Protobase: It's About Time for Backend/Database Co-Design

A Demo on Rapid Microservice Prototyping for Third-Party Dataset Analytics

Marcus Pinnecke[1], Gabriel Campero[1], Roman Zoun[1], David Broneske[1], Gunter Saake[1]

Abstract: In this interactive demonstration, we show the current state of Protobase, our main-memory analytic document store that is designed from scratch to enable rapid prototyping of efficient microservices that perform analytics and explorations on (third-party) JSON-like documents stored in a novel columnar binary-encoded format, called the Cabin file format. In contrast to other solutions, our database system exposes neither a particular query language, nor a fixed REST API to its clients. Instead, the entire user-defined backend logic, whose user code is written in Python, is placed inside a sandbox that runs in the systems process. Protobase in turn exposes a *user-defined* REST API that the (frontend) application interacts with. Thus, our system acts as a backend server while at the same time avoids full exposure of its database to the clients. Consequently, a Protobase instance (database + user code + REST API) serves as (the entire) microservice - potentially minimizing the number of systems running in a typical analytic software stack. In terms of execution performance, Protobase therefore takes the inter-process communication overhead between backend and database system out of the picture *and* heavily utilizes columnar binary document storage to scale-up for analytic queries. Both features lead to a notable performance gain for non-trivial services, potentially minimizing the number of required nodes in a cloud setting, too. In our demo, we overview Protobases internals, spot major design decisions, and show how to prototype a scholarly search engine managing the Microsoft Academic Graph, a real-world scientific paper graph of roughly 154 mio. documents.

Keywords: NoSQL; Document Stores; Analytics; Rapid Prototyping; Backend/Database Co-Design

1 Analytics on Schema-Free and Hierarchical Datasets

There is a common saying that the early bird catches the worm. This popular phrase serves as an advice on performing an action as soon as possible before competitors are able to do so to maximize the own fitness. Therefore, it is not surprising that service providers have a certain need to offer their solutions to the market quickly. Consequently, agile development methods[2] have established themselves as de-facto standard for many developer teams today. In such a context, tools that help the agile workflow are favored while cumbersome non-agile tools are typically avoided. Especially for agile microservice development, document stores rather than relational systems are often the first choice when it comes to large-scale data management. One reason for choosing document stores is the native support of the first-class citizen data exchange format for many (web) developers, the JavaScript Object Notation

[1] DBSE Working Group, University of Magdeburg, (pinnecke|campero|broneske|zoun|saake)@ovgu.de
[2] Manifesto for Agile Software Development: `https://agilemanifesto.org`

doi:10.18420/btw2019-35

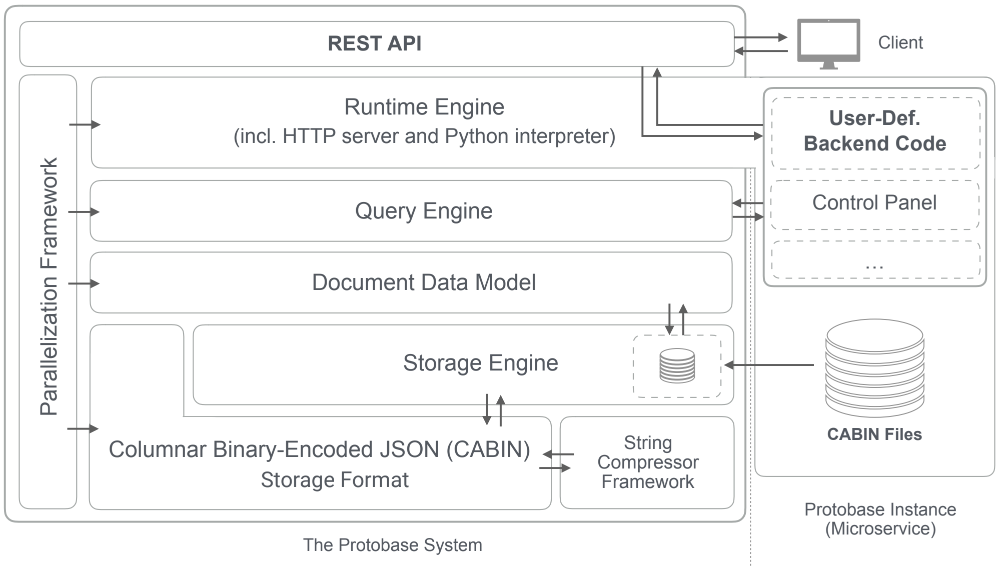

Fig. 1: PROTOBASE architecture including client request to and interaction with user code

(JSON) format[3]. Even if agile development is not an argument, it is likely that a team is confronted with JSON when working with a third-party data source, anyway. Since it is expressive, flexible, human readable, and easy to parse, JSON (or XML as an alternative) is a prominent serialization format for data exchange. However, *exploring* and *analyzing* large repositories of such JSON files can quickly become a time-consuming, non-trivial task [LKC+17, WZS+18].

A third-party provided JSON file (or *document*) is typically small: only a handful of properties of potentially incomplete records are contained on each tree level, and the typical maximum depth of a JSON tree is smallish, too. For example, a request to GitHub's[4a] repository API may return only a single document of approx. 5 KB size. Such a document contains around 50 (mostly string-typed) properties, and one nested record with around 20 properties providing some information on the repository owner. Naturally, this is *not* a good starting point for any analytic system. Therefore, a typical document store is operational rather than analytical, and will store a single document as a single unit of information following a key-value pair serialization (maybe binary encoded, such as BSON[4b] or UBJSON[4c]). Since analytics favor columnar storage schemes instead of key-value storage schemes, it is common practice for analytics on a high number of documents to be based on several technologies in orchestration. Namely, a state-of-the-art software stack may consists of Kubernetes[4d], Docker[4e], MongoDB[4f], Elasticsearch[4g], Spark[4h], and others. Hence, many systems are needed to fulfill a particular task - which is a challenge by its own [LHB13] - even if the queries are quite simple.

[3] RFC 7159, *The JSON Data Interchange Format (Marc 2014)*: `https://tools.ietf.org/html/rfc7159`

[4] [a]GitHub: `https://github.com`, [b] Binary JSON: `http://www.bsonspec.org`, [c] Universal Binary JSON: `http://www.ubjson.org`, [d] Kubernetes: `https://www.kubernetes.io`, [e] Docker: `https://www.docker.com`, [f] MongoDB: `https://www.mongodb.com`, [g] Elasticsearch: `https://www.elastic.co/de`, [h] Spark: `https://spark.apache.org`

We overcome the limitation that naturally leads to key-value pair serialization: we convert a set of JSON files into a single file that is formatted columnwise and binary encoded, a Cabin file. Likewise, we focus on backend/database co-design for development teams that are issued with a particular task: quickly deploying fast-responding microservices to create novel value on third-party, schema-free, hierarchical, read-mostly datasets by providing analysis, aggregation and exploration features. As a case study, we decide for a scholar search engine motivated by our previous work [CJP+18]. Our experiences call for a Protobase instance directly deployed as backend for a microservice backing a scholar search engine frontend (called Pan).

2 The Protobase Document Store

Protobase[5] is our open-source document store that executes backend code for analytic applications in a sandbox inside the database system process (see Figure 1 for its architecture).

Overview Backend code is written in Python, exposes a REST-based user-defined service, and potentially accesses Protobases query engine and storage engine directly. Backend code is executed by the runtime engine in Protobase that handles HTTP requests, invokes user-defined code and manages delegation between components. However, backend code is primarily used for the user-defined microservice but may be intermixed with pre-defined components (such as Protobases control panel) or other components. Underneath, the storage engine operates on a memory resistant database loaded once from a Cabin-formatted database image. To take advantage of modern processors asynchronous capabilities, the entire architecture stack is supported by our parallelization framework.

Instances The interaction between a backend and our database system does not require inter-process communication. This is a good fit for microservice prototypes due to (a) data marshalling can be kept to a minimum, and (b) less systems are required to get things done. In a sense, a service backed by Protobase (an *instance*) acts as a specialized database system exposing a user-defined, domain-specific interface that frontends interact with.

Cabin Files Our data model is our novel *strict document* data model which is similar to the JSON data model (cf. [BRSV17] for a basic theoretical framework) but with two restrictions: (i) all property values of type array must contain elements of the same type, and (ii) arrays of arrays are invalid. Both (i) and (ii) allow us to efficiently recursively transform any strict document into a *columnar document* re-written to organize its data in a columnar fashion. In detail, a columnar document (a) stores keys and values in two separate columns grouped by value type, (b) aggressively performs *null* compression, and (c) converts properties mapping to arrays of objects to an equivalent (sparse) columnar table. Multiple documents are bundled, binary-encoded and lightweight compressed into a *Columnar Binary-Encoded JSON* (Cabin) file[6].

[5] Protobase code repository: `https://github.com/protolabs/protobase`

[6] Cabin file format specification: `http://www.cabinspec.org`

3 Demonstration Outline

In our demo, we show an example of (fullstack) prototyping of a (tiny) scientific research search engine (called Pan) using Protobase and the Microsoft Academic Graph[7] as dataset.

Pan is a web app rendering information fetched from a microservice. For ease of simplicity, Pan offers four features: (1) displaying publication information given a paper title, (2) listing publications within a certain time span given a specific author's name, (3) displaying quantitative information, and (4) lists publications via the *is-cited-by* relationship. Finally, we implement a microservice in Protobase to provide a backend for Pan.

Outline The outline for our demonstration on working with Protobase is as follows:

(i) A brief introduction on fundamentals of Protobase: the systems architecture and instance model (incl. structure, interaction user code and system, and database I/O).

(ii) Showcasing the database meta information provided by Protobase to get informed on the dataset at hand (e.g., property names, type conflicts, or schema information).

(iii) Finally, we show several concepts and some code snippets to outline how to query, aggregate, and how to use the built-in cache and indexing mechanism in Protobase.

The audience is invited to play around with Pan and Protobase.

References

[BRSV17] Pierre Bourhis, Juan L Reutter, Fernando Suárez, and Domagoj Vrgoč. JSON: Data Model, Query Languages and Schema Specification. In *Proceedings of the Symposium on Principles of Database Systems (ACM PODS)*, pages 123–135, 2017.

[CJP+18] Gabriel Campero, Anusha Janardhana, Marcus Pinnecke, Yusra Shakeel, Jacob Krüger, Thomas Leich, and Gunter Saake. Exploring Large Scholarly Networks with Hermes. In *International Conference on Extending Database Technology (EDBT)*, pages 650–653, 2018.

[LHB13] Harold Lim, Yuzhang Han, and Shivnath Babu. How to Fit when No One Size Fits. In *Conference on Innovative Data Systems Research (CIDR), Online Proceedings*, 2013.

[LKC+17] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A Fast JSON Parser for Data Analytics. *Proceedings of the VLDB Endowment (VLDB)*, pages 1118–1129, 2017.

[WZS+18] JunPing Wang, WenSheng Zhang, YouKang Shi, ShiHui Duan, and Jin Liu. Industrial Big Data Analytics: Challenges, Methodologies, and Applications. *arXiv preprint arXiv:1807.01016, Online Proceedings*, 2018.

[7] Microsoft Academic Graph as part of the Open Academic Graph: `https://aminer.org/open-academic-graph`