A Distributed Database System for Event-based Microservices

Rodrigo Laigner
University of Copenhagen
Copenhagen, Denmark
rnl@di.ku.dk

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

Marcos Antonio Vaz Salles
University of Copenhagen
Copenhagen, Denmark
vmarcos@di.ku.dk

ABSTRACT

Microservice architectures are an emerging industrial approach to build large scale and event-based systems. In this architectural style, an application is functionally partitioned into several small and autonomous building blocks, so-called microservices, communicating and exchanging data with each other via events.

By pursuing a model where fault isolation is enforced at microservice level, each microservice manages their own database, thus database systems are not shared across microservices. Developers end up encoding substantial data management logic in the application-tier and encountering a series of challenges on enforcing data integrity and maintaining data consistency across microservices.

In this vision paper, we argue that there is a need to rethink how database systems can better support microservices and relieve the burden of handling complex data management tasks faced by programmers. We envision the design and research opportunities for a novel distributed database management system targeted at event-driven microservices.

CCS CONCEPTS

• **Information systems → Database management system engines**.

KEYWORDS

microservices, event-driven architecture, database system

ACM Reference Format:
Rodrigo Laigner, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2021. A Distributed Database System for Event-based Microservices. In *The 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21), June 28-July 2, 2021, Virtual Event, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3465480.3466919

1 INTRODUCTION

Modern business scenarios require systems to make decisions in real-time based on events. To tackle such scenarios, event-driven architectures (EDAs) are often advocated as a compelling approach to meet the stringent requirements required by data-intensive systems that react to events [27]. An EDA is composed by highly decoupled, single-purpose event processing components that asynchronously receive and process events, each performing a singular task in the application [23]. A particular approach to realizing an EDA is through the *microservice architectural style* [16], an emerging industry paradigm to design highly-modular and scalable applications [28]. In microservices, an application is designed as a set of small and independent building blocks that communicate with each other via pre-defined interfaces (e.g., HTTP APIs) or events. Each microservice may manage its own database, and thus select its underlying technology to best support the data formats and workloads for the computations encoded. Thus, microservices follow a *decentralized data management* principle. In line with this principle, the dominating practice is that microservices and their underlying databases are deployed in separated containers to preclude errors from propagating across microservice boundaries.

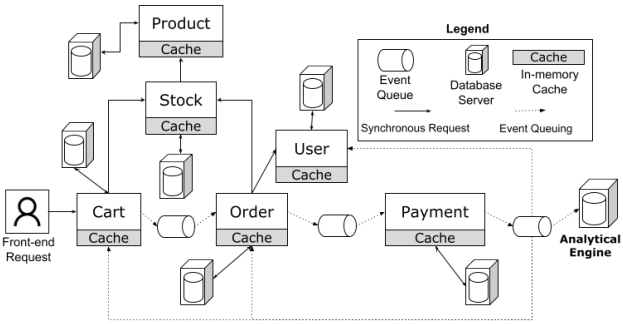

Figure 1: E-commerce microservice architecture example

Motivating Example. Microservice applications deviate from traditional monolithic transaction processing, as illustrated by the e-commerce application shown in Figure 1. After adding several items to a cart, where a cart is an entity managed by the *cart* microservice, the customer may initiate the order's payment process. From here, different options could apply, e.g.: (i) for each cart item added, the *cart* microservice acquires a temporal (i.e., expiring) lock, a promise from the *stock* microservice that the stock item is safeguarded from being acquired by other clients; (ii) the cart items are added arbitrarily (i.e., without any synchronization) and the *order* microservice is responsible for checking against the *stock* microservice the availability of cart items by the time a checkout request is issued by the user.

In either case, the *stock* microservice may verify whether the products are available and whether mismatches exist (e.g., in the product price). After confirming that all items are in stock, the order is then confirmed. However, prior to proceeding with payment, the *order* microservice retrieves user information from the *user* microservice and checks the validity of the user's discounts (e.g., in case of no longer valid discounts applied previously). After applying all the proper updates (e.g., calculating totals) to the customer order, the *order* microservice emits an event so that the *payment* microservice can proceed with the payment confirmation.

DEBS '21, June 28-July 2, 2021, Virtual Event, Italy

ACM ISBN 978-1-4503-8555-8/21/06…$15.00
https://doi.org/10.1145/3465480.3466919

The *payment* microservice processes the payment by acquiring the confirmation of funds from the credit card holder (e.g., external system, often a blocking operation), and afterwards updates the user's credit score by contacting the *user* microservice. Lastly, an analytical engine is eventually updated with the new order.

Despite the apparent attractiveness of a loosely-coupled design and an inherently decentralized data management architecture, the presented application unavoidably requires substantial coordination across microservices, thus forcing developers to deal with challenges that would not arise in a traditional monolithic architecture. In particular, by following a model reminiscent of BASE [22] and OLEP [15], microservice developers end up encountering challenges that should have been solved by database systems. For instance, these models prescribe decomposing a schema and associated application logic into functional partitions, hard-wiring functional dependencies through asynchronous events. As result, these models force a substantial amount of data management tasks into the application-level, such as enforcing constraints cutting across several microservices (e.g., referential constraints) and ensuring liveness, since a transaction is often broken down into several steps due to the functional decomposition as in our example. Furthermore, microservices introduce challenges not originally envisioned by the BASE and OLEP models, which we discuss next in the context of our example application.

Cross-microservice synchronization and validations. The substantial data management logic encoded in the application-tier creates a barrier for enforcing application safety across microservices. Developers are offered neither efficient nor intuitive interfaces [10] for encoding distributed synchronization (e.g., locks and leases) at the application level correctly. As a result, microservice developers often end up resorting to eschewing synchronization altogether, leading to data consistency issues. For instance, in *eShopContainers* [1], the application we based our example scenario on, an asynchronous event generated by the *payment* microservice (after payment has been processed) triggers the removal of items from stock. However, the application code does so unsafely, i.e., races and failures can lead to inconsistent state being recorded across microservices.

Event-based constraint enforcement. Built from our example in Figure 1, consider the case where a *product-price-update* event is emitted concurrently with a *checkout-cart* event request. The order in which the *cart* microservice should process these concurrent events must depend on an ordering constraint, otherwise any arbitrary order may apply. However, programmers have no way to specify event processing order invariants related to the possible interleaving of event streams and end up encountering challenges to guarantee consistency.

Cross-microservice queries. Queries spanning multiple microservices are popular in microservice architectures [6]. To implement such queries, programmers often encode data processing logic for aggregating and joining data from different microservices at the application-level. By resorting to such ad-hoc mechanisms, programmers are exposed to a myriad of anomalies, such as fractured reads [3]. Besides, faulty microservices are another challenge when retrieving data from multiple microservices. Here, the impossibility of accessing a microservice's private state may lead to an incomplete view of the state.

For instance, based on our example, suppose an analyst necessitates a report aggregating the last day's worth of historical data from users, their orders, and discounts. The lack of a principled approach for state management across microservices forces developers to resort to ad-hoc and error-prone mechanisms to query and join such data.

Data replication. Data replication is usually employed to alleviate the amount of requests required in queries spanning multiple microservices. Through propagated events, the OLEP model predicates that microservices can incrementally maintain materialized views composed by data owned by other microservices. However, given the heterogeneity of microservice databases, it is often the case that practitioners resort to *ad-hoc* application-level mechanisms to support data replication through events, a complex and error-prone approach that leads developers to make do with only eventually consistent views. In our example, suppose the *Order* microservice maintains a materialized view of users' discounts asynchronously updated via events. Weak replication semantics may lead to issuing an incorrect discount to an user.

Fault tolerance. Consider the case when a business transaction performs writes to a microservice's private state and queues an event into a message broker (to trigger an operation in another microservice, as observed in the *order* microservice). In these cases, the BASE model advocates that all writes must be performed in the same resource (i.e., data store) [22] to avoid a distributed commit protocol. However, it is often the case such support is not available, thus leading developers to resort to error-prone fault-tolerance strategies at the application-level [26].

Contributions and Outline. The paper is organized as follows. Section 2 discusses the limitations of state-of-the-art database systems in light of the data management challenges described. Section 3 presents our contributions: (i) We argue that although traditionally applications have been treated as black boxes, such a paradigm is insufficient for addressing the data management challenges in microservices, as it offers no way to expose the complex data interplay outside the database among microservices. We thus advocate for identifying such complex interactions and data management tasks taking place at the application-tier and then pushing them down for database processing; (ii) We propose a novel abstraction called *virtual microservices*, to represent the computations performed by microservices inside the database; (iii) We present the declarative constructs to allow for the identification of virtual microservices at the application-level, and; (iv) We present a vision for a microservice-oriented event-driven database system. Section 4 discusses challenges and research avenues for this novel database approach and Section 5 concludes the paper.

2 STATUS QUO AND LIMITATIONS

2.1 State-of-the-art database systems

Although it is possible to observe a variety of architectural designs in classic databases, separation of database applications is enforced at schema-level. This can be considered a weak functional isolation scheme since the performance degradation of a application may impact other applications. Besides, whenever distinct applications need to share data, it is assumed to take place within the database-tier, which does not meet the microservices' state of the practice.

Cloud-native multi-tenant databases [4] logically isolate tenants and provide elastic resources backed by the cloud infrastructure. However, they fail to support event-based programming and advanced data management requirements, such as cross-tenant data replication and computations. By assuming tenants are completely isolated, these systems cannot capture dependencies and interactions amongst microservices.

Taking a step back, there has been a tension yet not properly addressed by the database systems community between the needs and requirements of developers in the wild and the classic database abstractions, which treat the application side as a black box. This view clashes with the needs of programmers that prefer encoding their complex business logic in the application-tier, which often relies on application-level feral validations [2]. This tension is worsened by the emergence of distributed applications that take advantage of the flexibility and cost-effectiveness offered by the cloud. In this case, computations are no longer being held based on a single database, but rather traverse several small building blocks of the application, often making use of a myriad of data systems, such as caching or pub/sub systems, to meet data management challenges. A holistic solution incorporating selected data management functions of multiple such building blocks is required to fully address the needs of microservice applications.

2.2 Stream processing systems

Usually framed as a compelling abstraction for microservices [13, 25], stream processing systems (a.k.a. dataflow systems) are designed to perform continuous queries over unbounded data streams [13]. The computational model of stream processing engines contrasts with microservices, since microservices are often independently developed by different teams and deployed separately, instead of within a single streaming processing engine, to provide strong isolation. At the same time, microservices are free to communicate, often through non-blocking primitives, and operate over data from other microservices. Besides, failures or changes in a microservice should not propagate over or interrupt other building blocks of the system, which contrasts with existing stream engines.

It has been recently argued that microservice applications can be built on streaming dataflow systems by making stream processors full-fledged data management engines (e.g., by supporting for transactions across microservices) [13]. However, it remains an open question how to match static dataflow graphs prescribed by such a solution with the loose-coupleness, autonomy, and dynamicity principles of microservices [28].

2.3 Function as a Service

Although recent advances in serverless computing through the function as a service (FaaS) API [12] aim at offering an easy-to-use platform that provides programmers high-level computation expressibility and automatic resource management, FaaS is usually perceived as a fit for more stateless and less stringent data-intensive computations.

By contrast, one may position stateful functions as a proper abstraction for microservices, since they provide an API for state management that is particular to the business logic encapsulated by the function [25]. However, it remains an open question how stateful functions could support advanced data management features required in microservices, e.g., ordering constraints in complex interleaving of data streams, online queries, and cross-microservice synchronization and validations.

2.4 Frameworks for distributed applications

Orleans. While Orleans can be used to develop stateful middle-tier applications like microservices, for not being a full-fledged database system, it still forces developers to reason about application safety at the application level, such as the impact of the interleaving of events to private state and explicitly handling data durability concerns. Furthermore, applications must respect a strict set of characteristics to benefit from the Orleans paradigm [21]. In this sense, it is unclear how microservices, such as the example scenario, can be modeled through virtual actors [27].

Dapr. Dapr is a framework for facilitating the development of microservice applications [19]. By exposing a standard API for microservices to connect to the Dapr middleware, Dapr is able to intermediate message queuing across microservices. However, practitioners are still forced to deal with the aforementioned challenges at the application-level. Most importantly, Dapr offers a centralized and homogeneous (i.e., key-value) state management abstraction, contrasting with the prescribed data sovereignty of microservices.

3 MICROSERVICE-ORIENTED DATABASES

3.1 The gist

Although existing data systems partially support real-world microservice deployments, advanced data management features required by microservices are not explicitly or sufficiently addressed in conjunction. Particularly, application semantics are mostly unknown to the database, which hinders the database from being able to capture data management tasks encoded at the application level.

Given this clear impedance, we advocate for rethinking how database systems interact with this growing class of applications. We hypothesize that through appropriate abstractions, we can proactively identify and push data management functionality down to the database and provide built-in advanced data management support directly to the application.

Centralization vs. Decentralization. In order to achieve the necessary isolation between microservices, the conventional wisdom advocates a decentralized data governance paradigm, which is often implemented by using the database-per-service pattern. This paradigm is the root cause to the aforementioned data management challenges of microservices. To enable pushing down data management tasks to the database system, in contrast to the conventional wisdom, we propose a central data governance paradigm for microservices, i.e. using a single scalable and distributed database system to manage the states of all microservices. We argue that such a paradigm does not necessarily contradict the decentralized data management principle of microservices, as long as the developers are able to express the logical boundaries of each microservice, and the database system can consistently enforce such boundaries. As demonstrated by the success of multi-tenant database systems, which offer data management services to independent and isolated tenants through a central database, we believe that providing fault-isolation, performance isolation, and data sovereignty guarantees to microservices is independent from the database being centralized

or decentralized. Furthermore, given the recent developments of HTAP and Polystore database systems, which are able to respectively cope with heterogeneous workloads and data formats, we observe that a centralized data governance paradigm can be made orthogonal from the database system being able to manage multiple underlying databases.

3.2 Virtual microservices

In order to push down data management tasks to the database system, a proper abstraction needs to faithfully characterize the semantics of microservice applications inside the database. This tension leads us to the following question: **How can we map a microservice application's invariants and data management tasks to an internal representation that the database can effectively manage?**

To address this question, we envision the notion of virtualized microservices as the core building blocks of a database architecture. In other words, each microservice application ought to have an abstract representation in the database, a *virtual microservice* twin.

More precisely, a **virtual microservice** is a construct that logically encapsulates the state of a particular microservice, along with its constraints and data dependencies, as well as abstracts inbound and outbound event streams. In other words, by exposing internal representations of microservices, the database system gains knowledge about the event dataflow across microservices in addition to the constraints that cut across microservices. As a result, *the application is no longer a black box to the database.*

As an idealized microservice twin, a virtual microservice is not only an independent entity, but also an internal representation of the application managed by the database. Thus, it inherits the same characteristics of a microservice, which must then be enforced consistently in the database, namely: (i) communication by event-based asynchronous messages; (ii) private mutable state; (iii) shared data that is not mutable. The objective is to natively support the data management tasks encoded in microservices within the database, but at the same time provide features that normal microservices currently do not have, e.g., explicit data dependencies, integrity constraints, and atomic actions across private state and event streams.

3.3 A cross-stack architectural vision

The advanced data management requirements (§ 1) and shortcomings found in state-of-the-art database systems (§ 2.1) pose significant challenges to effectively supporting data-intensive microservices. It is unclear how a database system can jointly support event-based querying APIs, cross-microservice queries, event-based constraint enforcement, high consistency in data replication across microservices, and proper cross-microservice synchronization, while at the same time providing isolation boundaries between microservices.

To address this conundrum, we now turn our attention to our **success criteria**: *enabling pushing data management tasks to the database and materializing the virtual microservice abstraction into a principled database architecture that tackles the challenges of data management in microservices by design.* Our vision is shown in Figure 2, which depicts an event-driven microservice-oriented database architecture. Broadly, the architecture provides an application framework, which controls the interaction of the application-tier

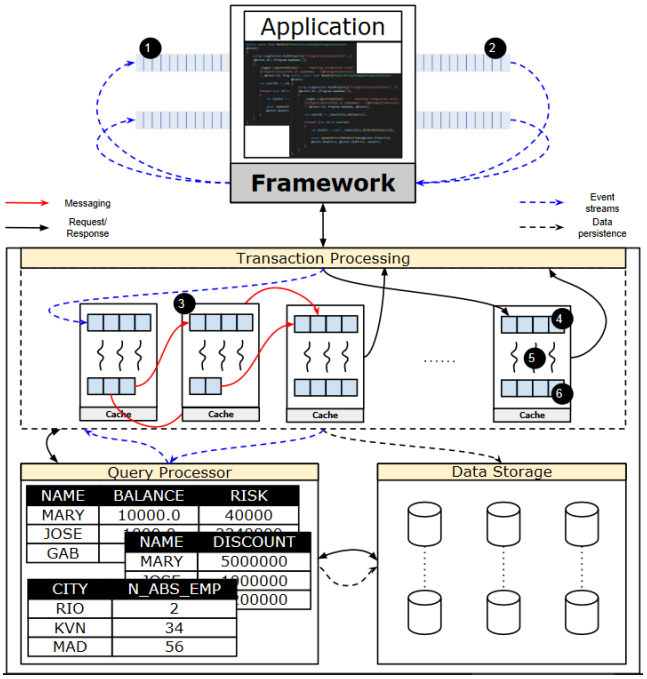

Figure 2: A microservice-oriented event-driven database

with the system, and a set of well-defined components that target particular data management concerns and can scale independently.

We explain in the following the interconnection between the components, how the key insight and abstraction are enabled, and how the pressing challenges are met.

Application abstraction. To enable pushing data to the database, it is necessary to enrich the abstraction provided to the application to allow for identifying the complex data management tasks encoded. A simple but powerful tool that has been historically used to add functionalities seamlessly to applications is a framework [8]. Frameworks provide a dynamic introduction of behavior in the application without the need for user-defined code.

A framework is a key-enabler not only because it is responsible for identifying data management logic and informing the database about the application semantics concerning complex data management tasks, but also due to its ability to control the application [8], a necessary condition to enforce event-based constraints and to control the scheduling order of functions.

We envision such a framework should provide programming constructs in sync with the state-of-practice. We use as inspiration the Java Persistence API [20] and declarative transaction implementation in industry-strength web frameworks like Spring [9] to demonstrate our framework constructs, shown in Listing 1, but the approach is generalizable to other programming languages.

In our example, the *Cart* microservice is equipped with a sufficient abstraction to safeguard a given item is correctly locked (lines 2 and 10). The safety guarantee is encapsulated in a *query* directive that is pushed down for database enforcement. In line 18, we exhibit an order dependence, a necessary condition to safeguard consistency guarantees on the total price of an order. The order is enforced both by the database (in terms of concurrent operations) and the framework (at the application-level).

```
// Stock microservice
@Lease("FROM Stock s WHERE s.id = :itemId FOR UPDATE
     DECREMENT s.qtd BY :qtd WITH PERIOD :lease")
public Optional<Item> getStockItemInfo(String itemId, int
     qtd, Lease lease);

// Cart microservice
@Inbound(event=AddCartItemRequest)
@Outbound(event=AddCartItemAttempt)
@Transactional(type="RW", isolation="serializable")
public void handleAddCartItem(String customerId, String
     itemId, int qtd) {
    Optional<Item> item = getStockItemInfo(itemId, qtd,
         Lease.byHours(1));
    if (item.isPresent()){
        Cart cart = findCartByCustomer(customerId);
        cart.add(new CartItem(itemId, qtd));
    }
    send(AddCartItemAttempt.build(customerId, qtd,
         item.isPresent()));
}

@Inbound(event=ProductPriceUpdate,
     precedence=CheckoutRequest)
@Transactional(type="RW", isolation="snapshot")
public void handlePriceUpdate(Event productPriceUpdate)
// code omitted, handles updates to cart items

// Order microservice
@Query("FROM CustDiscounts cd, Customer c JOIN c.id =
     cd.c_id WHERE c.id = :custId AND cd.disc_id IN
     (:discounts[id]) AND cd.expired() <> FALSE")
List findDiscounts(List discounts, String custId);

@Inbound(event=CheckoutStarted)
@Outbound(event=OrderPlaced)
@Transactional(type="RW", isolation="serializable")
public void processOrder(Checkout checkout) {
    List discounts = findDiscounts(checkout.discounts,
         checkout.custId);
    if (discounts.size() != checkout.discounts.size()) {
        // adjust total price
    }
    /* perform necessary data integrity checks, including
         cart items' leases, and build new Order's object */
    send(OrderPlaced.build(order));
}
```

Listing 1: Example abstractions provided to the application

Lastly, we demonstrate a compelling abstraction to allow for API-oriented encoding of directives in defined queries. Line 24 shows the API provided by the *User* microservice ("expired()") that encapsulates the business logic regarding the expiration of a discount (defined as a query in the producer side). Flexibility is provided such that data management logic does not need to be hard-coded in the consumer side. The query traverses the Order's microservice without the need to encode error-prone, synchronous calls with weak isolation, since the request is in fact shipped to the database.

Pushing data to the database. We now describe how the encoded data management tasks are recognized and pushed down to the database. In Figure 2, a microservice in an application is encoded by its application logic encapsulated in: (a) functions; (b) logical input and output queues (**#1** and **#2**, respectively); (c) invariants (e.g., unique and foreign keys); and (d) external data dependencies.

At start-up time, the framework, by using AOP [14] for example, can proactively identify data management tasks encoded in annotation directives, package them into contextual information representing the microservice, and inform the database. The database then builds an internal representation of the microservice, a virtual microservice, and coordinates with the framework when the register operation has succeed. Then, the framework is able to start to react to and push events, as well as push tasks downward.

Transaction processing. We now describe the entry point of our database, the transaction processing system, a distributed transaction executor responsible for handling concurrency control and event ordering constraints by managing a virtual representation of microservices. This system targets fulfilling the challenges related to cross-microservice synchronization and validation, event-based constraints enforcement, and strong isolation.

Mapping virtual microservices to computational resources. We envision the database being deployed in both single multi-core machine to be used in small to moderate scale scenarios, and in distributed settings, deployed in a cluster of machines to cope with large scale microservice applications. To support the mapping of computational resources and performance-and-fault isolation, the component will leverage container-based deployments [5]; these need, however, to be adapted to respect virtual microservice isolation boundaries. We leave further details about a virtual microservice execution model for future work, but we believe one would not deviate from the following: *(a)* Input queue of requests (**#4**); *(b)* Scheduler, to map an input event to a transaction worker; *(c)* Transaction workers, for processing transactions and interacting with the query processing component whenever necessary (**#5**); *(d)* Private mutable state; and *(e)* an output queue (**#6**).

Query processing. We envision a distributed query processor to tackle three principled challenges (§ 1): (a) the impossibility of accessing a microservice's private state when the service is down; (ii) to refrain developers from encoding *ad hoc* data replication error-prone mechanisms on application-tier; and (iii) to effectively allow for cross-microservice queries through system-level support, including appropriate consistency guarantees.

Specifically, the strong modularity and the prevalence of short update transactions within a microservice's private state (that may present increased contention) suggests that microservices are better served by specific-purpose transaction workers that share no state and resources across virtual microservices. On the other side, given the ubiquity of accesses to several microservices' private states in online queries, sharing resources while still respecting the strong isolation principle may decidedly improve performance. The design of the query processing component will balance these competing trade-offs to incorporate the event-based processing nature of microservices.

Persisting data. This layer is a distributed storage system aimed to store microservices' data and query results. We assume the storage exposes proper interfaces for handling blocks of data that will be processed by the upper layers, namely, the transaction processing and query processing subsystems. The reasoning for a decoupled

storage alternative is to allow for flexibility in plugging different storage solutions as well as to offer opportunities to adapt dynamically to varied workloads often found in microservice architectures. We leave further details about data persistence to future work.

4 OPPORTUNITIES AND CHALLENGES

Holistic coordination. Through the virtual microservice abstraction, a dataflow of microservices' interactions can be derived based on the defined inbound and outbound events and the data dependencies that cut across microservices. Treating cross-microservices constraints as event requests within the database opens up opportunities to enforce a processing order across virtual microservices that would conflict otherwise, discharging the virtual microservices to engage in coordination.

Asynchronous code. The abstraction presented in this work expects asynchronous event sending to take place at the end of a procedural function, which is a condition that satisfies most data-intensive microservices [7, 17]. However, we are witnessing an increasing interest in the use of *asynchronous function calls* that return promises [18]. Asynchronous calls allow for increased opportunities of parallelism and non-blocking application logic inside the database [24]. Investigating the integration of asynchronous primitives for data management logic and virtual microservices is a worthy avenue to pursue.

Application-aware concurrency control. We envision the definition of isolation levels that are most appropriate for the underlying microservice's computation. In our example (Listing 1, line 19), the reasoning is that for client-driven requests, it is assumed that cart items that are already locked face no concurrency issues with requests from other clients, which makes serializable isolation not strictly necessary in this case. At the same time application-defined consistency semantics creates opportunistic window for performance, it creates challenges for concurrency control design within the database.

5 CONCLUSION

The data management challenges brought about by microservices necessitate that we rethink the traditional database architecture. With state-of-the-art abstractions, database systems in microservice architectures are unaware of the significant data flowing outside of the database [11] and hence are condemned to play a secondary role, mostly relegated to only providing data durability.

In this work, we make a case for event-driven microservice-oriented databases. By pushing down virtualized representations of microservices into the database, the database system is able to natively support all benefits pursued by practitioners on adopting microservices, e.g., strong isolation, data ownership, and autonomy, but at the same time offer advanced data management features practitioners currently lack in state-of-the-art database systems.

Realizing this new class of database systems opens up a wealth of research opportunities, including – but not limited to – holistic virtual microservice coordination, asynchronous in-database programming, and application-aware concurrency control.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie agreement No 801199 and Independent Research Fund Denmark grant No 9041-00368B.

REFERENCES

[1] .NET Application Architecture Reference Apps. [n.d.]. *eShopOnContainers*. https://github.com/dotnet-architecture/eShopOnContainers
[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1327–1342.
[3] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2016. Scalable atomic visibility with RAMP transactions. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 1–45.
[4] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[5] docker docs. 2021. *Run multiple services in a container*. https://docs.docker.com/config/containers/multi-service_container (Accessed on 2021-03-15).
[6] B2W Engineering. 2018. *restQL: Tackling microservice query complexity*. https://medium.com/b2w-engineering-en/restql-tackling-microservice-query-complexity-27def5d09b40 (Accessed on 2021-02-28).
[7] Uber Engineering. 2020. *Revolutionizing Money Movements at Scale with Strong Data Consistency*. https://eng.uber.com/money-scale-strong-data (Accessed on 2021-03-08).
[8] Martin Fowler. 2005. *InversionOfControl*. https://martinfowler.com/bliki/InversionOfControl.html (Accessed on 2021-03-19).
[9] Spring Frawework. 2021. *16. Transaction Management*. https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html (Accessed on 2021-03-02).
[10] Pat Helland. 2017. Life beyond distributed transactions. *Commun. ACM* 60, 2 (2017), 46–54.
[11] Pat Helland. 2020. Data on the Outside vs. Data on the Inside, Vol. 18. ACM Queue. Issue 3.
[12] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. arXiv:1812.03651 [cs.DC]
[13] Asterios Katsifodimos and Marios Fragkoulis. 2019. Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications.. In *EDBT*. 682–685.
[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*. 220–242.
[15] Martin Kleppmann, Alastair R. Beresford, and Boerge Svingen. 2019. Online Event Processing: Achieving Consistency Where Distributed Transactions Have Failed. *Queue* 17, 1 (2019), 116–136.
[16] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sergio Lifschitz, and Yongluan Zhou. 2020. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. In *46th Euromicro Conference on Software Engineering and Advanced Applications*. 213–220.
[17] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. arXiv:2103.00170 [cs.DB]
[18] B. Liskov and L. Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *SIGPLAN Not.* 23, 7, 260–267.
[19] Microsoft. 2021. *Dapr*. https://github.com/dapr/dapr
[20] Oracle. 2021. *The Java EE 6 Tutorial Part VI Persistence*. https://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html (Accessed on 2021-03-02).
[21] Orleans. [n.d.]. *Best Practices*. https://dotnet.github.io/orleans/docs/resources/best_practices.html (Accessed on 2021-05-16).
[22] Dan Pritchett. 2008. Base: An Acid Alternative. In *File Systems and Storage*, Vol. 6. ACM Queue. Issue 3.
[23] Mark Richards. 2015. *Software Architecture Patterns* (1st ed.). O'Reilly.
[24] Vivek Shah and Marcos Antonio Vaz Salles. 2018. Reactors: A Case for Predictable, Virtualized Actor Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*. 259–274.
[25] Tzu-Li (Gordon) Tai. 2020. *Stateful Functions Internals: Behind the scenes of Stateful Serverless*. https://flink.apache.org/news/2020/10/13/stateful-serverless-internals.html (Accessed on 2021-03-05).
[26] GitHub user arielmoraes. 2018. *#700*. https://github.com/dotnet-architecture/eShopOnContainers/issues/700 (Accessed on 2021-03-22).
[27] Yiwen Wang, Julio Cesar Dos Reis, Kasper Myrtue Borggren, Marcos Antonio Vaz Salles, Claudia Bauzer Medeiros, and Yongluan Zhou. 2019. Modeling and Building IoT Data Platforms with Actor-Oriented Databases. In *EDBT*. 512–523.
[28] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (2017), 301–310.