Query Strategies on Polyglot Persistence in Microservices

Luís H. N. Villaça
Petrobras Transporte S.A.
Federal University of the State of Rio de Janeiro (Unirio)
Rio de Janeiro, Brazil
luis.villaca@uniriotec.br

Leonardo G. Azevedo
IBM Research
Federal University of the State of Rio de Janeiro (Unirio)
Rio de Janeiro, Brazil
azevedo@uniriotec.br
lga@br.ibm.com

Fernanda Baião
Federal University of the State of Rio de Janeiro (Unirio)
Rio de Janeiro, Brazil
fernanda.baiao@uniriotec.br

ABSTRACT

Organizations have to quickly leverage both internal and external capabilities to survive and compete in the current world, using all possible resources to effectively solve their problems. In a microservice architecture, solutions are built through collaboration with external and internal parties, who promote distributed services across networks. An issue in this scenario is to query data across multiple services, within or beyond the organization. As an example, a single report may correlate information from services that deal with specific data sources such as graph, XML, document-oriented and relational databases – configuring a polyglot persistence setting. This work characterizes and analyzes available solutions to query data in a microservice architecture, based on academia and industry. Strengths and weaknesses of the solutions are provided, according to relevant software quality characteristics based on ISO 25010 model, aiming to guide efforts on future researches.

CCS CONCEPTS

• **Information systems** → **Distributed retrieval**; **Mediators and data integration**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **Software and its engineering** → Abstraction, modeling and modularity;

KEYWORDS

microservices;polyglot persistence;query

ACM Reference format:
Luís H. N. Villaça, Leonardo G. Azevedo, and Fernanda Baião. 2018. Query Strategies on Polyglot Persistence in Microservices. In *Proceedings of SAC 2018: Symposium on Applied Computing , Pau, France, April 9–13, 2018 (SAC 2018),* 8 pages.
DOI: 10.1145/3167132.3167316

1 INTRODUCTION

Microservices architecture comprises autonomous services that work together, using different technologies to achieve specific purposes [24]. Characteristics for this technology include: organization around business domain (which defines microservices' boundaries); evolutionary design; and collaborative work. In this architecture, independent deployable units of software achieve specific purposes through cooperation with other components [10].

A microservices' feature is "Decentralized Data Management" [10], where services manage their own database, eventually using different instances of the same database technology, or even entirely different data store technologies. A scenario where Data Management technology diverges among collaborative services is known as polyglot persistence. Polyglot Persistence [19] implies choosing the best fit persistence model for each task to be performed.

According to the CAP theorem [3] (Consistency, Availability and Partition tolerance), at most two of these aspects are achievable for systems that share data. Since network partitions are inherent in microservice scenario, we have to balance availability and consistency, which combined with different persistence technologies bring unforeseen demands that require complex integrations [9].

This work focuses specifically on issues that arise once we query data from services that use different databases technologies. It characterizes the state of the art of query strategies for microservices employing polyglot persistence based on academia and industry approaches. Strengths and weaknesses of each approach are provided, along with use cases. The strategies were evaluated according to software quality criteria based on ISO 25010 model [16]. This model is an evolution of ISO 9126 model, which has been considered the most important standard to assess quality features on software solutions [2]. This study provides a fundamental survey for future researches on polyglot persistence on microservices.

The remainder of this work is organized as follows. Section 2 presents the background. Section 3 presents the evaluation criteria. Section 4 presents the strategies evaluation. Section 5 discusses the evaluation results. Finally, Section 6 presents the conclusion and proposal of future work.

2 CONCEPTS

2.1 Microservices

Microservices architectural style has emerged as a standard from real-world use. It is an approach to develop applications as a suite of small services, deployed separately, and communicating through lightweight mechanisms [10, 24]. There are seven microservices main principles [36] based on popular publications [10, 24]:

- **Fine-grained interfaces**: Define units with specific responsibilities that encapsulate both processing logic and data, where the latter is exposed via Web APIs or asynchronous message queues. These can be managed and scaled independently of each other;

SAC 2018, Pau, France

DOI: 10.1145/3167132.3167316

- **Business-driven development practices**: Employ techniques, and principles to identify and conceptualize services - such as Domain-Driven Design, focusing on the most valuable organization's features [5];
- **Cloud-native design principles**: Employ guidelines such as distribution, elasticity, automated management and loose coupling [6].
- **Decentralized continuous delivery**: Promotes a high level of automation and autonomy, which requires maturity to build and test services.
- **Lightweight containers**: Make use of effective packaging methodologies to promote artifacts through continuous deployment processes (*e.g.*, Docker [21]).
- **DevOps**: Employ automated approaches for configuration, performance and fault management, extending agile practices and using service monitoring.
- **Multiple paradigms**: Combine and leverage best fit computing approaches and storage types (*e.g.*, relational or NoSQL databases) in a polyglot programming and persistence strategy.

2.2 Polyglot Persistence

Polyglot Persistence [19] implies choosing the best persistence model for each task to be performed, considering each one can be designed to best solve different problems. Figure 1 illustrates an example scenario [30], where an "e-commerce platform" consumes four services which have their own DBMS (Database Management System) with different technologies. Polyglot Persistence here refers to scenarios where different types of data store are used.

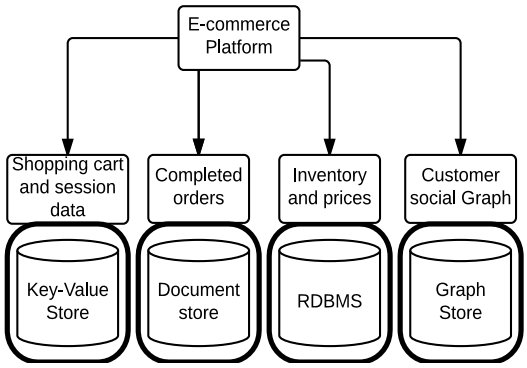

Figure 1: Polyglot persistence - adapted from [30]

3 QUALITY CRITERIA

The international standard ISO 9126 [15] is the most important orientation to evaluate the quality of software solutions [2]. It defines characteristics that fits formalized specifications and implicit needs, *e.g.*, subjective conditions as the capability for evolution in time.

In ISO 9126, software product quality attributes are classified in a hierarchical tree structure of characteristics applicable to every kind of software. This standard has been superseded by ISO/IEC 25010 [16], which categorizes product quality properties based on eight main characteristics (Functional Suitability, Reliability, Performance Efficiency, Usability, Security, Compatibility, Maintainability, and Portability), and it is a comprehensive basic standard that can be extended and tailored for specific needs [1].

We used these characteristics in the evaluation, except Functional Suitability and Security, since functional requirements were not available and security was not covered by any strategy references. The remaining applicable characteristics, along with related sub-characteristics are listed below. A code representation was assigned to each characteristic and sub-characteristic to link them to strategies analysis and evaluations.

- **Reliability (R):** Degree the solution or component performs specific functions under certain conditions for a period of time. This aspect is composed of:
  - *R1 (Availability):* systems are operational and accessible when required;
  - *R2 (Fault Tolerance):* systems remain functional despite of hardware or software faults;
  - *R3 (Recoverability):* systems recover data affected by incidents and re-establish the desired behavior.
- **Performance Efficiency (PE):** Performance in terms of:
  - *PE1 (Time Behavior):* response and processing times, and throughput rates are adequate;
  - *PE2 (Resource Utilization):* amounts and types of consumed resources are adequate.
- **Usability (U):** Degree to which the solution may be acknowledged, properly operated and controlled:
  - *U1 (Learnability):* systems can be understood and used to achieve effectiveness and efficiency;
  - *U2 (Operability):* systems have attributes that make it easy to operate and control.
- **Compatibility (C):** Degree a solution can exchange information with other systems without impacting any other product:
  - *C1 (Co-existence):* systems perform required functions efficiently while sharing resources with other products without critical impact on any other product;
  - *C2 (Interoperability):* degree systems are able to both provide and consume information from other systems.
- **Maintainability (M):** Degree of effectiveness and efficiency related to solution evolution. Critical factors are:
  - *M1 (Modularity):* coupling degree, a change to one component has minimal impact on other components;
  - *M2 (Reusability):* an asset can become a component of a solution composition;
  - *M3 (Analysability):* capability to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose deficiencies;
  - *M4 (Modifiability):* systems can be effectively and efficiently modified without introducing defects or degrading quality;
  - *M5 (Testability):* test criteria can be established, and tests can be performed to check compliance.
- **Portability (P):** Degree associated with management of transferring activities, according to:

- *P1 (Adaptability):* systems can be adapted to different or evolving hardware, software or environments;
- *P2 (Installability):* systems can be successfully installed or uninstalled in specific environments;
- *P3 (Replaceability):* a system can replace a product created for the same purpose in the same environment.

4 STRATEGIES EVALUATION

This section presents the strategies for querying on polyglot persistence in microservices, along with their strengths and weaknesses, and evaluation according to ISO/IEC 25010 criteria (Section 3). Positive and negative feedbacks were collected from publications that refer, either directly or indirectly (in this case they refer to similar situations) to each strategy, and related studies are included.

There are five main strategies. The first four are based on a recent publication [24]. The last one is based on [11], published before 2012 when "microservices" term was not coined yet [10]. It was chosen due to their architecture design similarity.

4.1 Shared Database

In this strategy, multiple services benefit from the same data source [24], and possible mechanisms for data persistence are [22]:

(i) Each service owns a set of tables in a shared schema;
(ii) Each service has a database schema in a shared DBMS;
(iii) Each service has its own database system (the data sources are integrated to provide a consolidated view).

The last choice may use different technologies for data stores (configuring polyglot persistence), requiring solutions such as database gateways and links [20], or federated database compositions [32].

Strengths

(1) [U1,U2]: Development and operation tasks are simplified, and services have instant access to data [22];
(2) [PE1,PE2]: Query performance gain is expected, due to direct access to data [22]. Network traffic is minimized, since database joins are constrained to databases and related integration mechanisms.

Weaknesses

(1) [M1,M2,M4,C1,C2]: High coupling - the service becomes inseparable from the database engine [22]. Schema changes may impact all services [24];
(2) [R1,R2,R3]: Database becomes the single point of failure;
(3) [M3,M5]: Test and debug activities depend on the implementation of the DBMS [22] and connector mechanisms;
(4) [P1,P2,P3]: Solution may require suppliers' gateways, links or connector components. Also, migrating existing assets to a different provider may be challenging, since the solution is coupled to data source features.

Related Work - In the *Database is the Service* approach [22], the DBMS embeds the business logic that implements the desired service (as long as it provides hooks to extend its capabilities), acting as a business service, and client requests are routed to it via a load balancer (Figure 2). In this scenario, the traditional service layer disappears (along with application server infrastructure), and there is no need to implement a discovery logic into clients, since the database cluster layer supplies service registry capabilities.

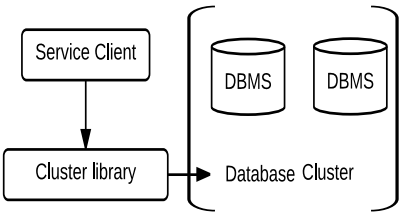

Figure 2: "Database is the Service" - adapted from [22]

4.2 CQRS

This approach, known as Command Query Responsibility Segregation (CQRS), proposes the use of a materialized view of the main databases for information retrieval purposes (*query-side*), isolating transactional operations (*command-side*) from query processing [24]. Distinct data representations are used for command and query components to properly assist their tasks.

Possible implementations [7, 24] for command-side components include a *Command Service* (Figure 3). It receives commands (*updates*) from a client to modify data entities. These commands are validated according to business rules using a *Command Data Model*. If no issue is found in the validation, they are sent to the *DB Writer* and *Publisher*. *DB Writer* applies the commands to the command-side data store (*Command Domain Data*). The Publisher generates an event, further sent to *query sides* (which provide *Query Services*) through some mechanism such as *Database Trigger* or *Event Store*) to synchronize the databases.

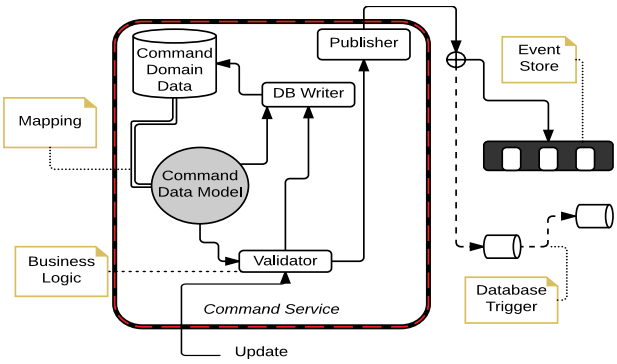

Figure 3: CQRS command-side components

The *Query Service* (Figure 4) receives notifications of data entity change events. From there, several alternatives are possible to provide reports. For instance, data warehouse tables (represented through *Query Data Objects*) can be populated, or the state of those entities can be reconstructed by replaying a series of events from a queue (on a process known as *Source Routing*). In this case, state snapshots can be taken periodically to avoid reprocessing all events.

Strenghts

(1) [PE1, PE2, R1]: This approach is an option to split parts of the system that have an emphasis on consistency from

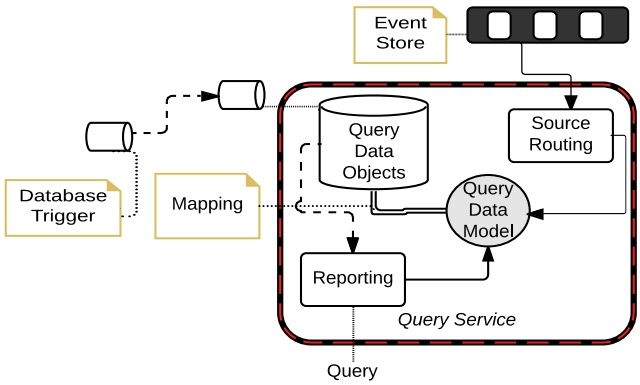

Figure 4: CQRS query-side components

parts that should have emphasis on availability or partition tolerance [18], optimizing resources and reducing response time for query operations (specially on high concurrent load scenarios), since queries no longer interfere (neither are impacted by) database transactions.

(2) [C2,M1,M2,P1,P2,P3]: Promotes the design of more granular and loosely coupled components [23], so coupling with the underlying database of source systems is avoided.

(3) [C1]: The proposed segregation allows different kinds of scaling. The command and query sides may be deployed as separate services, over separated hardware infrastructures and using different types of data stores [24].

Weaknesses

(1) [R2,R3]: Consistency of queries may be compromised, *e.g.*, if synchronization fails or takes too long. When collaborating with external systems an application state may need to be rebuilt by replaying events[30], which can be complex.

(2) [M3,M4,M5,U1,U2]: Complexity to maintain and track information updates, especially when using event sourcing [23, 24].

Related Works - Medis.NET is a medical information system for primary and ambulatory care medical facilities in the Republic of Serbia [29]. This system aggregates information of more than 2 million patient admission forms, along with medical exams, laboratory analysis and therapeutic treatments.

It started showing slow responses as the amount of information became larger and data got scattered in separate tables. The performance for queries was impacted as the amount of data traffic on join operations became much bigger. Hence, to reduce the impact on load operations in the main database, a separate component with a new data store was created just for reading data (Figure 5), improving overall performance.

In another example scenario, a fictitious company (named *Shipping Inc.*) provides three microservices which interact with each other through lightweight interfaces [23].

(1) Customer Manager: Maintains customer accounts;

(2) Shipment Manager: Responsible for the life cycle of a package from drop-off to final delivery; and,

(3) Shipment Tracker: Provides reports to customers to track their shipments.

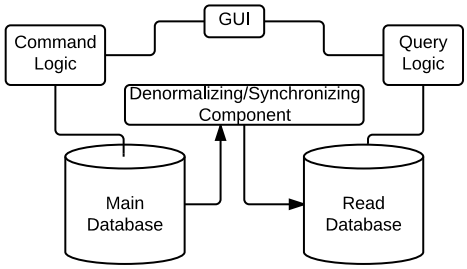

Figure 5: Medis.NET CQRS components - adapted from [29]

The Shipment Manager triggers events as the package moves through facilities along the delivery route. Shipment Tracker, which consumes those events, build its own data structure according to the best technology that fits its need. It can also pull and correlate information from Customer Manager through published events.

4.3 Event Data Pump

In this strategy, a service notifies another service when information is updated, so that the latter can pull it further ahead [24]. Events are temporal in nature. Data can be sent to components designed for reporting as events raise, instead of the report component waiting for scheduled jobs to pick up data updates. A *publisher-subscriber* mechanism can be used to queue and distribute events to parties.

Strenghts

(1) [PE1,PE2,R1]: Query performance is improved as all relevant information is available in advance for each node. Good performance, availability and low resource consumption metrics for a use case deployed in Production with over 400,000 users were collected by Viennot *et al.* [33];

(2) [C1,C2,M1,M2,P1,P2,P3]: Promotes loose coupling, data availability [24], facilitating integration among services.

Weaknesses

(1) [R2,R3,M3,M4,M5]: Failures of a publisher or a subscriber may impact consistency and, due to distributed nature of the architecture, it can become difficult to investigate.

(2) [U1,U2]: Complexity to operate. The architecture for this approach can evolve to a complex ecosystem of services that subscribe to data from each other [33], and since all required information is broadcast as events [24], precautions (*e.g.*, data traffic estimations) should be adopted.

Related Work - Synapse, a scalable replication system, uses this strategy [33]. It allows different services (handling different data structures) to be independently deployed, considering their own databases. Differences may exist on schema, indexes, layouts, and engines, but they share subsets of their own data.

In this framework, services can act as publishers (sharing some of its information) or subscribers (receiving shared read-only information). For that, each service specifies its metadata through ORM (Object-Relational Mapping [25]), which derives the data model definition constructs. This model abstracts DB-related details and let programmers specify information in terms of high-level objects, which may correspond to individual rows (or NOSQL documents) in the underlying databases.

Figure 6 illustrates Synapse architecture. At the publisher, Synapse Query Intercept Module is located between the ORM and the DB driver. It identifies which objects are being written and intercepts the updates, passing them to the Synapse Publisher, which then marshals all attributes of any created or updated object, attaches the IDs of any deleted objects, and constructs a write message. Synapse sends the write message to a persistent and scalable message broker system (Reliable Pub/Sub), which distributes the message to the subscribers (Synapse Subscriber). The subscribers update their databases accordingly with the received write message.

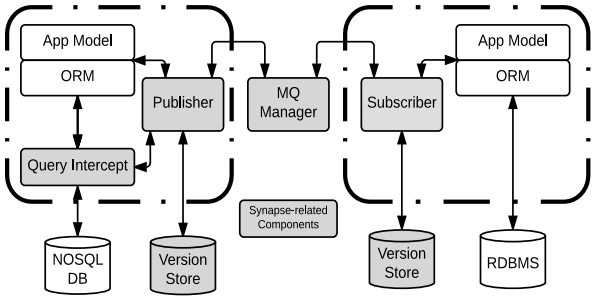

Figure 6: Synapse components - adapted from [33]

4.4 Data Retrieval via Service Call

This strategy relies on pulling the required data from source systems via service calls [24]. Usually, multiple calls are needed to produce reports. Queries may become very slow when pulling large volumes of data to compose a view from different services. Besides, service for data provisioning should be designed carefully. For instance, a service may provide data comprising sets of attributes in different methods that do not suffice service consumers (which may require all the data), demanding several calls. Caching mechanisms may also be applied, but dynamic reporting needs may result in potentially high cache miss rates.

Strenghts

(1) [P1]: Services (regardless of data sources) can be incorporated into the existing infrastructure (this approach emphasizes decoupling);
(2) [M2,U1,U2]: Individual cohesive applications can be reused and grouped into composite solutions [35].
(3) [M1,M3,M5,P2,P3]: Services, designed as self-contained units, can be optimized, evaluated and scaled with less impact to other components [4].

Weaknesses

(1) [PE1]: Performance can be severely impacted when performing reporting operations that gather data from multiple data stores (through microservices). Use cases that require larger volumes of data are likely to be impacted [24].
(2) [PE2]: Due to collaborative nature of those components, demands for network resources (such as network bandwidth) increase once the number of components increases [36].
(3) [C1,C2,M4]: Due to the autonomy in development, software evolutions (if not carefully planned) may lead to redundancy and ambiguity when information concepts are the same, or similar, across business domains. Services exposed by various microservices may not be designed for reporting use cases [24].
(4) [R1,R2,R3]: Microservices are designed to tolerate failure or unavailability of their information suppliers. Additional precautions are required to handle it [10].

Related Work - Trajectory Inventory Service [31] is a use case scenario in the area of mobile objects trajectory data. The amount of data handled by solutions in this area is relevant. *E.g.*, OpenStreetMap system [14] supports more than five billion uploaded GPS points, and the structure of sent data is not consistent between the devices. An estimate of the database size, considering sensors carrying just latitude, longitude and a time stamp, is 820 GB of data [31]. Also, in this scenario, both volume and structure of information increases exponentially, demanding persistence mechanisms to scale processing and storage capabilities. This study handles this requirement through microservices, allowing components to be developed by different teams according to technologies that best fit the needs.

As presented in Figure 7, each persistence service holds information for a specific repository, where trajectories data (coordinates, timestamp etc.) is stored. Information about those repository items (*e.g.*, trajectory identification, storage date and presence of azimuth identification) is collected into a catalog component named *Metadata*. This catalog information is maintained through *Trajectory Inventory Service*, layered above all microservices that deal with the distinct data sources. This inventory component abstracts, for functional services, details of which persistence microservice should be used. Hence, applications' requests (*e.g. Functional Services*) are routed to specific persistence services (access trajectories data sources), and results are consolidated based on *Trajectory Metadata*.

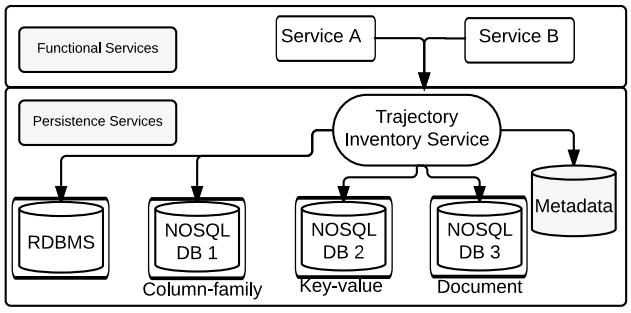

Figure 7: The Trajectory Inventory Service Architecture - adapted from [31]

Each persistence service follows the Data Access Object (DAO) pattern, where Create, Read, Update and Delete (CRUD) operations are implemented by specific components according to underlying databases, minimizing vendor lock-in issues (where customers cannot easily transition to a competitor's product) and providing abstraction from the business rules to data management operations.

This strategy is not suitable for a heavy demand of data join between the data sources, although the Trajectory Inventory Service, based on the Metadata, may pull specific portions of information.

4.5 Canonical Data Model

In this strategy, a central service contains a unifying data model, proposed to ensure consistency and to access underlying data sources in a polyglot persistence scenario, thus facilitating data integration and querying. A Canonical Data Model (CDM) comprehends the semantics and the structure of information, according to rules agreed upon a set of parties [13], simplifying data exchange among services. This is similar to the last strategy, but service compositions here are driven by the CDM.

A CDM is created/composed based on "local" data models, defined according to data sources information. Those models may be defined by different mechanisms and technologies, such as domain ontologies [11] - which may be richer in details, providing high level query mechanisms.

To ensure compatibility among all components, a mediator node abstracts data source details from users' views, translating queries into sub-queries (based on central and local models). Queries are executed on the corresponding data sources. Afterwards, results are integrated to compose the response.

Due to the complexity to represent and maintain all different business scenarios in a single universal model, the usage of multiple CDMs (also referred as bounded contexts) is also discussed in microservices architecture [10, 28]. In this case they would be built as lightweight aggregations, each derived based on a functional domain (some may have common entities), and they evolve as the architecture components grows in maturity and size.

Strenghts

(1) [U1,U2]: APIs based on CDMs can be developed, allowing flexibility of service composition and expressiveness on searches - especially useful for scenarios that demand usage of semantic data.
(2) [C1,C2]: Microservices have "Local" abstractions (models reflected in the CDM) which reduce impact of data source changes for integrations. Also, as those services become integrated to the mediator node, point-to-point integrations are minimized [28], improving interoperability of the solution components. For instance, if ontologies are used [11] and are populated, SPARQL federated queries can be performed, regardless of data source structures.
(3) [M1,P1,P2,P3]: No coupling to specific technologies of distributed services. Local data sources define and maintain infrastructure technologies that best apply to their needs. Local data mapping operations are not coupled to schema mapping. Data manipulation on instances in the database will not affect the corresponding ontology, and development of a source ontology is independent of other sources or ontologies [11].
(4) [M2]: For single CDM approaches, local nodes are natural composition components. In a multiple CDM approach, the same service can be reused (as long as it applies to multiple functional domains).
(5) [M5,M3]: A unified model suggests that data consistency issues can be foreseen and solved during design time.

Weaknesses

(1) [PE1]: Performance optimization for sub-queries (processed on local nodes) and composition of results (by a mediator component) may become complex and inefficient. Ontology load, for instance, may be part of this process and may impact overall performance, *e.g.*, when we try to mitigate ambiguity issues [27]. Any possible transformation of data to be processed in a canonical format may also imply in performance overhead [28].
(2) [M4]: As distributed sources evolve independently, the canonical schema definition may need to undergo changes that may require versioning strategies [28] to deal with synchronization issues.
(3) [R1,R2,R3]: The mediator node needs to take precautions and handle failure or unavailability of information providers.

Related Work - Ghawi and Cullot [11] proposal use ontologies for explicit description of information sources semantics using a *Hybrid Ontology*, where each information source has its local ontology [34], and a global domain ontology is used to represent the whole domain. Hence, each data source-related ontology can be developed independently from other sources' ontologies. The integration task can be simplified, and the addition and removal of sources are flexible.

Additionally, web services are used to facilitate the communication between different components of the architecture, aimed at performing specific tasks, like mapping, querying and visualization. Those components can be developed and deployed independently (given that they are compliant to the generic ontology), and may even have no-relational data structures. Although this strategy does not make explicit references to "microservice" term (coined after this publication), the components (web services) fit the characteristics listed in Section 2.

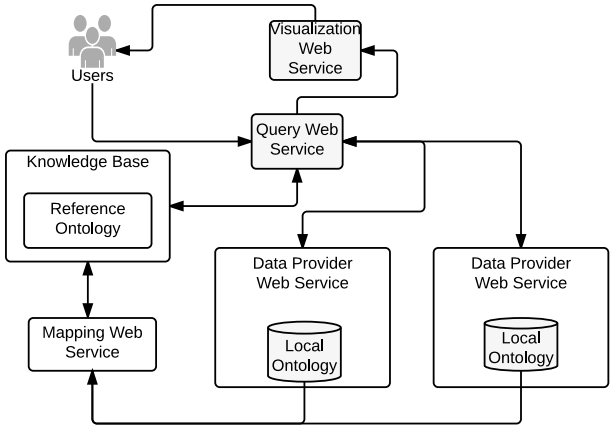

Figure 8: Global Ontology Architecture Components - adapted from [12]

Users submit queries based on the reference ontology. Queries can be executed on heterogeneous and distributed information sources simultaneously. In this solution the following components are detailed:

- *Knowledge Base*: Contains the *Reference Ontology* (*i.e.*, terms, attributes, roles of each local domain in a global model), and mappings between global and local ontologies.
- *Query Web Service*: Analyzes submitted queries expressed in SPARQL language, decomposes submitted information

based on concepts or properties that relate to local ontologies, groups all elements into sub-queries composed of those concepts and delivers each sub-query to the appropriate data provider service. The responses are recomposed in one coherent response, and sent to visualization service.
- *Visualization Web Service*: Transforms query results into an adequate view for the end user.
- *Data Provider Web Service*: Maintains a local ontology, which solves the gap of heterogeneity between data sources, and two additional mapping files: (i) One to perform translations between its information source and the local ontology; (ii) Another one to map from local ontology to reference ontology. Based on those mappings, a data provider service translates local SPARQL sub-queries into database-specific queries and transforms the results in terms of the local ontology, returning to the query service.

The local ontology contains classes and properties, but not the instances. The migration of database instances into ontology instances is performed as needed in response to user queries. The work does not make clear how complex is to implement ontology population in this scenario. In addition, the authors indicate several tools proposed to map database schema into ontologies, and present a tool, named DB2OWL, to automatically create an ontology (expressed in OWL) from a relational database model.

5 EVALUATION DISCUSSION

This section summarizes the quality evaluation of the strategies. Section 3 presented the criteria used in the evaluation. Section 4 presented the analysis of each technology.

For each strategy, ISO/IEC 25010 characteristics were measured by considering the selected sub-characteristics related to it. Positive evaluations associated to each sub-characteristic were counted as 1, otherwise 0 was the assigned value. The formula used to calculate the perceived quality of each characteristic is defined as: $\frac{\sum_{i=0}^{n} Vsc_i}{Nsc}$, rounded off to the second decimal place, given as percentage, where Nsc is the number of related sub-characteristics and Vsc_i is the assigned value for each sub-characteristic.

Table 1 highlights how strategies are perceived to meet ISO criteria (from Section 4). Cases where result was neither 0 (no

Strategies / ISO 25010 Criteria	R	PE	U	C	M	P
1 Shared DB	0	100%	100%	0	0	0
2 CQRS	33%	100%	0	100%	40%	100%
3 Event Data Pump	33%	100%	0	100%	40%	100%
4 Data Retrieval via Service	0	0	100%	0	80%	100%
5 Canonical Data Model	0	0	100%	100%	80%	100%

R - Reliability U - Usability M - Maintainability
PE - Performance Efficiency C - Compatibility P - Portability

Table 1: Strategies evaluation for ISO/IEC 25010 characteristics

sub-characteristics had positive evaluations) nor 100% (all sub-characteristics had positive evaluation) had the following drawbacks (according to Section 4):

- Under Reliability: Fault Tolerance and Recoverability for Strategies 2 and 3;
- Under Maintainability: Analysability, Modifiability and Testability for Strategies 2 and 3; Modifiability for Strategies 4 and 5.

In addition, Table 2 summarizes the mean incidence of positive observations for the characteristics: each amount of positive observations (*Strenghts*) was divided by the number of sub-characteristics (Nsc), further divided by the number of strategies (providing a number between 0 and 1), presented here in percentage terms.

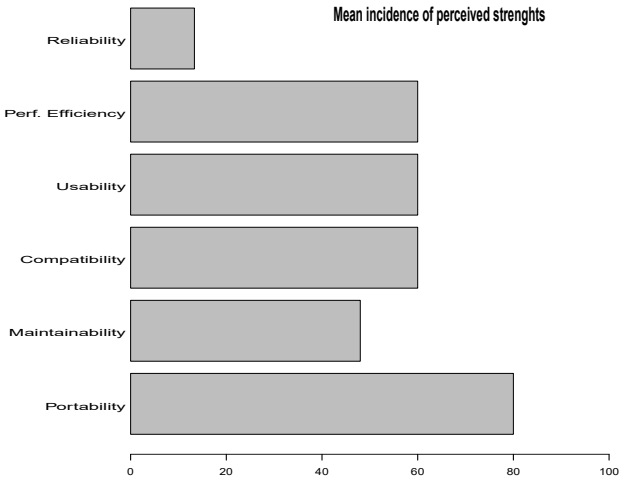

Table 2: Evaluation results - positive observations

Strategy 1 trades low coupling, maintainability and other aspects in favor of performance efficiency. This is strongly affected and dependent on suppliers implementation, thus portability, maintainability and compatibility had 0 as results. Reliability is affected by dependence on the chosen database infrastructure.

Reliability characteristic for the other strategies can be understood under the prism of CAP Theorem. Due to the nature of distributed services, once network partitions are in place consistency (handling distributed data updates) or availability (errors may prevent nodes from being functional) can be compromised. Consistency is evaluated in ISO/IEC 25012 [17], a complementary model for Data Quality. Although not directly analyzed, it relates to the quality of data exchanged between the nodes, thus affected by all three sub-characteristics of reliability (e.g. issues arising from distributed updates). Availability is a sub-characteristic of reliability.

Results for Reliability, Performance and Maintainability are based on factors that are natural costs of microservices [8]:

- *Reliability*: Remote calls are likely to fail as the number of microservices grow. One approach to handle failure cases is through asynchronous collaboration (adopted in strategies 2 and 3), where updates are isolated and data replication components assist on querying activities - which requires extra complexity of figuring out the consequences of failure for every remote call. Another option, on strategies 4 and 5,

is to embrace the issues and deal with eventual consistency failures - which may become even more complicated.
- *Performance*: Remote service calls tend to be slow. Through queuing mechanisms, strategies 2 and 3 stores collaborating service data, providing consumer upfront access to updated information (yielding better performance metrics). Another approach is increasing the granularity of service calls (strategies 4 and 5), improving reuse and inter-service communications, although this can complicate design and programming tasks, and may introduce latency.
- *Maintainability*: It is assumed low coupling and high cohesion design options facilitates maintainability. The infrastructure and controls required to put in place strategies 2 and 3 (such as message queue or other middleware components) were also considered in this analysis.

The other aspects (Usability, Compatibility and Portability) were evaluated according to (again in the authors' perception) architectural aspects, considering clarity of the overall design of the components and its benefits.

6 CONCLUSION

There are important challenges for querying data on polyglot persistence in microservices architecture. Several studies point to different strategies to overcome them, while balancing between service availability and data consistency.

Five strategies were identified in the literature. Each one was detailed and evaluated to highlight potential benefits and drawbacks based on to ISO 25010 characteristics of software quality. Besides, examples of implementations for those strategies were presented.

This work contributes to make clear the design decisions on each strategy, highlighting their strengths and weaknesses, and reporting eventual limitations so that it can be observed by researchers and practitioners when evaluating microservices solutions.

This analysis indicates there is no optimal generic approach, but best strategies applicable for specific scenarios. All strategies present at least one downside, and at least one strategy indicated a positive evaluation related to each characteristic. Reliability appears as the most critical aspect.

As future work, we propose to explore lessons learned from successful implementations of microservices in industry and academia, such as Netflix and Uber [26]. By evaluating a higher number of solutions and studies, there may be an opportunity to infer best practices and standards to guide architectural designs. We also plan to develop new approaches or evolve current ones to overcome known limitations.

REFERENCES

[1] A. Adewumi, S. Misra, and N. Omoregbe. Evaluating open source software quality models against iso 25010. In *CIT/IUCC/DASC/PICOM, 2015 IEEE International Conference on*, pages 872–877. IEEE, 2015.

[2] B. Behkamal, M. Kahani, and M. K. Akbari. Customizing iso 9126 quality model for evaluation of b2b applications. *Information and software technology*, 51(3):599–609, 2009.

[3] E. A. Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[4] T. Erl. *Service-oriented architecture (SOA): concepts, technology, and design*. Prentice Hall, 2005.

[5] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[6] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer Science & Business Media, 2014.

[7] M. Fowler. Cqrs. *Command Query Responsibility Segregation, https:// martinfowler. com/ bliki/ CQRS.html*, 2011.

[8] M. Fowler. Microservice trade-offs. *URL: http:// martinfowler.com/ articles/ microservice-trade-offs.html*, 2015.

[9] M. Fowler. Polyglot persistence. 2015. *URL: https:// martinfowler.com/ bliki/ PolyglotPersistence.html*, 2015.

[10] M. Fowler and J. Lewis. Microservices. *ThoughtWorks. https:// martinfowler.com/ articles/ microservices.html*, 2014.

[11] R. Ghawi and N. Cullot. Database-to-ontology mapping generation for semantic interoperability. In *Third International Workshop on Database Interoperability (InterDB 2007)*, volume 91, 2007.

[12] R. Ghawi and N. Cullot. Building ontologies from xml data sources. In *Database and Expert Systems Application, 2009. DEXA'09. 20th International Workshop on*, pages 480–484. IEEE, 2009.

[13] M. Gilpin. From the field: The first annual canonical model management forum. forrester blogs. *URL: https:// go.forrester.com/ blogs/ 10-03-15-from_the_field_the_first_annual_canonical_model_management_forum*, 2015.

[14] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[15] ISO/IEC. Iec 9126 software engineering, product quality, part 1: Quality model. 2001. *Geneva: International Organization for Standardization*.

[16] ISO/IEC et al. Iec25010: Systems and software engineering–systems and software quality requirements and evaluation (square). *International Organization for Standardization*, 34:2910, 2011.

[17] ISO/IEC et al. System and software quality models. *ISO/IEC 25010:2011, https:// www.iso.org/ standard/ 35733.html*, 2011.

[18] J. Kabbedijk, S. Jansen, and S. Brinkkemper. A case study of the variability consequences of the cqrs pattern in online business software. In *Proceedings of the 17th European Conference on Pattern Languages of Programs*, page 2. ACM, 2012.

[19] S. Leberknight. Polyglot persistence. *Near infinity[Online], Available: http://www. nearinfinity. com/blogs/scott_leberknight/polyglot_persistence.html*, 2008.

[20] D. Maron, J. Torquato, and P. C. Alves. Oracle distributed database (in portuguese). *Universidade Federal da Bahia, Salvador*, 10, 2011.

[21] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[22] A. Messina, R. Rizzo, P. Storniolo, and A. Urso. A simplified database pattern for the microservice architecture. *DBKDA 2016 : The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications*, 2016.

[23] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 2016.

[24] S. Newman. *Building microservices*. O'Reilly Media, Inc., 2015.

[25] E. J. O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356. ACM, 2008.

[26] A. Panda, M. Sagiv, and S. Shenker. Verification in the age of microservices. 2017.

[27] G. Petasis, V. Karkaletsis, G. Paliouras, A. Krithara, and E. Zavitsanos. Ontology population and enrichment: State of the art. In *Knowledge-driven multimedia information extraction and ontology evolution*, pages 134–166. Springer-Verlag, 2011.

[28] T. d. Preez. Canonical data models and microservices. *URL: https:// www2.deloitte.com/ content/ dam/ Deloitte/ za/ Documents/ strategy/ ZA_Deloitte_Digita_Canonical_Schemas.pdf*, 2016.

[29] P. Rajković, D. Janković, and A. Milenković. Using cqrs pattern for improving performances in medical information systems. *BCI'13 September 19-21*, 2013.

[30] P. J. Sadalage and M. Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.

[31] G. Scheibel. A software architecture for storing trajectories using polyglot persistence (in portuguese). *Master dissertation, University of the State of Santa Catarina*, 2016.

[32] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990.

[33] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh. Synapse: a microservices architecture for heterogeneous-database web applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 21. ACM, 2015.

[34] H. Wache, T. Voegele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information-a survey of existing approaches. In *IJCAI-01 workshop: ontologies and information sharing*, volume 2001, pages 108–117. Seattle, USA, 2001.

[35] Z. Xiao, I. Wijegunaratne, and X. Qiang. Reflections on soa and microservices. In *Enterprise Systems, 4th International Conference on*, pages 60–67. IEEE, 2016.

[36] O. Zimmermann. Microservices tenets. *Computer Science-Research and Development*, pages 1–10, 2016.