# Knowledge-based Recommendation for Polyglot Persistence

Philipp Eisenhuth
Institute for Computer Science, University of Bayreuth
Bayreuth, Germany
philipp.eisenhuth@uni-bayreuth.de

Stefan Jablonski
Institute for Computer Science, University of Bayreuth
Bayreuth, Germany
stefan.jablonski@uni-bayreuth.de

## ABSTRACT

Oftentimes modern applications have converging functionalities, which cannot all be handled with one type of data management system efficiently. This leads to applications requiring a variety of different data management systems for their individual components. Besides the traditional relational database management systems, new systems have been developed in the last few decades that are subsumed under the term NoSQL. The selection of suitable data management systems for each of the application components has potentially great impact on different aspects like performance. Therefore, this selection process needs sophisticated analysis and requires expert knowledge. To reduce the time required for this process and to enable an adequate selection even for developer teams without experts covering all kinds of data management systems, we suggest an approach based on a knowledge-based recommender system. We define its required input and describe how it processes it. As a proof of concept, we show the application of our approach for an exemplary use case. The database recommendation process is embedded in the overall design phases of applications with a polyglot data management landscape.

## 1 INTRODUCTION

Back then when database management systems (DBMS) came up, the area of data management was mainly concerned with relatively small, structured, and homogeneous data. Relational DBMS were typically a proper solution for all kinds of use cases. Through the rapid development of technology and the resulting growth in computing power and storage capacity together with the rise of the internet, there was a profound paradigm change in data management over the last decades. This gave the developers of applications completely new possibilities, which, in turn, also caused fundamentally new challenges for data management. A popular buzzword referring to this situation is the term *Big Data*. In this context, typically the following 5 V's are taken to characterize data management: *velocity, volume, value, variety, veracity*.

As a consequence, relational DBMS have not been the only solution for all kinds of data management problems anymore. They are not designed to handle these new requirements efficiently in a broadly manner. For example, they struggle with unstructured data, large volumes of data coming in at a very high frequency, or enormous simultaneous access. This has led to the emerge of new types of data management systems to compensate the weaknesses of the former, broadly subsumed under the term *NoSQL*. These systems were further classified into four main categories based on their data model, namely *Key Value Stores*, *Document Stores*, *Column Family Stores* and *Graph Databases* [7] [14]. Usually, the systems from each of these categories are designed to excel at very specialized use cases. In contrast though, they are also not suitable as a one-solution-fits-all approach in a broader application scenario. For example, Key Value Stores do not put great value on sophisticated data structuring. However, through their simple architecture they show excellent access performance. Similarly, the other NoSQL types foster certain features by neglecting others.

In modern applications it is very common that the requirements towards data management range from a traditional setting – solvable with relational DBMS – to very specialized demands – solvable by specialized NoSQL systems. Parts of an application with diverging but in itself homogeneous requirements can be regarded as application components, which, in turn, encapsulate distinct functionalities. Each application component therefore has its specific requirements towards data management. To meet these requirements, (at least one) suitable data management component (DMC) should be assigned to each application component. In the rest of this paper, we use the terms *application component* and *data management component* to characterize separated parts of an application with their individual required data management solution.

The co-existence of different types of data management systems within one comprehensive application is often the case and the resulting management and design processes become a challenge. This special approach for data management is mostly discussed under the term *polyglot persistence* [12]. Challenges in dealing with such a polyglot database architecture mainly arise in two different phases of an application's life-cycle:

**Design and setup phase:** Here, the selection and usage of adequate data management components for different application components plays a central role. This requires profound knowledge (e.g., based on experience of developers), otherwise the requirements of certain application components may not be met adequately. Additionally, data modeling for each of the selected data management component has to be done with respect to the corresponding data model of such a data management component.

**Administration and maintenance phase:** The operation of multiple data management systems within an application can become very complex. There could be queries spanning multiple heterogeneous data management components, different forms of distribution transparencies that have to be guaranteed, and updates of de-normalized data. These and various other challenges have to be coped with by the overall application.

Our approach to tackle these challenges is based on the application of a holistic, semi-automated design framework for applications requiring polyglot database architectures. Its focus lies mainly on the following issues:

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License and appears in CDMS 2022, 1st International Workshop on Composable Data Management Systems, September 9, 2022, Sydney, Australia.

1) Providing a ranking of suitable data management components for each application component, whereby corresponding explanations must be provided.

2) Based on 1), a preferably optimal composition of data management components, covering all requirements of the different application components as good as possible in a polyglot setting.

In this paper we focus on the selection and allocation process of suitable data management components for application components. We cope with this issue by the deployment of a knowledge-based recommender system. To enact this we provide a formalization of relevant data management requirements and properties of the considered data management components, together with an algorithm for their evaluation.

The rest of this paper is structured as follows: Section 2 focuses on related work of our proposed framework. Section 3 provides background information about recommender systems. In Section 4, the overall context of our framework is described, whereas Section 5 details our approach with a dedicated recommender system. A use case of our recommendation process is presented in Section 6. Section 7 concludes the paper with a lookout on future work.

## 2 RELATED WORK

A recent survey, mainly concerned with different design methods in the new database era [11], compared and categorized data modeling approaches. Amongst other things, they identified the following major issues in the investigated approaches: 1) The modeling methods do not consider application specific requirements sufficiently and 2) they mostly do not provide guidelines for choosing the best-fitting type of DBMS or modeling strategy for a given application component. This indicates that there is a clear need for supporting the decision process of choosing an appropriate data management component, based on various requirements and subsequent use the same information for decisions in the actual data modeling. The same authors tackled this problem with their own approach [8–10], where they developed a method for selecting the most suitable database type for fragments of a data model, based on application requirements. As a basis, they use a conceptual data model from which they create multiple fragments by clustering elements with respect to the structure of the data model itself, the expected queries and multiple non-functional requirements. The most suitable database type for each fragment is selected by comparing the properties of the fragment with properties of the different database types.

The concept of our framework extends their basic ideas, whereby we enhance the approach in various directions. We consider additional requirements for the application components, especially non-functional ones. The description of the data management components at choice is done with concrete properties of the respective systems, which results in the need of mapping the requirements from the application components to constraints on these properties. Our approach also considers the properties of concrete systems, instead of just describing properties on an abstract and therefore in-concrete level of types of data management systems. In our framework, structuring the properties of data management components is predefined but adaptable. Furthermore, our approach emphasizes the addition of descriptions of new data management components.

The influence of requirements on the selection of a suitable database system is also discussed in [4]. There, a toolbox for the manually selection of an appropriate data management system using a decision tree is provided. Our recommendation process would (semi-)automate this process.

Other approaches concerned with modeling for various novel data management systems (e.g., [2], [5], [3]) are relevant for the context of our overall process (Section 4). They mainly focus on data modeling languages (starting at the conceptual or logical level) and on how to generate concrete schemata (i.e., data model on the physical level) finally. In their transformation process, they neglect important considerations for choosing concrete systems and also modeling alternatives (e.g., based on various requirements).

The application of a recommender system for selecting an adequate cloud database is discussed in [13]. We follow a similar approach with our recommendation process, however, in a different application domain.

To the best of our knowledge, there is no (semi-)automatic database recommendation approach proposed so far, which also provides the possibility of easy extension and adaptation.

## 3 BACKGROUND: RECOMMENDER SYSTEMS

This work is concerned with multiple aspects of database design and selection, especially in a polyglot context, and connects this to the field of recommender systems. We provide some background to the general concepts of recommender systems, focusing mainly on their required input and functionality. The transfer of these concepts to our application domain is described in more detail in Section 4.

Recommender systems are typically known for product recommendations in an E-Commerce setting or for movie and music suggestions on streaming platforms. Their goal is to narrow the amount of *presented items* for each user, especially by highlighting the most relevant ones. However, their scope is not limited to these settings and they can potentially also be deployed in completely different domains.

There are multiple types of recommender systems, which all require different kinds of input. Most of them use historical data connected with their users, like the previous products users have bought or the movies they have watched, as a basis for the recommendation method. Often, also a rating – either explicit or implicit – is considered. Comparisons to *similar* users, either based on comparable histories or on similar preferences, typically also play a central role. In contrast to these systems, so called *Knowledge-based Recommender Systems* let the users describe the items to be recommended through a set of properties or requirements. These systems can further be distinguished into case-based and constraint-based systems. Our framework is built on the concept of the latter category. These types of systems have the core advantage that they can be used without historical data and can therefore start from scratch. They rely on a knowledge base for their recommendations, which has to be built up and filled prior to their deployment. Domain experts have to define various mappings for the requirements to generate constraints on the properties of the items to be recommended [1].

## 4 DESIGN PROCESS FOR POLYGLOT PERSISTENCE APPLICATIONS

Our overall method for polyglot database architectures will provide support throughout the whole development life cycle of an application. Thereby we focus on applications with multiple, at least partly, distinct functionalities and their corresponding data related requirements, encapsulated in distinct application components. Since these often have diverging requirements, the associated data also has to be handled differently.

For example, in a typical E-Commerce application, the data of a shopping cart is accessed by multiple users simultaneously. Requirements towards these data are completely different to those of data which is related to the buying process for a product. For the shopping cart data, performance and availability are most essential, whereas for the final purchase data, consistent and transactional execution is essential.

Our framework especially addresses inexperienced developers through a semi-automated, well-founded recommendation process, suggesting the usage of different data management systems for specific requirements. Following these suggestions and the consequent selection of adequate systems, logical data models can be constructed which in turn can then be used to automatically generate components for data processing [5]. Other important issues during the final development steps of an application are, amongst other things, concerned with providing uniform data access and dealing with synchronisation between multiple, partly replicated data collections in the polyglot landscape [15]. This also includes the efficient handling of cross-references between different system types, the adherence of data transparencies, and the handling of cross-database transactions.

To provide the broader context for the contribution of this paper, we briefly give an overview of the overall design process in the rest of this section. The process can be separated roughly into three logical parts as depicted in Figure 1. The first phase is concerned with the collection and formalization of multiple requirements for the application to be developed (*Requirements Engineering Phase*). This also includes a complete conceptual data model covering all parts of the applications data (*Step: Requirement Analysis*). Aiming at assigning different data management components to parts of this holistic data model (and therefore the application components), we need to partition it with respect to comparable requirements. Therefore we allocate these requirements to elements of the model (i.e., the concrete entities and relationships) and afterwards group them based on their similarities into partitions [9]. Again, this reflects the different application components (*Step: Model Partitioning*). The results from this partitioning together with the formalized requirements are brought to our knowledge base, which defines the input for our recommendation process (*Step: Database Selection*). The focus of this work lies on this step and, hence, is discussed further in Section 5. As a result, a list of data management components is generated, which should be applied for the various partitions of the data model. The next steps are concerned with the actual usage of this information (*Step: Polyglot Database Design*). Logical data models for each application component are created in the corresponding data model type of the selected system. Out of these, usable building blocks for the interaction with concrete systems within the application are created (*Step: Implementation*).

## 5 KNOWLEDGE-BASED DATABASE RECOMMENDATION

The focus of our work lies on the architecture of a general framework that enables a profound recommendation for adequate data management components for respective application components. Applied in this application domain, the input for a constraint-based recommendation algorithm can be categorized as follows (based on the definitions from [1]):

**User specifications:** In our case, these are the requirements gained in the requirements engineering process. These are represented as follows:

$$Requirement = (ReqCategory, RelatedPartition, ReqValues)$$

*ReqCategory* defines the category and name of a particular requirement. *RelatedPartition* determines the relation to the partition of the data model and therefore the covered entities and relationships of the concerned application component. Lastly, *ReqValues* expresses the potentially multiple values corresponding to a concrete requirement. The possible values belonging to one *ReqCategory* also can be ordered (e.g., for *Frequency* there is an ascending order from *Very frequent* to *Rarely*). We depict a list of most relevant requirements together with their associated value ranges for the selection process in Table 1 and Table 2. As a foundation for this, we evaluated several research approaches and actual industry projects, which are concerned with the influential factors for the database selection process ([6], [4], [9]). The *Requirement* column contains all possible values for the *ReqCategory*, whereas in the *Values* column, all associated *ReqValues* are defined. Each instantiation of such a formalized requirement refers to a concrete partition from the overall conceptual data model and can only be instantiated once for each application component. Each element from the data model is assigned to (at least) one partition, through which the relation to the requirement instances is defined.

The requirements are roughly categorized in *data-related*, *functional* and *non-functional* requirements. Data-related requirements refer to structural properties of the entities and their relationships, whereas functional requirements characterize the expected queries. Non-functional requirements further define qualitative properties of the former two. An example of a requirement represented in our formalization would be the following:

$$Req1 = (\text{"}Accessprofile\text{"}, partition1, \text{"}Reading(select)\text{"})$$

This requirement (*Req1*) states, that the *"Access profile"* for all data elements grouped in *partition1* has the value *"Reading (select)"*.

**Item and Item Attributes:** Items to be recommended are described by a set of properties. In our case, the items are the considered data management components with a set of relevant attributes. We use the term *profile* for describing a data management component with certain properties from now on. These are formalized in the following way:

$$Profile = (ProfileType, ProfileAttributes)$$

$$ProfileAttribute = (AttributeName, AttributeValues)$$

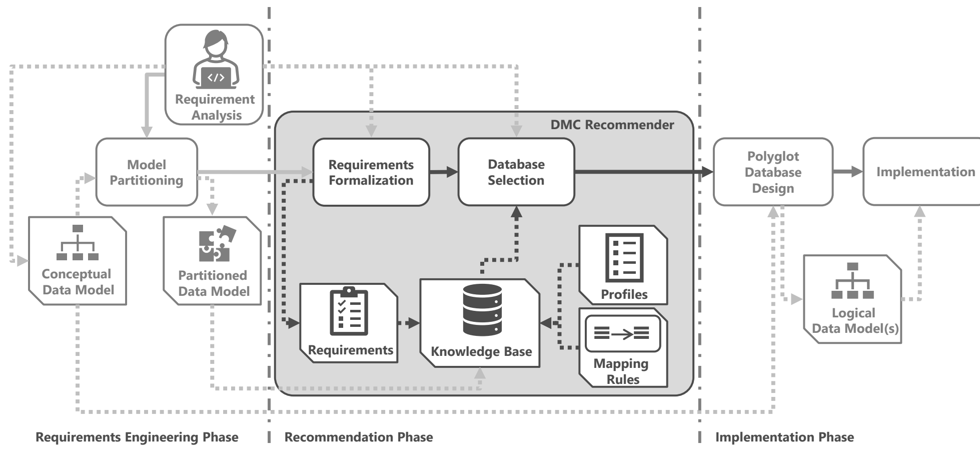


**Figure 1: Environment of our DMC Recommender Framework**

| Requirement | Values |
|---|---|
| ***Data-related*** | |
| Data connectivity | High, Middle, Low |
| Data structuredness | Unstructured, Semi-structured, Structured, Graph, Complex data types |
| Modeling flexibility | Flexible (no schema), Semi-flexible (implicit schema), Strict (explicit schema) |
| Variability | Single representation, Multiple representations |
| ***Functional*** | |
| Access profile | Reading (select), Writing (insert), Changing (update) |
| Access operations | Random access, Range query, Join, Restriction, Sorting, Full-text search |
| Frequency | Very frequent, Occasionally, Rarely |
| Purpose | Analysis, Regular operation, Administrative operation |

**Table 1: Data-related and functional requirements**

| Requirement | Values |
|---|---|
| ***Non-functional*** | |
| Consistency | Strong, Eventual |
| Integrity | High, Middle, Low |
| Access flexibility | Flexible, Static, Restricted |
| Data volume per entity | High, Middle, Low |
| Velocity of data income | Very slow, Slow, Medium, Fast, Very fast |
| Response time (access) | High, Medium, Low |
| Required throughput | High, Medium, Low |
| Expected data veracity | Clean data, Noisy data, Sparse / incomplete data |
| Data-related risk | High risk (business critical), Medium risk, Low risk (uncritical) |
| Querying constraints | Attribute range, Referential integrity, No constraints |
| Required deletion concept | Logical, Physical |
| Connected security mechanisms | Standard (authentication), Encryption, None |

**Table 2: Non-functional requirements**

Here, a *ProfileType* is described by a set of *ProfileAttributes*. These, in turn, are composed of an *AttributeName*, together with potentially multiple *AttributeValues*. The list of different *AttributeNames* is fixed, whereas the concrete *AttributeValues* are free to be chosen depending on a specific data management component. The *ProfileAttributes* can be classified into one of these three types according to their potential values: *Scale*, *Boolean* and *Categorical*. Values for elements from the *Boolean* type can be either *true* or *false*, whereas for *Scale*, they are numerical values ranging from 0 to 10. *Categorical* attributes have specific values, depending on the concrete attribute. We assign the following attributes to each type:

**Scale**: *Simple data types, Complex data types, Depiction of heterogeneous datasets, Aggregations, Scope of operations, Query language, Secondary access path, Performance read operations, Scalability read operations, Read latency, Read availability, Performance write operations, Scalability write operations, Write latency, Write availability, Tool support, Existing Expert knowledge*

**Boolean**: *Attribute names, Join operations, Sorting operations, Scan queries, Filter queries*

**Categorical**: *Multi-model support (Values: <list of covered data models>), Execution guarantee (Values: Partition tolerance (CAP), Availability (CAP), Consistency (CAP), Atomicity (ACID), Consistency (ACID), Isolation (ACID), Durability (ACID))*

Profiles can be adjusted or new ones can be added, although the used terms are predetermined through the just mentioned attributes. The possibility to also alter the list of available profile attributes will be considered in future work. The values of these attributes are used in the constraints later on to determine the compatibility

with a specific set of requirements. Examplary profiles are shown in Table 4 in Section 6.

**Constraint:** A constraint imposes restrictions on a set of attributes from the recommended items (i.e., the profile attributes). Constraints have the following structure:

$$Constraint = (AttributeName, Operator, AttributeValues)$$

*AttributeName* defines the category of a particular property from the profiles. Through *Operator* their desired values are restricted. Possible *Operators* are $=, <, \leq, >, \geq, \in$. Depending on the type of the attributes, only a subset of these operators are applicable, i.e., for *Scale* all but the $\in$ operator are eligible, for *Boolean* only the $=$, and for *Categorical* only the $=$ and $\in$ operator can be applied.

**Domain knowledge:** In a *Knowledge base*, transformation rules for mapping user specifications (here: requirements and their values) to item attributes (here: profile attributes) are organized. These rules constitute domain knowledge by producing constraints in terms of item attributes, whereas they take conditions in terms of user specifications as input. For example, if requirements analysis results in an *Access profile* that is mainly *Reading* and the corresponding *Frequency* is greater than *Rarely*, one resulting constraint (by application of the respective mapping rule) could be that the *Performance of read operations* or the *Scalability of read operations* has to be greater than a certain threshold (e.g., 3). Both latter attributes are attributes from the profile, whereas the former are requirements for an application component.

The structure of a mapping rule is as follows:

$$if(Requirements)then(Constraints)$$

Examples of such rules are shown in Figure 2 (Section 6). For further refinement of the recommendation process of our framework, additional input information for recommendations will be considered in future work. This includes, amongst other things, the feedback or ratings of completed runs of the algorithm. These results are then stored in this *Knowledge base*.

**Algorithm:** Algorithm 1 shows the pseudocode of the actual recommendation algorithm. The procedure is the following: The compatibility of all requirements from an application component (AC) with the premises of the mapping rules is tested (lines 1-3). If there is a compatible rule within the knowledge base, the consequence of this rule is added to the constraints of the knowledge base (line 4). Afterwards, for each data management component it is tested, whether the (recently added) constraints from the knowledge base can be satisfied (lines 5-6). If so, the data management component is added to the ranking with a previously computed score and the degree of satisfaction is recorded (line 7-8). If not, the conflicting constraints are computed and added to a corresponding data structure (line 9). If no data management component can satisfy all the constraints, a compromise has to be computed (line 10-12). Finally the ranking and either the satisfiable or unsatisfiable constraints are returned in both cases (line 13-14). The returned degree of satisfaction for the constraints can be used for explaining the provided rankings. Afterwards, the user will then be given the choice to select a concrete data management component from the provided ranking. This has not necessarily to be the highest ranked system. Such divergent choices are stored within the knowledge base, to improve further recommendations by adjusting the mapping rules.

```
   Data: Application Component (AC) with its Requirements (Req),
         List of DMCs, Knowledge Base (KB)
   Result: Ranking of suitable DMCs for Application Component,
           Fulfillment of constraints
1  foreach Req from AC.Reqs do
2      foreach MappingRule from KB.mappingRules do
3          if compatible( Req, MappingRule.premise) then
4              KB.constraints.add( MappingRule.consequence )
           end
       end
   end
5  foreach dmc from KB.DMCs do
6      if satisfiable ( KB.constraints, dmc ) then
7          KB.ranking.add( evaluateWithWeights ( dmc ) )
8          KB.satisfiableConstraints[dmc].add(
             computeSatisfactionConstraints ( KB.constraints, dmc ) )
       else
9          KB.unsatisfiableConstraints[dmc].add(
             computeConflictingConstraints( KB.constraints, dmc )
       end
   end
10 if isEmpty( KB.ranking ) then
11     foreach dmc from KB.DMCs do
12         KB.altRanking.add( evaluateCompromise( KB.constraints,
             dmc ) )
       end
   else
13     return KB.ranking, KB.satisfiableConstraints
   end
14 return KB.altRanking, KB.unsatisfiableConstraints
```

**Algorithm 1:** DMC Recommendation Algorithm

In this section, we provided the necessary information for a profound recommendation process. This includes the structure for the previously mentioned profiles (i.e., the list of their attribute names and corresponding domains). Instances following this structure can be added by an elaborated pre-analysis of these systems. Since a predefined list could by no means be complete – as it doesn't and possibly can't include all available systems –, extensibility is a fundamental demand.

The list of considered requirements or the structure of the profiles also can be extended or altered. However this has a greater impact on the overall framework, and therefore should be done with care. The necessary steps for this are: 1) An additional entry to the list of requirements of profile attributes has to be added, including its corresponding values, 2) additional mapping rules have to be added, which translates the requirements into constraints on the profile attributes, whereby possibly conflicting rules have to be revised.

## 6 USE CASE

For an exemplary application of our recommendation process, we consider the application component of a product catalogue from an E-commerce application. This component could have the requirements as shown in Table 3.

Especially functional requirements may be dependent on each other, which is denoted with a number within brackets. Here, one

| Requirement | Value |
|---|---|
| Data structuredness | Semi-structured |
| Access profile | Reading (select) (1),<br>Writing (insert) (2) |
| Access operations | Restriction (1) |
| Frequency | Frequently (1), Rarely (2) |
| Access flexibility | Flexible |
| ... | ... |

**Table 3: Req.: Application Component Product Catalogue**

| AttributeName | AttributeValue | |
|---|---|---|
| | *MongoDB* | *MySQL* |
| Attribute names | true | true |
| Scope of operations | 7 | 10 |
| Perf. read ops. | 1 | 5 |
| Scalability read ops. | 10 | 2 |
| Perf. write ops. | 2 | 1 |
| Scalability write ops. | 2 | 2 |
| Aggregations | true | false |
| Complex datatypes | 10 | 0 |
| Join operations | true | true |
| ... | ... | ... |

**Table 4: Profiles: MongoDB and MySQL**

of these is the *Access profile* and *Frequency*, where for example the values *Reading (select)* and *Frequently* are related.

Exemplary descriptions of specific data management components for a document database (MongoDB) and a relational database (MySQL) are shown in Table 4. Due to space limitations we restrict ourselves to present more characterizations. Figure 2 shows exemplary mapping rules for our use case.

```
if ( Data structuredness = Semi-structured )
    then ( ( Aggregations = true ) | ( Complex datatypes > 4 ) )

if ( ( Access profile = Reading (select) ) & ( Frequency > Rarely ) )
    then ( ( Perf. read ops. > 1) | ( Scalability read ops. > 3 ) )

if ( ( Access profile = Writing (insert) ) & ( Frequency > Rarely ) )
    then ( ( Perf. write ops. > 1 ) | ( Scalability write ops. > 3 ) )

if ( Access operation = Restriction )
    then ( ( Attribute names = true ) & ( Scope of operations > 3 ) )

if ( Access flexibility = Flexible )
    then ( ( Join operations = true ) | ( Scope of operations > 5 ) )

...
```

**Figure 2: Relevant Mapping Rules**

After the concrete requirements of the application component are defined, the compatible mapping rules are applied. Through the fulfillment of the resulting constraints, a ranking of the most suitable data management components is computed. In this small example, the only data management component which satisfies the resulting constraints would be MongoDB.

# 7 CONCLUSION AND FUTURE WORK

In this paper we show the utilization of a knowledge-based recommender system for the ranking of suitable data management component for a set of requirements. For its input, we define a set of relevant requirement categories and provide the structure for describing data management components. The utilization of a recommender system within this context is a very promising approach to tackle the selection process of suitable data management components based on various requirements. Descriptions of further data management systems can easily be added, making the framework flexible enough to cope with future developments.

As already mentioned in Section 4, we will include our recommendation process in a framework to support the whole development life cycle of an application in our future work. Therefore we plan to extend our work on several parts. For the preceding steps of our recommendation process, we will provide a flexible data modeling tool to define the conceptual data model of the whole application and then interactively choose between different compositions of fragments for the data model (which are then used as input for the recommendation process). Afterwards we will use the results as input for generating logical data models for the chosen systems. Here the choice of adequate modeling techniques, also based on the previously defined requirements and constraints, has to be made. These will then be used to generate communication modules for the concrete systems in a later step, also including refined components for supporting the management of complex tasks in such a polyglot setting.

# REFERENCES


[1] Charu C Aggarwal et al. 2016. *Recommender systems*. Vol. 1. Springer.
[2] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. 2020. TyphonML: a modeling environment to develop hybrid polystores. In *Proceedings of the 23rd ACM/IEEE Int. Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–5.
[3] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. 2021. Athena: A Database-Independent Schema Definition Language. In *Int. Conference on Conceptual Modeling*. Springer, 33–42.
[4] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. 2017. NoSQL database systems: a survey and decision guidance. *Computer Science-Research and Development* 32, 3 (2017), 353–365.
[5] Maxime Gobert, Loup Meurice, and Anthony Cleve. 2021. Conceptual Modeling of Hybrid Polystores. In *Int. Conference on Conceptual Modeling*. Springer, 113–122.
[6] Robin Hecht. 2015. *Konzeptuelle und Methodische Aufarbeitung von NoSQL-Datenbanksystemen*. Ph.D. Dissertation. University of Bayreuth.
[7] Robin Hecht and Stefan Jablonski. 2011. NoSQL evaluation: A use case oriented survey. In *2011 Int. Conference on Cloud and Service Computing*. IEEE, 336–341.
[8] Noa Roy-Hubara. 2019. The Quest for a Database Selection and Design Method. In *CAiSE (Doctoral Consortium)*. 69–77.
[9] Noa Roy-Hubara, Peretz Shoval, and Arnon Sturm. 2019. A Method for Database Model Selection. In *Enterprise, Business-Process and Information Systems Modeling*. Springer, 261–275.
[10] Noa Roy-Hubara, Peretz Shoval, and Arnon Sturm. 2022. Selecting databases for Polyglot Persistence applications. *Data & Knowledge Engineering* 137 (2022), 101950. https://doi.org/10.1016/j.datak.2021.101950
[11] Noa Roy-Hubara and Arnon Sturm. 2019. Design methods for the new database era: a systematic literature review. *Software and Systems Modeling* 19 (2019), 297–312.
[12] Pramod J. Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* (1st ed.). Addison-Wesley Professional.
[13] Soror Sahri, Rim Moussa, Darrell DE Long, and Salima Benbernou. 2014. DBaaS-expert: A recommender for the selection of the right cloud database. In *International Symposium on Methodologies for Intelligent Systems*. Springer, 315–324.
[14] Lena Wiese. 2015. Advanced Data Management. In *Advanced Data Management*. De Gruyter.
[15] L. Wiese. 2015. Polyglot Database Architectures = Polyglot Challenges. In *LWA*.