Query-based Schema Evolution Recommendations for Hybrid Polystores

Benats, Pol; Meurice, Loup; Cleve, Anthony

Published in:
Proceedings of the 41st International Conference on Conceptual Modeling (ER 2022), Forum track

Publication date:
2022

Document Version
Version created as part of publication process; publisher's layout; not normally made publicly available

Link to publication

Citation for pulished version (HARVARD):
Benats, P, Meurice, L & Cleve, A 2022, Query-based Schema Evolution Recommendations for Hybrid Polystores. in *Proceedings of the 41st International Conference on Conceptual Modeling (ER 2022), Forum track.*

Query-based Schema Evolution Recommendations for Hybrid Polystores

Pol Benats, Loup Meurice, Maxime Gobert and Anthony Cleve

Namur Digital Institute, University of Namur, 21 rue Grandgagnage, Namur, 5000, Belgium

Abstract

This paper presents a approach supporting the continuous evolution of *hybrid polystores*, i.e., software systems relying on the joint manipulation of several heterogeneous databases (relational, NoSQL). This approach monitors and analyzes polystore data usage, with a particular focus on query performance, in order to recommend polystore schema evolutions when relevant. The approach relies on TyphonML, a modeling language allowing users to express the conceptual schema of the polystore and to map it to diverse native backends. The approach takes as input (1) the current conceptual schema of the polystore and its mapping to underlying native databases; (2) the history of queries that have been executed on the polystore as well as their duration and (3) the (evolving) size of each polystore entity. The suggested schema evolution recommendations include, among others, the creation of indexes, the merging of conceptual entities, and the migration of entities from a native backend to another.

Keywords

software systems, heterogeneous databases, query performance, recommendations, evolution

1. Introduction

Schema evolution is a complex and risky process that has been subject to a large research literature [1]. Most existing approaches and tools supporting schema evolution consider systems relying on a *single* database, most often a *relational* database. Among the most comprehensive tools, PRISM++ supports the database evolution process by evaluating the impact of schema modifications on queries and on data. It then helps developers with the rewriting of historical queries and the migration of related data, thereby reducing the downtime of the system by reducing manual effort. Complementary approaches make *recommendations* to the developers on how to apply schema evolution operations to their system [2, 3].

More recent approaches support the evolution of systems manipulating NoSQL datastores. For instance, ControVol [4] is a framework capable of warning developers of risky cases of mismatched data and (implicit) schema. It also suggests and performs automatic fixes to resolve possible schema migration problems.

In this paper, we present a query-based recommendation approach, designed for *hybrid polystores*, i.e., systems made of *several* possibly heterogeneous relational *and* NoSQL databases. The approach captures the queries sent to the polystore, analyses their performance and provides polystore schema change recommendations for improving the performance of particular queries.

ER 2022: 41st international conference on conceptual modeling, October 17–20, 2022, Online (virtual)

pol.benats@unamur.be (P. Benats); loup.meurice@unamur.be (L. Meurice); maxime.gobert@unamur.be (M. Gobert); anthony.cleve@unamur.be (A. Cleve)

Our approach is based on the tools of the Typhon project[1] proposing polyglot and hybrid persistence architectures for big data analytics on relational and NoSQL databases. At the time of writing this paper, the database technologies covered in the project were MariaDB, MongoDB, Redis, Cassandra and Neo4j.

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 presents the approach in detail, based on a running example. Section 4 gives concluding remarks.

2. Related work

Recent studies [5, 6] revealed the emergence of more complex data-intensive systems, called *hybrid polystores*. Those modern architectures rely on *several* heterogeneous, possibly overlapping databases and on the combination of relational and NoSQL technologies. While understanding and evolving the database schemas of a hybrid polystore can be time-consuming and complex, there is still a lack of automated support to assist developers in this task.

Evolution

Database-dependent systems evolution and NoSQL and hybrid polystores schema evolution are complex processes. Most relational-based existing tool-supported approaches to this process rely on transformational techniques, generative techniques or a combination of both. For instance, several authors attempt to contain the ripple effect of changes to the database schema, e.g., by generating wrappers [7] views or APIs that provide/enable backward compatibility and by transforming the programs in order to interface them with those intermediate layers.

Scherzinger *et al.* [8] investigate schema evolution for feature-rich, interactive web applications that are backed by NoSQL data stores, for document and extensible record stores in particular. They propose a generic schema evolution interface to NoSQL systems. Such a tool is intended for developers, administrators, and software architects to declaratively manage the structure of their production data. The authors investigate the established field of schema evolution in the new context of schema-less NoSQL data stores, contribute to a declarative NoSQL schema evolution language, introduce a generic NoSQL database programming language that abstracts from the APIs of the most prominent NoSQL systems, implement schema evolution operations in in their NoSQL database programming language, investigate whether a proposed schema evolution operation is safe to execute, and apart from exploring eager migration, they introduce the notion of lazy migration and point out its potential for future research in the database community.

De Lima *et al.* [9] propose an approach for the conversion of conceptual schemas into NoSQL document logical schemas. It starts with a conceptual schema and workload information given by the application designer, as shown in Figure 1. The workload information is estimated over a conceptual schema, being also used as input for the Logical Design phase in order to generate appropriate logical structures. The mapping of the conceptual schema to a NoSQL document logical schema is governed by a set of rules that converts each conceptual constructor to an equivalent representation in the NoSQL document logical model. The logical model is an abstract model to represent NoSQL document implementation models. In the Implementation

[1]https://github.com/typhon-project

Design phase, a NoSQL document logical schema is translated to the common implementation model for NoSQL documents, i.e., the JSON specification. Even though the Implementation Design level is considered by the approach, the paper focuses on generating optimized NoSQL document structures from a conceptual schema in the Logical Design level.

Recommendation

A few works propose NoSQL and hybrid polystores schema evolution through recommandations. Mior *et al.* [10] present a system for recommending database schemas for NoSQL applications. The cost-based approach uses a novel binary integer programming formulation to guide the mapping from the application's conceptual data model to a database schema. They implemented a prototype of this approach for the Cassandra extensible record store. The prototype, the NoSQL Schema Evaluator (NoSE) is able to capture rules of thumb used by expert designers without explicitly encoding the rules. Given a conceptual model (optionally with statistics describing data distribution), an application workload, and an optional space constraint, the schema design problem is to recommend a schema such that (a) each query in the workload is answerable using one or more get requests to column families in the schema, (b) the weighted total cost of answering the queries is minimized, and optionally (c) the aggregate size of the recommended column families is within a given space constraint. Solving this optimization problem is the objective of the schema advisor. In addition to the schema, for each query in the workload, NoSE recommends a specific plan for obtaining an answer to that query using the recommended schema.

Dynamic program analysis

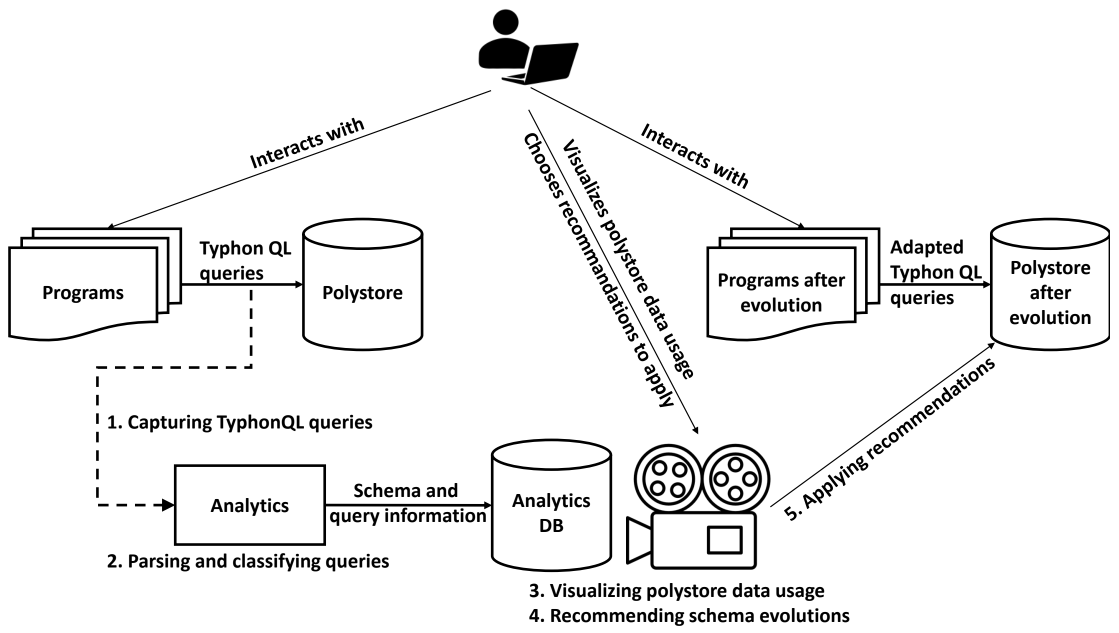
Several approaches have been proposed for capturing and analysing database queries at runtime [11, 12, 13, 14]. In contrast with our work, they all consider applications relying on a single, relational database. Their focus is on recovering implicit knowledge about the database schema.

To the best of our knowledge recommending database schema evolution is not common in research, even less when considering systems using SQL and NoSQL databases in combination. Our approach helps in recommanding polystore schema evolutions with a particular focus on query performance.

3. Approach

Our approach aims to monitor data usage performance in a Typhon polystore in order to provide users with schema evolution recommendations, when relevant. It communicates with several polystore components to retrieve the polystore schema expressed in TyphonML [15] and the polystore databases. This allows our approach to automatically retrieve useful information about the polystore, including: (1) the polystore configuration, i.e., the TyphonML entities and their mapping to underlying native databases; (2) the queries (expressed in TyphonQL [16]) that are executed on the polystore, and their duration; and (3) the (evolving) size of the TyphonML entities over time.

Figure 1: Overview of the approach


Interacts with
Visualizes polystore data usage
Chooses recommandations to apply
Interacts with
Programs
Typhon QL queries
Polystore
Programs after evolution
Adapted Typhon QL queries
Polystore after evolution
1. Capturing TyphonQL queries
Analytics
Schema and query information
Analytics DB
2. Parsing and classifying queries
5. Applying recommendations
3. Visualizing polystore data usage
4. Recommending schema evolutions

3.1. General overview

The general architecture of the approach is depicted in Figure 1. Each time a query is sent to the polystore the Anaytics module captures it, and keeps it only if it corresponds to a DML query execution. The Analytics module latter parses, analyses and classifies the query. This information is stored in the Analytics database, that is used as input for polystore data-usage visualization. The approach also proposes performance-based schema reconfiguration recommendations based on the same information. Each of these steps is described in further details in the remaining of this section.

3.2. Running example

In order to illustrate the approach, we will use as running example a polystore structured according to the simplified TyphonML schema given in Figure 2. The schema presents 2 conceptual entities (i.e. *Employees* and *EmployeeAddress*) mapped to 2 physical databases (respectively MariaDB and MongoDB). Based on this example schema, we automatically generated TyphonQL queries each query having a fictitious execution time. We also considered a fictitious history of the polystore entities, especially, as far as their size is concerned.

3.3. Step 1: Capturing polystore queries

The approach exploits the polystore monitoring mechanisms. Thanks to those mechanisms, we can capture at runtime the successive TyphonQL queries that are sent to the polystore, and

Figure 2: Polystore TyphonML model example

```
entity Employees {
 lastName: string[20]
 firstName: string[10]
 title: string[30]
 birthDate: datetime
 phone: string[24]
 photoPath: string[255]
 address -> EmployeeAddress[1]
}

entity EmployeeAddress {
 address : string[60]
 city: string[15]
 region: string[15]
 postalCode: string[10]
 country: string[15]
 employee -> Employees[1]
}
```

```
relationaldb relDatabase {
 tables {
  table {
   EmployeesTable : Employees
   idSpec ('Employees.firstName')
  }
 }

documentdb docDatabase {
 collections{
  EmployeeAddressCollection : EmployeeAddress
 }
}
```

executed by the TyphonQL engine. To do so, we consume and analyse the TyphonQL queries, generated and pushed by the TyphonQL engine to the Analytics module.

3.4. Step 2: Parsing and classifying queries

The queries at Step 1 include the TyphonQL queries that have been executed by the TyphonQL engine. The Analytics module parses each of those queries, in order to extract relevant information to be used during the analytics and recommendation phases. Our tool focuses on DML queries, i.e., select, insert, delete, and update queries. It ignores other events such as, for instance, the execution of DDL queries (e.g., create entity, delete entity, ...) sent to the TyphonQL engine. The tool parses each captured TyphonQL query in order to extract relevant information, including:

- the type of query (select, insert, delete, update);
- the accessed TyphonML entities;
- the join conditions, if any;
- the query execution time, expressed in milliseconds.

Once the query is parsed and analyzed, the approach also classifies it. This classification aims to group together all TyphonQL queries of the same form. A group of TyphonQL queries is called a query category. The queries belonging to the same query category are queries that would become the same query after replacing all input values with place-holders. For instance, the following three TyphonQL queries can be classified into the same query category:

```
from EmployeeAddress a select a where a.country == ”Italy”
from EmployeeAddress a select a where a.country == ”Belgium”
from EmployeeAddress a select a where a.country == ”Germany”
```

Indeed, those queries only differ in terms of their input values. When replacing the only input value corresponding to the address country (”Italy”, ”Belgium” and ”Germany”, respectively) with a place-holder (”?”), we obtain the following query category:

```
from EmployeeAddress a select a where a.country == ”?”
```

In addition to parsing, analyzing and classifying the queries executed by the TyphonQL engine, the approach also extracts - at regular time intervals - information about the Typhon polystore, with a particular focus on TyphonML entities. This includes, in particular, the size of each TyphonML entity, expressed in terms of number of records, e.g., number of rows for a relational table or number of documents for a MongoDB collection. The extracted information is stored in an internal database, that we will call the *Analytics Database* in the remaining of this document. The structure of this database is shown in Figure 3.

The QLQuery entity is at the core of the *Analytics Database*. It corresponds to the information stored for each TyphonQL DML query, while QLNormalizedQuery only contains the corresponding query categories. In our running example, we started from 1007 generated query events (QLQuery instances), corresponding to 314 different query categories (QLNormalizedQuery instances). The TyphonModel collection relates to the successive versions of the TyphonML schema during the considered period. TyphonEntity and TyphonEntityHistory include information related to the polystore entities, *e.g.,* size and data manipulation usage, and their evolution history.

The analytics database populated during this step constitutes the main input of the next three steps.

3.5. Step 3: Visualizing polystore data usage

The implementation of the approach offers a web interface in order to visualize Typhon polystore analytics data. The main page of the application is depicted in Figure 4. It provides the user with a general overview of the polystore data usage at a coarse-grained level and the polystore configuration.

Polystore schema view

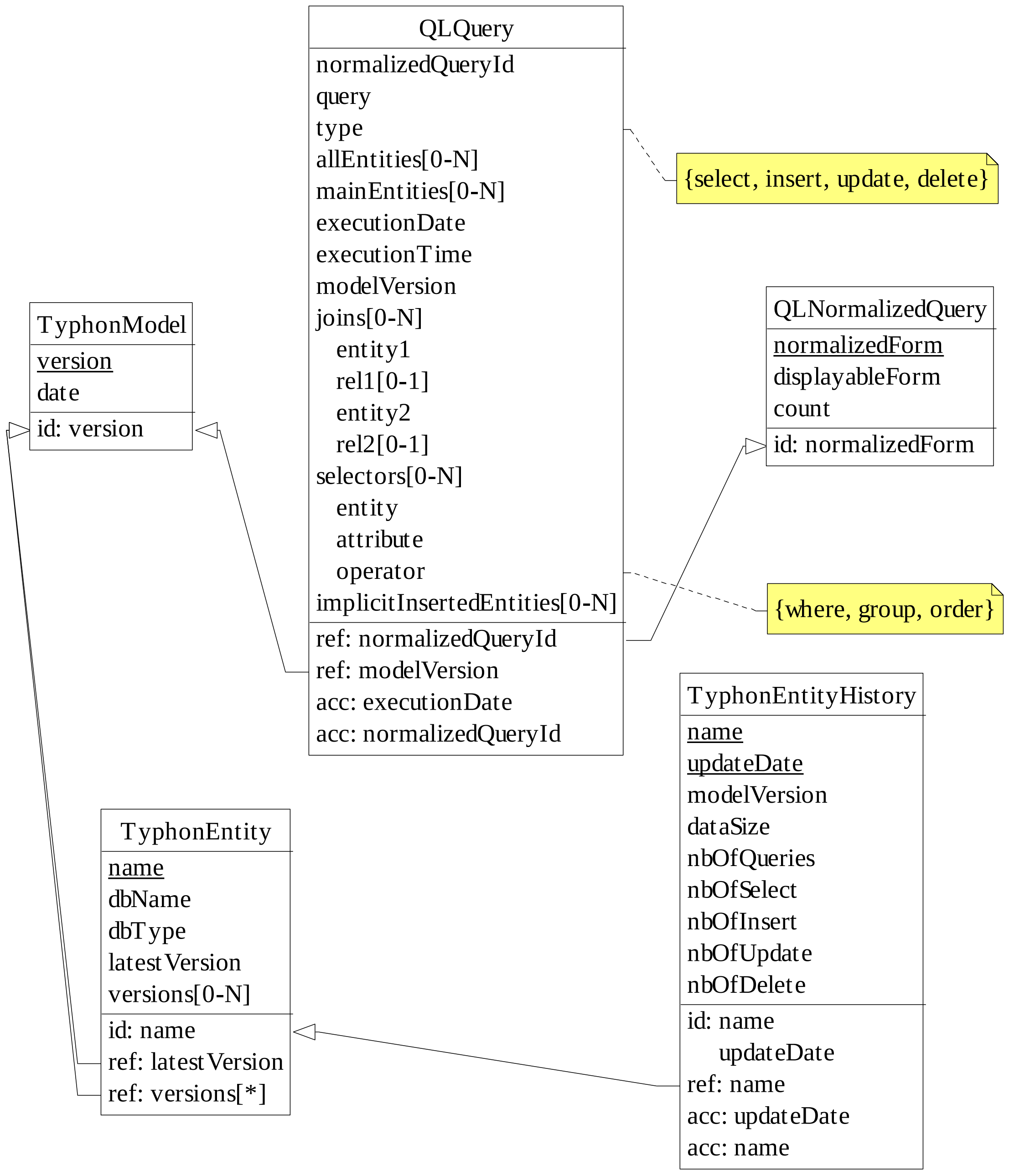
On the right-hand side of the main page, the user can see the current schema configuration of the polystore. In our example, the polystore currently consists of two databases: a relational database including 9 entities (tables), and a document database consisting of 3 entities (collections). The size of a circle relates to the number of records (rows, documents) in the corresponding database/entity. By clicking on a database (relational or document database), the user can zoom and get more details about the entities it includes.

Polystore entities view

The tool provides the user with a global overview of the current size of the polystore entities, as shown on the left-hand side of Figure 4. Positioning the mouse on a given entity would provide with the exact number of records for the entity. The evolution of the entity size over time is also provided. The user can select the entities of interest, and the tool then shows the evolution of the size of the selected entities over time in another tab.

Another visual metric concerns the volume and distribution of select, insert, delete and update operations that have been applied to the polystore during the considered period. Note that positioning the mouse pointer on a given operation allows to see the exact number of query occurrences of this type.

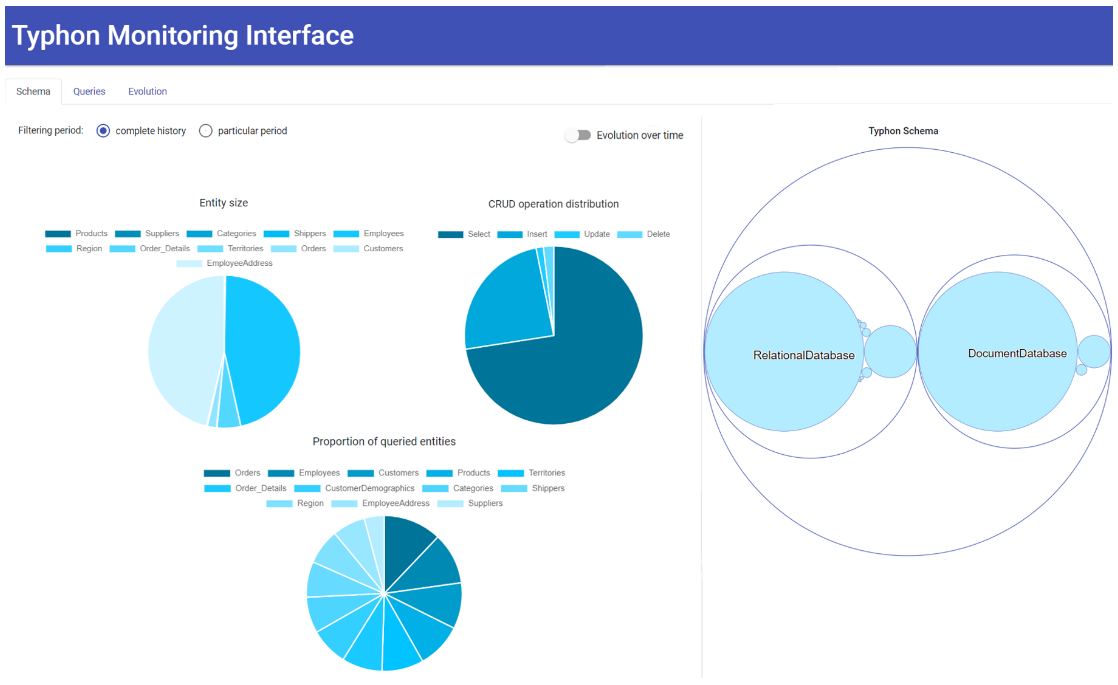
Figure 3: Structure of the Analytics Database.



Polystore CRUD operations view

A similar metric is provided for the distribution of CRUD (create, read, update and delete) operations by TyphonML entity. Again, positioning the mouse pointer on a given entity allows to see the exact number of queries involving this entity.

Figure 4: Schema tab of the web application



The distribution of executed queries (by CRUD operation or by entity) can also be shown for a particular period of time, chosen by the user, by checking the corresponding radio button at the top of the page.

The user can also look at the evolution of the number of CRUD operations executed over time in more details, at the level of the entire polystore, by clicking on the operations of interest. A similar visual analytics view is proposed at the level of polystore entities. One can see, for each entity, the evolution of the number of queries manipulating it over time. The user can either see the trend for all entities, or select the entity/entities of interest.

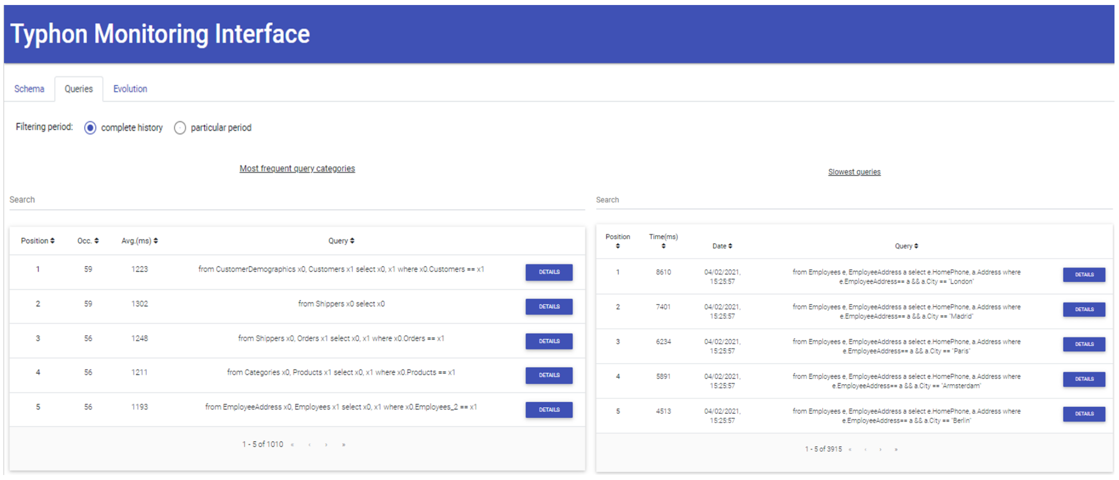
Polystore queries view

The user can then have a finer-grained look at the TyphonQL queries executed by the TyphonQL engine on the polystore. In the query tab, the tool provides the user with two searchable lists:

- the most frequent query categories, in decreasing order of number of occurrences (left-side of Figure 5);
- the slowest queries, in decreasing order of execution time (right-side of Figure 5).

Note that the user can also get the same lists, by considering a particular period of time. She can also search for particular queries using the search bar.

Figure 5: Query tab of the web application



Most frequent query categories

Position	Occ.	Avg.(ms)	Query	
1	59	1223	from CustomerDemographics x0, Customers x1 select x0, x1 where x0.Customers == x1	DETAILS
2	59	1302	from Shippers x0 select x0	DETAILS
3	56	1248	from Shippers x0, Orders x1 select x0, x1 where x0.Orders == x1	DETAILS
4	56	1211	from Categories x0, Products x1 select x0, x1 where x0.Products == x1	DETAILS
5	56	1193	from EmployeeAddress x0, Employees x1 select x0, x1 where x0.Employees_2 == x1	DETAILS

1 - 5 of 1010

Slowest queries

Position	Time(ms)	Date	Query	
1	8610	04/02/2021, 15:25:57	from Employees e, EmployeeAddress a select e.HomePhone, a.Address where e.EmployeeAddress== a && a.City == "London"	DETAILS
2	7401	04/02/2021, 15:25:57	from Employees e, EmployeeAddress a select e.HomePhone, a.Address where e.EmployeeAddress== a && a.City == "Madrid"	DETAILS
3	6234	04/02/2021, 15:25:57	from Employees e, EmployeeAddress a select e.HomePhone, a.Address where e.EmployeeAddress== a && a.City == "Paris"	DETAILS
4	5891	04/02/2021, 15:25:57	from Employees e, EmployeeAddress a select e.HomePhone, a.Address where e.EmployeeAddress== a && a.City == "Armsterdam"	DETAILS
5	4513	04/02/2021, 15:25:57	from Employees e, EmployeeAddress a select e.HomePhone, a.Address where e.EmployeeAddress== a && a.City == "Berlin"	DETAILS

1 - 5 of 3915

By looking at the query view, the user can figure out that the most frequent query category corresponds to the following TyphonQL query, that select the customers and their demographics information:

```
from CustomerDemographics x0, Customers x1 select x0, x1 where x0.Customers == x1
```

A total of 59 occurrences of this query category where executed, with an average execution time of 1223 milliseconds.

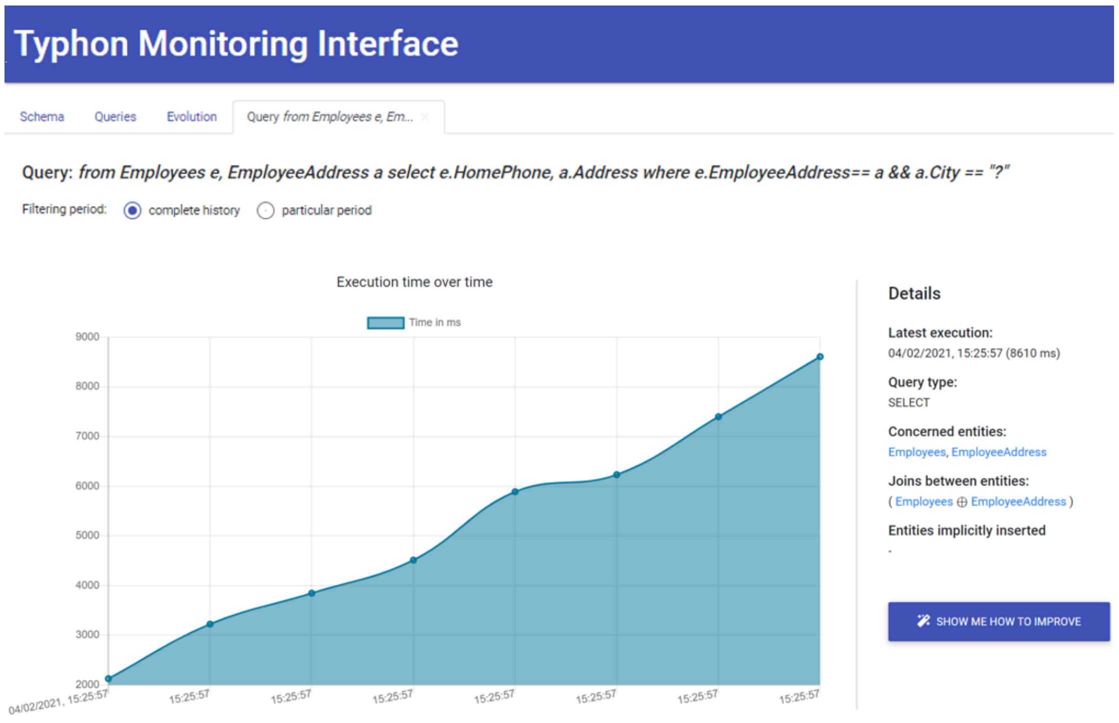
The following TyphonQL query was the slowest query during the entire considered period:

```
from Employees e, EmployeeAddress a select e.HomePhone, a.Address where e.EmployeeAddress== a && a.City == ”London”
```

3.6. Step 4: Recommending schema evolutions

The recommendation of schema evolutions based on polystore queries also depends on the underlying physical databases. Different sources [17, 18] explain the bad smells or anti-patterns to avoid when using and structuring a database, depending on the technology that implements it. For example, a TyphonQL query expressing a filtering condition on an entity attribute stored in the ”value” part of a key-value database (*e.g.,* Redis) will cause a full scan to be performed to retrieve all the keys that the database contains; the filter on the attribute will then be performed by the TyphonQL engine. In such a scenario, the choice of a key-value database could be questioned when databases such as MariaDB and MongoDB offer a native query language more in line with the expressed TyphonQL query (select from where). Migration of the data contained in Redis to a SQL or MongoDB database might be recommended. Each database technology has its own limitations and strengths. MongoDB, for example, does not offer an efficient way to perform joins between two collections. Therefore, if a TyphonQL query causes a join between two MongoDB collections, recommendations could be offered to the developer. Indeed, merging the two collections into one would avoid the need for a join and improve the query execution time. In order to use the recommandation system the user can inspect a

Figure 6: Query details tab of web application



Query: *from Employees e, EmployeeAddress a select e.HomePhone, a.Address where e.EmployeeAddress== a && a.City == "?"*

Filtering period: [✓ complete history] particular period

Details

Latest execution:
04/02/2021, 15:25:57 (8610 ms)

Query type:
SELECT

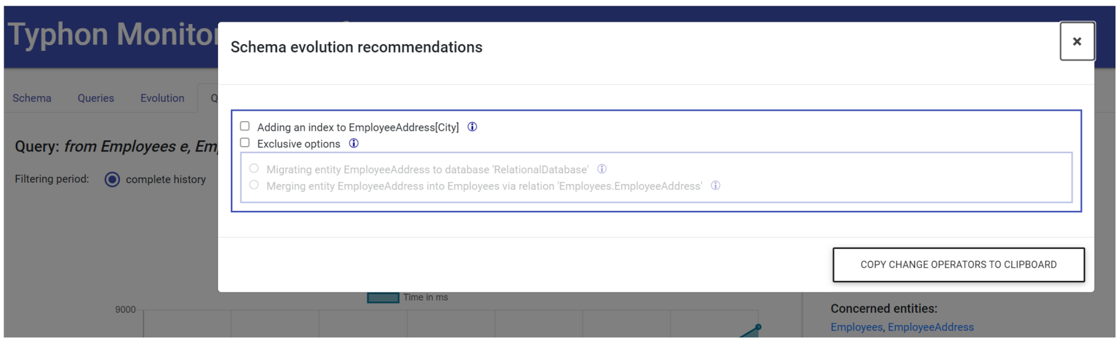
Concerned entities:
Employees, EmployeeAddress

Joins between entities:
(Employees ⊕ EmployeeAddress)

Entities implicitly inserted
-

particular (slow) query using the "Details" button attached to it. The user can inspect this query information and evolution of query execution time, as shown in Figure 6. The approach also proposes recommendations on how to improve the execution time of the query. When possible, it then recommends polystore schema reconfigurations, in the form of a menu with click-able options, including one of several recommendations. Some may be mutually-exclusive, which means that they cannot be selected together in the menu.

Figure 7: Recommendation pop-up of the web application



Schema evolution recommendations

- ☐ Adding an index to EmployeeAddress[City]
- ☐ Exclusive options
  - ○ Migrating entity EmployeeAddress to database 'RelationalDatabase'
  - ○ Merging entity EmployeeAddress into Employees via relation 'Employees.EmployeeAddress'

The recommendation menu shown in Figure 7 corresponds to the slowest TyphonQL query presented above. It consists in a join between *Employees* and *EmployeeAddress* entities and a selection operator based on the value of the *EmployeeAddress.City* attribute. In this case, the approach recommends three possible schema reconfigurations that respectively consist in (1) defining an index on attribute *EmployeeAddress.City*; and (2) choosing between migrating the *EmployeeAddress* entity to *RelationalDatabase* database or merging the *EmployeeAddress* and *Employees* entities.

By positioning the mouse on the information icon, the user can get further information about the expected positive impact of the recommended schema change on the query performance. Adding an index is a well known technique to speed up a query including an equality condition on an attribute (*e.g.,* table column or collection field) in its where clause. In the particular case of the considered query, *EmployeeAddress.City* is used in the where clause.

Migrating an entity and its relations from a database to another is interesting in our example as *Employees* entity is mapped to a relational database (a table) while *EmployeeAddress* entity is stored in a document database collection. The join condition of the considered query is slow since it involves to separately query two different databases, and then to aggregate the results in the form of a join. This recommendation is an exclusive choice with the merge recommendation described below, since the latter would include the migration of the entity.

The last proposed recommendation is the merge of two entities into a single entity. The merge constitutes another recommendation that prevents from a costly join condition in a slow query. In our example, the recommendation to merge *EmployeeAddress* into *Employees* is motivated by the fact that both entities are referencing each other using a one-to-one relationship (thus have the same number of records), that both entities rapidly grow in terms of size, and that they are stored in different databases (one in a relational database, and the other in a document database) making the join condition slower and slower. This recommendation will probably be more interesting in terms of performance since it includes an entity migration (from a database to another) and a relationship removal (by merging both entities).

3.7. Step 5: Applying recommendations

After the user has selected the evolution recommendation(s) he wants to actually apply on the polystore, it is possible to copy to clipboard the different evolution operators by clicking on the corresponding button. The tool will automatically generate the list of schema evolution operators (in the TyphonML syntax) corresponding to the selected recommendations. The user can simply paste the operators from the clipboard to the TML file of his polystore schema, and then run the schema evolution tool by passing the modified TML file as input.

The schema evolution tool will then apply the recommended schema reconfigurations to the polystore. This includes the adaptation of the polystore TyphonML schema, the underlying native structures and the data instances. In our example, we selected and executed the merge evolution operator. A new TyphonML schema has been produced as output where *EmployeeAddress* entity has been merged into *Employees* entity. The corresponding relational *Employees* table now contains its own columns and foreign keys, plus the new columns and data from *EmployeeAddress* collection.

Upon request of the user, the query evolution tool [19] can support the adaptation of existing TyphonQL queries in order to evolve a query into another according to the applied evolution operators. Thanks to the query evolution tool, the slowest TyphonQL query category can be automatically adapted, resulting in the following output TyphonQL query:

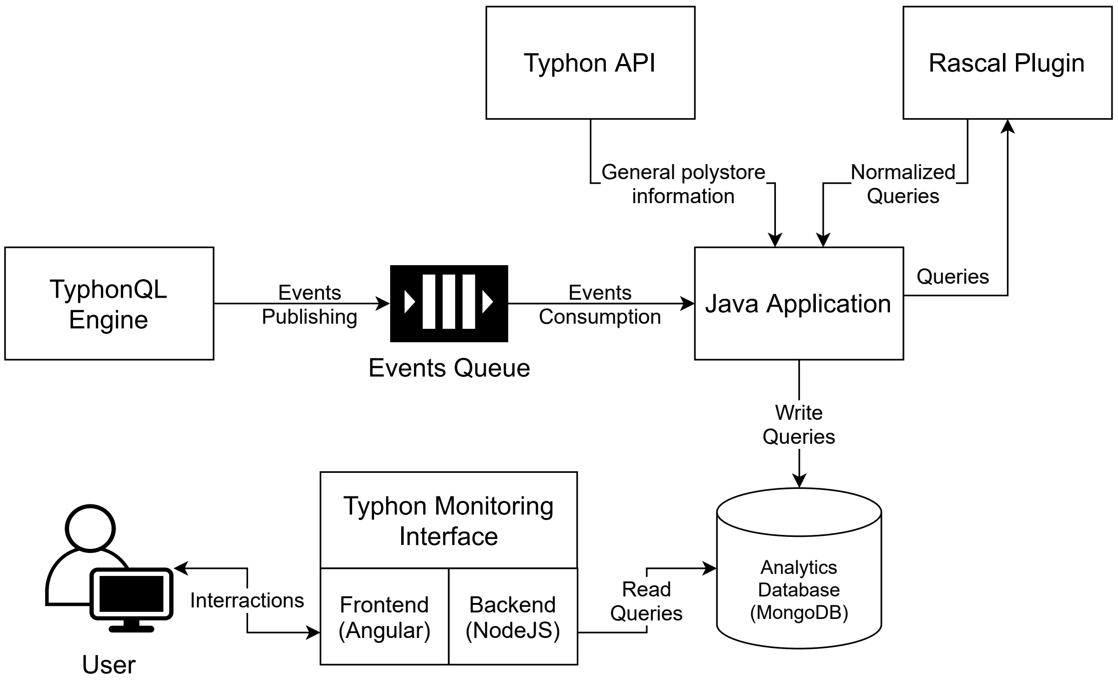
```
from Employees e select e.HomePhone, e.Address where e.City == "?"
```

The resulting query does not include any join condition between both entities. The performance of the query should therefore be significantly better than in the initial situation.

3.8. Implementation

The implementation of the approach is depicted in Figure 8. Each time a asynchronous post-

Figure 8: Overview of the architecture of the application



execution event is published to the post-event queue, a Java application wakes up and retrieves the event. If the event corresponds to a DML query execution, the Java application sends the corresponding TyphonQL query to a Rascal plugin[2]. The latter parses, analyses and classifies the query and sends back the corresponding query information to the Java application. This information is stored in an internal MongoDB database Figure 3, that is used as input by an interactive web application. The web application, relying on an Angular frontend and a Node.js backend, provides users with visual analytics of the polystore data usage, as well as with performance-based schema reconfiguration recommendations. The evolution module being asynchronous with the rest of the Typhon project tools, it does not impact the performance of the queries executed on the polystore.

[2]https://www.rascal-mpl.org/

4. Conclusion

This paper presents a tool-supported approach to performance-based schema recommendations for hybrid polystores. Starting from the capture and analysis of the polystore queries, the approach provides the user with visual analytics of the polystore data usage. The user can see, among others, which types of queries access which entities, which query categories are the most frequent, and which queries suffer from a lack of performance. The approach then makes, for each query category considered as problematic, possible schema reconfiguration recommendations aiming the speed-up the execution of the considered query. The user can then select the recommendations to follow, and then apply the associated schema evolution operators for schema change and data migration.

Those recommendations are currently made at the level of a single query that is considered too slow. A possible future improvement would be to consider a broader scope of analysis, by taking more information into account when making recommendations, such as (1) the most typical combinations of query categories involved; (2) the frequency of occurrence of those query categories, and (3) the respective performance levels of the underlying native DB backends.

Availability The approach and its implementation are publicly available as an open-source project under the terms of the Eclipse Public License [20].

Acknowledgments

This work was supported by the European Union H2020 research and innovation programme under the TYPHON project (#780251), and by the F.R.S-FNRS under the EOS SECO-ASSIST project (#30446992).

References

[1] E. Rahm, P. A. Bernstein, An online bibliography on schema evolution, SIGMOD Rec. 35 (2006) 30–31. URL: http://se-pubs.dbs.uni-leipzig.de/pubs/results/taxonomy%3A9. doi:10.1145/1228268.1228273.

[2] L. Meurice, C. Nagy, A. Cleve, Detecting and preventing program inconsistencies under database schema evolution, in: Proc. of IEEE QRS 2016, IEEE, 2016, pp. 262–273.

[3] J. Delplanque, A. Etien, N. Anquetil, S. Ducasse, Recommendations for evolving relational databases, in: Proc. of CAiSE 2020, volume 12127 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 498–514. URL: https://doi.org/10.1007/978-3-030-49435-3_31. doi:10.1007/978-3-030-49435-3_31.

[4] S. Scherzinger, T. Cerqueus, E. C. De Almeida, Controvol: A framework for controlled schema evolution in nosql application development, in: Proc. of IEEE ICDE 2015, IEEE, 2015, pp. 1464–1467.

[5] R. N. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, M. Kalinowski, Data management in

microservices: State of the practice, challenges, and research directions, Proc. VLDB Endow. 14 (2021) 3348–3361.

[6] P. Benats, M. Gobert, L. Meurice, C. Nagy, A. Cleve, An empirical study of (multi-) database models in open-source projects, in: International Conference on Conceptual Modeling, Springer, 2021, pp. 87–101.

[7] P. Thiran, J.-L. Hainaut, G.-J. Houben, D. Benslimane, Wrapper-based evolution of legacy information systems, ACM Transactions on Software Engineering and Methodology (TOSEM) 15 (2006) 329–359.

[8] S. Scherzinger, M. Klettke, U. Störl, Managing schema evolution in nosql data stores, arXiv preprint arXiv:1308.0514 (2013).

[9] M. J. Mior, K. Salem, A. Aboulnaga, R. Liu, Nose: Schema design for nosql applications, IEEE Transactions on Knowledge and Data Engineering 29 (2017) 2275–2289.

[10] C. de Lima, R. dos Santos Mello, A workload-driven logical design approach for nosql document databases, in: Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services, 2015, pp. 1–10.

[11] C. Del Grosso, M. Di Penta, I. G.-R. de Guzman, An approach for mining services in database oriented applications, in: 11th European Conference on Software Maintenance and Reengineering (CSMR'07), IEEE, 2007, pp. 287–296.

[12] A. Cleve, J.-L. Hainaut, Dynamic analysis of sql statements for data-intensive applications reverse engineering, in: 2008 15th Working Conference on Reverse Engineering, IEEE, 2008, pp. 192–196.

[13] M. H. Alalfi, J. R. Cordy, T. R. Dean, Wafa: Fine-grained dynamic analysis of web applications, in: 2009 11th IEEE International Symposium on Web Systems Evolution, IEEE, 2009, pp. 141–150.

[14] A. Cleve, N. Noughi, J.-L. Hainaut, Dynamic program analysis for database reverse engineering, in: International Summer School on Generative and Transformational Techniques in Software Engineering, Springer, 2011, pp. 297–321.

[15] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Pierantonio, L. Iovino, Typhonml: a modeling environment to develop hybrid polystores, in: Companion Proc. of MoDELS 2020, 2020, pp. 1–5.

[16] D. S. Kolovos, F. Medhat, R. F. Paige, D. D. Ruscio, T. van der Storm, S. Scholze, A. Zolotas, Domain-specific languages for the design, deployment and manipulation of heterogeneous databases, in: Proc. of MiSE@ICSE 2019, ACM, 2019, pp. 89–92. URL: https://doi.org/10.1109/MiSE.2019.00021. doi:10.1109/MiSE.2019.00021.

[17] A. Raina, Redis anti-patterns every developer should avoid, 2022. URL: https://developer.redis.com/howtos/antipatterns/.

[18] L. Schaefer, D. Coupal, Separating data that is accessed together, 2020. URL: https://developer.mongodb.com/article/schema-design-anti-pattern-separating-data.

[19] J. Fink, M. Gobert, A. Cleve, Adapting queries to database schema changes in hybrid polystores, in: Proc. of IEEE SCAM 2020, IEEE, 2020, pp. 127–131.

[20] TyphonEvolutıon, 2022. URL: https://github.com/typhon-project/typhon-evolution.