id_str": "820624840461848578",                                         |
|       "name": "Amina",                                                        |
|       "screen_name": "_StylesMarryMe",                                        |
|       "location": null,                                                       |
|       "url": null,                                                            |
|       "description": "Tell me something I don't already know.",              |
|       "protected": false,                                                     |
|       "verified": false,                                                      |
|       "followers_count": 503,                                                 |
|       "friends_count": 1362,                                                  |
|       "listed_count": 1,                                                      |
|       "favourites_count": 2872,                                               |
|       "statuses_count": 3990,                                                 |
|       "created_at": "Sun Jan 15 13:33:03 +0000 2017",                         |
|       "utc_offset": null,                                                     |
+-------------------------------------------------------------------------------+
```

(a) Result Display



```
1  [ 92.2%] 25 [ 82.1%] 49 [ 98.0%] 73 [ 88.2%]
2  [ 94.1%] 26 [ 79.5%] 50 [ 98.0%] 74 [ 83.6%]
3  [ 93.5%] 27 [ 75.0%] 51 [ 90.2%] 75 [ 77.9%]
4  [ 89.6%] 28 [ 79.3%] 52 [ 98.7%] 76 [ 77.3%]
5  [ 88.2%] 29 [ 78.9%] 53 [ 98.7%] 77 [ 78.3%]
6  [ 93.4%] 30 [ 71.6%] 54 [ 98.7%] 78 [ 81.8%]
7  [ 92.9%] 31 [ 85.3%] 55 [ 96.1%] 79 [ 83.3%]
8  [ 82.2%] 32 [ 81.3%] 56 [ 94.8%] 80 [ 87.4%]
9  [100.0%] 33 [ 70.7%] 57 [ 81.7%] 81 [ 79.2%]
10 [ 92.9%] 34 [ 73.6%] 58 [ 93.5%] 82 [ 74.3%]
11 [ 96.1%] 35 [ 79.3%] 59 [ 95.4%] 83 [ 79.5%]
12 [ 93.5%] 36 [ 78.7%] 60 [100.0%] 84 [ 82.8%]
13 [ 82.0%] 37 [ 79.3%] 61 [ 80.7%] 85 [ 82.7%]
14 [ 82.6%] 38 [ 76.8%] 62 [ 78.9%] 86 [ 92.8%]
15 [ 71.8%] 39 [ 81.3%] 63 [ 78.8%] 87 [ 82.4%]
16 [ 78.9%] 40 [ 88.2%] 64 [ 79.3%] 88 [ 86.1%]
17 [ 81.3%] 41 [ 83.4%] 65 [ 80.1%] 89 [ 82.0%]
18 [ 82.1%] 42 [ 87.7%] 66 [ 87.4%] 90 [ 91.5%]
19 [ 80.3%] 43 [ 88.9%] 67 [ 86.8%] 91 [ 92.1%]
20 [ 84.0%] 44 [ 72.0%] 68 [ 77.2%] 92 [ 83.3%]
21 [ 83.0%] 45 [ 84.1%] 69 [ 87.5%] 93 [ 90.0%]
22 [ 82.2%] 46 [ 81.0%] 70 [ 75.0%] 94 [ 89.3%]
23 [ 79.6%] 47 [ 81.5%] 71 [ 81.6%] 95 [ 91.4%]
24 [ 83.4%] 48 [ 80.1%] 72 [ 78.3%] 96 [ 92.8%]
```

(b) Screenshot of Query Execution Display

Fig. 2: Screenshots of JODA (images cropped for readability)

A. *Querying*

We provide a set of query templates users can use to gradually construct results toward solving small data science problems. Users are also free to write additional queries. For example could a user get all hashtags in a Twitter dataset by using the following query:

```
LOAD twitter
AS ('': FLATTEN('/entities/hashtags'))
AGG ('/hashtags': DISTINCT('/text'))
STORE ht;
```

Listing 3: Computing distinct hashtags in JODA

While the queries are executed, users can investigate the resource utilization of the system in terms of CPU usage and memory allocation, as shown in Fig. 2b. In another task, geolocations can be extracted from the Twitter dataset, according to user-provided keywords or #hashtags and translated into the GeoJSON[2] format. The extracted locations can then be used to visualize the tweets by placing them on a map using geojson.io.

B. *Performance Comparison*

We prepare equivalent queries in MongoDB, as well as UNIX standard tools like grep and sed, and dedicated JSON processing tools like `jq`[3]. Users can experience the ease of use and significant performance gains of JODA.

In experiments we have conducted on Twitter data, comparing the performance of JODA to competitors MongoDB, Spark, and variants of Postgres (using Text or JSONB for storing JSON content), we have seen up to one order of magnitude performance gains over Spark and MongoDB (running in local, centralized server mode) and up to three orders of magnitude improvements over Postgres.

C. *Setup and Configuration*

Finally, a big benefit of JODA is the ease of setup and modest dependency on third party libraries, e.g., compared to what comes along with a Spark installation. We show users how to bring a bare Linux installation running on an AWS host to install and get started with JODA, in literally less than a minute.

[2]https://tools.ietf.org/html/rfc7946
[3]https://stedolan.github.io/jq/

We will demonstrate that the default settings are set to sensible values, like container sizes, such that the program utilizes all available resources of the system (cf., Fig. 2b). We invite visitors to execute the same query under different configurations and to compare execution time and resource consumption.

V. Conclusion

We proposed the demonstration of JODA, a tool designed for high-performance, easy-to-use data exploration, analysis, and transformation, for large amounts of JSON data. It uses a custom, PIG-style query language, which integrates path expressions as first class citizens and supports dynamic data types for all operations. JODA is optimized for vertically scaled machines, but supports deployment on low end machines, too. We invite the ICDE audience to get engaged with JODA, by hands-on experiences in a pre-selected or open set of tasks over different datasets and scales.

References

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

[2] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia, "Filter before you parse: Faster analytics on raw data with sparser," *PVLDB*, vol. 11, no. 11, pp. 1576–1589, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1576-palkar.pdf

[3] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki, "Slalom: Coasting through raw data via adaptive partitioning and indexing," *PVLDB*, vol. 10, no. 10, pp. 1106–1117, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1106-olma.pdf

[4] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann, "Instant loading for main memory databases," *PVLDB*, vol. 6, no. 14, pp. 1702–1713, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1702-muehlbauer.pdf

[5] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "Leanstore: In-memory data management beyond main memory," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, 2018, pp. 185–196. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00026

[6] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *PVLDB*, vol. 8, no. 3, pp. 209–220, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol8/p209-yu.pdf

[7] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: A fast JSON parser for data analytics," *PVLDB*, vol. 10, no. 10, pp. 1118–1129, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1118-li.pdf

[8] S. M. Stratos Idreos, Martin L. Kersten, "Database cracking," in *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, 2007, pp. 68–78.

[9] F. M. Schuhknecht, A. Jindal, and J. Dittrich, “An experimental evaluation and analysis of database cracking,” *VLDB J.*, vol. 25, no. 1, pp. 27–52, 2016. [Online]. Available: https://doi.org/10.1007/s00778-015-0397-y

[10] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.