Building a Polyglot Data Access Layer for a Low-Code Application Development Platform*
(Experience Report)

Ana Nunes Alonso[1], João Abreu[2], David Nunes[2], André Vieira[2], Luiz Santos[2], Tércio Soares[2], and José Pereira[1]

[1] INESC TEC and U. Minho
ana.n.alonso@inesctec.pt,jop@di.uminho.pt
[2] OutSystems
{joao.abreu,david.nunes,andre.vieira,luiz.santos,tercio.soares}@outsystems.com

Abstract. Low-code application development as proposed by the OutSystems Platform enables fast mobile and desktop application development and deployment. It hinges on visual development of the interface and business logic but also on easy integration with data stores and services while delivering robust applications that scale.
Data integration increasingly means accessing a variety of NoSQL stores. Unfortunately, the diversity of data and processing models, that make them useful in the first place, is difficult to reconcile with the simplification of abstractions exposed to developers in a low-code platform. Moreover, NoSQL data stores also rely on a variety of general purpose and custom scripting languages as their main interfaces.
In this paper we report on building a polyglot data access layer for the OutSystems Platform that uses SQL with optional embedded script snippets to bridge the gap between low-code and full access to NoSQL stores.

1 Introduction

The current standard for integrating NoSQL stores with available low-code platforms is for developers to manually define how the available data must be imported and consumed by the platform, requiring expertise in each particular NoSQL store, especially if performance is a concern. Enabling the seamless integration of a multitude of NoSQL stores with the OutSystems platform will offer its more than 200 000 developers a considerable competitive advantage over other currently available low-code offers.

Main challenges include NoSQL systems not having a standardized data model, a standard method to query meta-data, or in many cases, by not enforcing a schema at all. Second, the value added by NoSQL data stores rests precisely on a diversity of query operations and query composition mechanisms, that exploit specific data models, storage, and indexing structures. Exposing these as

* This work was supported by Lisboa2020, Compete2020 and FEDER through Project RADicalize (LISBOA-01-0247-FEDER-017116 | POCI-01-0247-FEDER-017116).

visual abstractions for manipulation risks polluting the low-code platform with multiple particular and overlapping concepts, instead of general purpose abstractions. On the other hand, if we expose the minimal common factor between all NoSQL data stores, we are likely to end up with minimal filtering capabilities that prevent developers from fully exploiting NoSQL integration. In either case, some NoSQL data stores offer only very minimal query processing capabilities and thus force client applications to code all other data manipulation operations, which also conflicts with the low-code approach. Finally, ensuring that performance is compatible with interactive applications means that one cannot resort to built-in MapReduce to cope with missing query functionality, as it leads to high latency and resource usage. Also, coping with large scale data sets means avoiding full data traversals by exposing relevant indexing mechanisms and resorting to approximate and incomplete data, for instance, when displaying a developer preview.

In this paper we summarize our work on a proof-of-concept polyglot data access layer for the OutSystems Platform that addresses these challenges, thus making the following contributions:

- We propose to use a polyglot query engine, based on extended relational data and query models, with embedded NoSQL query script fragments as the approach that reconciles the expectation of low-code integration with the reality of NoSQL diversity.
- We describe a proof-of-concept implementation that leverages an off-the-shelf SQL query engine that implements the SQL/MED standard [4].

As a result, we describe various lessons learned, that are relevant to the integration of NoSQL data stores with low-code tools in general, to how NoSQL data stores can evolve to make this integration easier and more effective, and to research and development in polyglot query processing systems in general. An extended version of this work is available in [1].

The rest of the paper is structured as follows. Section 2 describes our proposal to integrate NoSQL data stores in the OutSystems platform, including our current proof-of-concept implementation. Section 3 concludes the paper by discussing the main lessons learned.

2 Architecture

Our proposal is based on two main criteria. First, how it contributes to the vision of NoSQL data integration in the low-code platform outlined in Section 1 and how it fits the low-code approach in general. Second, the talent and effort needed for developing such integrations and then, later, for each additional NoSQL system that needs to be supported.

We can consider two extreme views. On the one hand, we can enrich the abstractions that are exposed to the developer to encompass the data and query processing models. This includes: data types and structures, such as nested tuples, arrays, and maps; query operations, ranging from general purpose data manipulation (e.g., flattening a nested structure) to domain-specific operations

(e.g., regarding search terms in a text index); and finally, where applicable, query composition (e.g., with MapReduce or a pipeline).

This approach has however several drawbacks. First, it pollutes the low-code platform with a variety of abstractions that have to be learned by the developers to fully use it. Moreover, these abstractions change with support for additional NoSQL systems and are not universally applicable. In fact, support for different NoSQL systems would be very different, making it difficult to use the same know-how to develop applications on them all. Finally, building and maintaining the platform itself would require a lot of talent and effort in the long term, as support for additional systems could not be neatly separated in plugins with simple, abstract interfaces.

On the other hand, we can map all data in different NoSQL systems to a relational schema with standard types and allow queries to be expressed in SQL. This results in a mediator/wrapper architecture that allows the same queries to be executed over all data regardless of its source, even if by the query engine at the mediator layers.

This approach also has drawbacks. First, mapping NoSQL data models to a relational schema requires developer intervention to extract the view that is adequate to the queries that are foreseen. This will most likely require NoSQL-specific talent to write target queries and conversion scripts. Moreover, query capabilities in NoSQL systems will remain largely unused, as only simple filters and projections are pushed down, meaning the bulk of data processing would need to be performed client-side.

Our proposal is a compromise between these two extreme approaches, that can be summed up as: support for nested data and its manipulation in the abstractions shown to the low-code developer, along with the ability to push aggregation operations down to NoSQL stores from a mediator query engine, will account for the vast majority of use cases. In addition, the ability to embed native query fragments in queries will allow fully using the NoSQL store when talent is available, without disrupting the overall integration. The result is a polyglot query engine, where SQL statements are combined with multiple foreign languages for different NoSQL systems.

The proposed architecture is summarized in Figure 1, highlighting the proposed NoSQL data access layer. To the existing OutSystems platform, encompassing development tools and runtime components, we add a new *Polyglot connector*, using the Database Integration API to connect to the *Polyglot Query Engine (QE)* through standard platform APIs. The Polyglot QE acts as a mediator. It exposes an extended relational database schema for connected NoSQL stores and is able to handle SQL and polyglot queries.

For each NoSQL Store, there is a *Wrapper*, composed of three sub-components: *metadata* extraction, responsible for determining the structure of data in the corresponding store using an appropriate method and mapping it to the extended SQL data model of the Polyglot QE; a query *push-down* component, able to translate a subset of SQL query expressions, to relay native query fragments, or produce a combination of both in a store-specific way; and finally, the *cursor*,

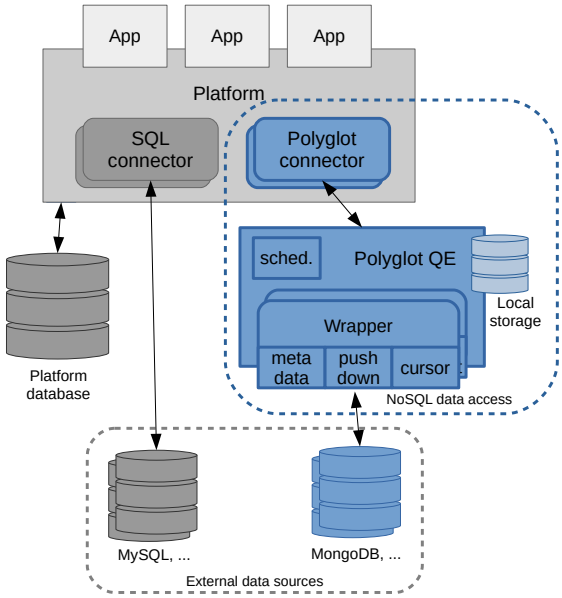

Fig. 1: Architecture overview

able to iterate on result data and to translate and convert it as required to fit the common extended SQL data model.

The Polyglot QE makes use of *Local storage* for the configuration of NoSQL store adapters and for holding materialized views of data to improve response times. The *Job Scheduler* enables periodically refreshing materialized views by re-executing their corresponding queries.

2.1 Implementation

We base our proof-of-concept implementation on open source components. The main component to select is the SQL query engine used as the mediator. Besides its features as a query engine, we focus on: the availability of wrappers for different NoSQL systems and the talent needed to implement additional features; the compatibility of the open source license with commercial distribution; the maturity of the code-base and supporting open source community; and finally, on its compatibility with the OutSystems low-code platform. We consider two options.

PostgreSQL with FDW[7]. It is an option as it supports foreign data wrappers according to the SQL/MED standard (ISO/IEC 9075-9:2008). The main attractive for PostgreSQL is that it is a very mature open source product, with a business friendly license, a long history of deployment in production, and an unparalleled developer and user community. There is also support for .NET and Java client application platforms. In terms of features, PostgreSQL provides

a robust optimizer and an efficient query engine, that has recently added parallel execution, with excellent support for SQL standards and multiple useful extensions. It supports nested data structures both with the `json`/`jsonb` data types, as well as by natively supporting arrays and composite types. It has extensive support for traversing and unnesting them. Regarding support for foreign data sources, besides simple filters and projections, the PostgreSQL Foreign Data Wrapper (FDW) interface can interact with the optimizer to push down joins and post-join operations such as aggregations. With PostgreSQL FDW, it is possible to declare tables for which query and manipulation operations are delegated on adapters. The wrapper interface includes the ability to either impose or import a schema for the foreign tables. Imposing a schema requires the user to declare data types and structure and it is up to the wrapper to make it fit by using automatic type conversions as possible. If this automatic process is not successful the user will need to change the specified data type to provide a closer type match. The wrapper can also (optionally) advertise the possibility of importing a schema. In this case, the user simply instructs PostgreSQL to import meta-data from the wrapper and use it for further operations. This capability is provided by the wrapper and currently, this is only supported for SQL databases, for which the schema can be easily queried. Furthermore, PostgreSQL FDW can export the schema of the created foreign tables. In addition to already existing wrappers for many NoSQL data sources, with variable features and maturity, the Multicorn[3] framework allows exposing the Python scripting language to the developer, to complement SQL and express NoSQL data manipulation operations.

In terms of our goals, PostgreSQL falls short on automatically using existing materialized views in queries. The common workaround is to design queries based on views and later decide whether to materialize them, which is usable in our scenario. Another issue is that schema inference is currently offered for relational data sources only. The workaround is for the developer to explicitly provide the foreign table definition.

Calcite[2] (in Dremio OSS[3]). The Calcite SQL compiler, featuring an extensible optimizer, is used in a variety of modern data processing systems. We focus on Dremio OSS as its feature list most closely matches our goal. Calcite is designed from scratch for data integration and focuses on the ability to use the optimizer itself to translate parts of the query plan to different back end languages and APIs. It also supports nested data types and corresponding operators. Dremio OSS performs schema inference, but treats nested structures as opaque and, therefore, does not completely support low-code construction of unnesting operations, in the sense that the user still needs to explicitly handle these. Still, it provides the ability to impose a table schema ad-hoc or flexibly adapt data types which is a desirable feature for overriding incorrect schema inference. Also, Dremio OSS adds a distributed parallel execution engine, based on the Arrow columnar format, and a convenient way to manage materialized views (a.k.a., "reflections"), that are automatically used in queries. Unfortunately, one

[3] https://github.com/Kozea/Multicorn

cannot define or use indexes on theses views, which reduces their usefulness in our target application scenarios.

Although Calcite has a growing user and developer community, its maturity is still far behind PostgreSQL. The variety of adapters for different NoSQL systems is also lagging behind PostgreSQL FDW, although some are highly developed. For instance, the MongoDB adapter in Dremio OSS is able to extensively translate SQL queries to MongoDB's aggregation pipeline syntax, thus being able to push down much of the computation and reduce data transfer. The talent and effort needed for exploiting this in additional data wrappers is, however, substantial. Both for Dremio and PostgreSQL, limitations in schema imposition/inference do not impact querying capabilities, only the required talent to use the system. For PostgreSQL FDW, this can be mitigated by extending adapters to improve support for nested data structures, integrating schema inference/extraction techniques. Finally, the main drawback of this option is that, as we observed in preliminary tests, resource usage and response time for simple queries is much higher than for PostgreSQL.

Choosing PostgreSQL with FDW. In the end, we found that our focus on interactive operational applications and the maturity of the PostgreSQL option, outweigh, for now, the potential advantages from Calcite's extensibility.

Additional development Completing a proof-of-concept implementation based on PostgreSQL as a mediator requires additional development in the low-code platform itself, an external database connector, and in the wrappers. As examples, we describe support for two NoSQL systems. The first is Cassandra, a distributed key-value store that has evolved to include a typed schema and secondary indexes. It has, however, only minimal ad-hoc query processing capabilities, restricted to filtering and projection. The second is MongoDB, a schema-less document store that has evolved to support complex query processing with either MapReduce or the aggregation pipeline. Both are also widely used in a variety of applications.

Schema conversion. In order to support relational schema introspection, we reuse `mongodb-schema`[4], extending it to provide a probabilistic schema, with fields and types, for each collection in a MongoDB database. Top-level document fields are mapped as table attributes. When based on probabilistic schemas, all discovered attributes are included, leaving it to the user/developer to decide which attributes to consider. Nested documents' fields are mapped as top-level attributes, named as the field prefixed with its original path. Nested arrays are handled by creating a new table and promoting fields of inner documents to top-level attributes. Documents from a given collection become a line of the corresponding table (or tables). An alternative would be to create a denormalized table. Notice that this is equivalent to the result of a natural join between the corresponding separate tables. However, separate tables fit better what would be expected from a relational database and thus improve the low-code experience. It should be pointed out that viewing the original collection as a set of separate relational tables has no impact on the performance of a query with a

[4] `https://github.com/mongodb-js/mongodb-schema`

join between these tables. The required unnesting directives, using the `$unwind` pipeline aggregation operator are also generated and added to the table definition. We also provide the option, on by default, of adding a column referencing the `_id` of the outermost table to all inner tables on schema generation, that can serve as an elementary foreign key.

MongoDB wrapper. There are multiple FDW implementations for MongoDB. We selected one based on Multicorn,[5] for ease of prototyping, and change it extensively to include schema introspection and, taking advantage of aggregation pipeline query syntax, to allow push-down to work with user supplied queries. This is greatly eased by MongoDB's syntax for the aggregation pipeline being easily manipulated by programs, by adding additional stages.

Cassandra wrapper. We also use a wrapper based on Multicorn.[6]. In this case, we add the ability to use arbitrary Python expressions to compute row keys from arbitrary attributes, as in earlier versions of Cassandra it was usual to manually concatenate several columns. Even if this is no longer necessary in recent versions of Cassandra, it is still common practice in other NoSQL systems such as HBase. The currently preferred interface to Cassandra, CQL, is not the best fit for being manipulated by programs, although, being so simple, it can be done with relatively small amount of text parsing.

Connectors. We implemented custom connectors for each NoSQL store based on the original PostgreSQL connector. This allows the developer to directly pick the target data store from the platform's visual development environment [6] drop-down menu and provide system specific connection options. It also allows system specific projection and aggregation operators to be handled.

Developer platform. The changes needed in the platform to fully accommodate the integration are the ability to express nesting and unnesting operators in the data manipulation UI, and to generate SQL queries that contain them when using the NoSQL integration connectors. It is, however, possible to workaround this by configuring multiple flattened views of data, as needed, when the schema is introspected and imported.

3 Lessons Learned

We discussed the challenges in integrating a variety of NoSQL data stores with the OutSystems low-code platform. This is achieved by a SQL query engine that federates multiple NoSQL sources and complements their functionality, using PostgreSQL with Foreign Data Wrappers as a proof-of-concept implementation. It allowed us to learn some lessons about NoSQL systems and to propose a good trade-off between integration transparency and the ability to take full advantage of each systems' particularities. Lessons target low-code platform providers (**1**,**2**), polyglot developers (**3**,**4**,**5**,**6**) and NoSQL data store providers (**7**,**8**).

1. Target an extended relational model. The relational data model when extended with nested composite data types such as maps and arrays can success-

[5] `https://github.com/asya999/yam_fdw`
[6] `https://github.com/rankactive/cassandra-fdw`

fully map the large majority of NoSQL data models with minimal conversion or conceptual overhead. Moreover, when combined with flatten and unflatten operators, the relational query model can actually operate on such data and represent a large share of target query operations. This is very relevant, as it provides a small set of additional concepts that have to be added to the low-code platform or, preferably, none at all as unnesting is done when importing the schema.

2. A query engine is needed. Due to the varying nature of query capabilities in different data sources, a query engine that can perform various computations is necessary to avoid that developers have to constantly mind these differences. This is true even for querying a single source at a time.

3. Basic schema discovery with overrides is needed. Although CloudMdsQl [5] has shown that it is possible to build a polyglot query engine without schema discovery, by imposing ad-hoc schemas on native queries, it severely restricts its usefulness in the context of a low-code platform. However, after getting started with automatically inferred schema, it is useful to allow the developer to impose additional structure such as composite primary keys in key value stores.

4. Embedded scripting is required. Although many data manipulation operations could be done in SQL at the query engine, embedding snippets of a general purpose scripting language allows direct reuse of existing code and reduces the need for talent to translate them. Together with the ability to override automatic discovery, this is key to ensuring that the developer never hits a wall imposed by the platform.

5. Materialized view substitution is desirable. Although our proof-of-concept implementation does not include it, this is the main feature from the Calcite-based alternative that is missing. The ability to define different native queries as materializations of various sub-queries is the best way to encapsulate alternative access paths encoded in a data-store specific language.

6. Combining foreign tables with scripting is surprisingly effective. Although CloudMdsQl [5] proposed its own query engine, a standard SQL engine with federated query capabilities, when combined with a scripting layer for developing wrappers such as Multicorn, is surprisingly effective in expressing queries and supporting optimizations.

7. A NoSQL query interface should be targeted at machines, not only at humans. NoSQL systems such as MongoDB or Elasticsearch, that expose a query model based on an operator pipeline, are very friendly to integration as proposed. In detail, it allows generating native queries from SQL operators or to combine partially hand-written code with generated code. Ironically, systems that expose a simplistic SQL like language that is supposed to be more developer friendly, such as Cassandra, make it harder to integrate as queries in these languages are not as easily composed.

8. Focus on combining query fragments. It might be tempting to overlook some optimizations that are irrelevant when a human is writing a complete query, e.g., as pushing down `$match` in a MongoDB pipeline. However, these optimizations are fairly easy to achieve and greatly simplify combining partially machine generated queries with developer written queries.

References

1. Alonso, A.N., Abreu, J., Nunes, D., Vieira, A., Santos, L., Soares, T., Pereira, J.: Towards a polyglot data access layer for a low-code application development platform. https://arxiv.org/abs/2004.13495 (2020)
2. Apache Calcite – Documentation. https://calcite.apache.org/docs/, accessed: 2020-02-27
3. Dremio – The data lake engine. https://docs.dremio.com, accessed: 2020-02-27
4. ISO/IEC: Information technology - Database languages - SQL - Part 9: Management of External Data (SQL/MED). ISO/IEC standard (2016)
5. Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R., Pereira, J.: CloudMdsQL: querying heterogeneous cloud data stores with a common language. Distributed and Parallel Databases pp. 1–41 (2015). https://doi.org/10.1007/s10619-015-7185-y
6. Development and deployment environments. https://www.outsystems.com/evaluation-guide/outsystems-tools-and-components/#1, accessed: 2020-03-02
7. PostgreSQL foreign data wrappers. https://wiki.postgresql.org/wiki/Foreign_data_wrappers, accessed: 2020-02-27