VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

PROGRAMME IN SOFTWARE ENGINEERING

Investigation of monolith systems data migration to microservice architecture

Monolitinių sistemų duomenų migravimo į mikro servicinę architektūrą tyrimas

Bachelor work

Done by 4 course 2 group student

Deividas Lukas Tilindis (signature)

Supervised by: Lekt. Vasilij Savin (signature)

Vilnius – 2023

Contents

Abstract....4
Santrauka....5
Object of research....6
Objective....6
Tasks....6
1. Microservice architecture overview....8
1.1. What is microservice architecture?....8
1.2. How these microservices interact with each other? Ecosystem of microservices.....8
1.3. Challenges of distributed system....8
2. Data ownership and consistency....10
2.1. Who owns what data in microservice architecture....10
2.2. Data consistency....11
2.3. Data migration....12
3. Analysis of data migration challenges, methods and patterns....13
3.1. Database schema is used by other apps....13
3.2. Splitting the schema and unsplittable tables....14
3.3. How to transfer ownership of data, data synchronization....15
3.4. Data consistency and how to achieve it.....17
4. Practical illustration....19
4.1. System under analysis....19
4.2. Technical flaws....20
4.3. Design flaws and their solutions....22
4.3.1. ORDER_DETAIL table and it's multiple responsibilities.....22
4.3.2. Suspected overburden of card details in PAYMENT_INFO table....28
4.4. Discount order detail migration plan....29
5. Conclusions and results....33
5.1. Results....33
5.2. Conclusions....34
6. References....35
7. Appendices....37
7.1. Appendix 1. Initial POS auditing database schema....37
7.2. Appendix 2. Database schema after technical flaw eradication....40
7.3. Appendix 3. Database schema after ORDER_DETAIL split....43

7.4. Appendix 4. Storage space optimization calculations. 47
7.5. Appendix 5. Pseudo API endpoints and messaging event 48

Abstract

Microservice architectural pattern has been a discussed and used in IT industry for quite some time, but some questions and problems remain unsolved. Increasing number of companies start to look into migrating their old, colossal monolith systems to more flexible and lightweight microservices, that fit rapidly changing business needs more conveniently [New15]. There are a lot of opinions on how one should start with decomposing monolithic system, Sam Newman [New19] suggests incremental approach which instructs to cut out functionality from monolith step by step. Others suggest not touching monolith at all and focus on creating new microservice ecosystem that would replace old system all at once [FBZ+19]. Even though there are quite a number of strategies on how to split up system functionality wise, not many of the strategies mention persistence layer that includes data, underlying database engines and data consistency. This layer is the main focus point of this paper.

Keywords: microservices, data management, monolith, persistence layer, data consistency

Santrauka

Mikroservisinė architektūra jau kuri laiką yra diskutuojama ir naudojama informacinių technologijų industrijoje, bet kai kurie klausimai ir problemos lieka neišspręstos. Didėjantis skaičius įmonių pradeda galvoti apie savo senos, kolosinės, monolitinės sistemos migravimą į lankstesnius ir neapkrautus mikroservisus, kurie daug geriau pritaikomi sparčiai besikeičiantiems verslo reikalavimams [New15]. Yra begalė nuomonių apie tai kaip turėtu būti pradėdamas monolitinės sistemos skaidymas, Sam Newman siūlo inkrementinį sprendimą, kurio pagrindinė idėja pažingsniui atskirti funkcionalumą nuo monolito. Kiti siūlo neliesti monolitinės sistemos išviso ir fokusuotis ties naujos mikroservisinės sistemos sukurimo, kuri pakeistų monolitą vienu ypu [FBZ+19]. Nors ir kupina strategijų kurios aprašo monolitinės sistemos skaidymą iš funkcionalumo pusės, ne daug strategijų pamini patvarumo sluoksnį, įskaitant duomenys, pamatinius duomenų bazių variklius, bei duomenų nuoseklumą. Šis sluoksnis yra pagrindinė šio darbo tema.

Raktiniai žodžiai: mikroservisai, duomenų valdymas, monolitas, patvarumo sluoksnis, duomenų nuoseklumas

Object of research

This paper aims to research problems and solutions of data migration exercise when migrating monolithic system to a microservice architecture based one.

Objective

The main objective of this research is to empirically determine the optimal data model for an illustrative database schema in exercise of preparing a monolith for migration to a microservices architecture, while evaluating the effectiveness of normalization and optimization of data storage.

Tasks

In order to achieve this objective, analysis of microservice architecture features and problems will be conducted, including what is microservice architecture, how data flows in it and what are general problems of having a distributed system.

With the intention of analyzing persistence layer, we will go deeper into detail about data ownership and consistency challenges including the actual data migration task.

In an effort to put analyzed theory into practical examples we will go through some suggested methods on how to deal with data migration challenges during the migration work in depth. Taking into consideration how they assist in figuring out data ownership and consistency in the migration.

After analyzing suggested methods, we will be aiming to put this knowledge to practice and run an experiment of determining best fitting data model for a monolithic system that will be undergoing migration to microservices.

Real, database schema will be used, that is provided by external party which had a request to remain anonymous, including details on amounts of data present in actual production environment. Amounts of data will be used to take design decisions when performing data schema restructuring exercise. Database schema will be altered additionally to remove any table columns that would not add much value to the paper.

Simulation will then commence, using the analyzed knowledge and methods. Business requirements and the business flows will be collected, through them we will be aiming to understand the purpose and specifics of the system. These details will be later on used when arguing and evaluating usage of certain methods, and determining if they are indeed realistic to use in our

particular scenario. Technical flaws will be identified and attempted to solve, while taking into account how these might affect distributed system

Most important task will be defining optimal database model by using methods focused on establishing logical domain boundaries and data ownership. Design flaws will be analyzed and mitigated. It is expected that some tables might require total overhaul to achieve clear data domain boundaries. During the restructuring exercise usage of previously analyzed knowledge and methods will be observed with an aim to investigate how the comprehension of data ownership and consistency knowledge, along with the implementation methods and strategies assists us.

Afterwards normal forms of our newly created tables will be analyzed with an objective to evaluate if new database schema is indeed optimal to migrate. To support our objective, data storage optimization evaluation will be done as well. Data storage improvement will be measured to identify if new data model is better at reducing data redundancy and overall achieve better data storage utilization.

Data consistency analysis will also be done, by simulating new business requirements and how complex it would be to implement them in the system after implementing new data model, illustrating how new data model enables us to adjust existing system to match new business requirements.

1. Microservice architecture overview

1.1. What is microservice architecture?

Microservice architecture style is a subgroup of service oriented architectural patterns where each separately hosted web service is responsible for its business function and can be developed and deployed independently [New19]. Having system designed as microservices enables teams to focus work and develop functionality on separate services that each can have their own technology stack that provides best environment for required tasks. [ANN+16] The predecessor of microservices can be considered architecture paradigm called Service Oriented Architecture (SOA). Notably IBM SOA management approach had a considerable impact on how the software architecture world progressed towards more of a service-based architecture style. A lot of thought went into defining best practices of managing services and defining how business process should be reflected in the services themselves. [GK7]

1.2. How these microservices interact with each other? Ecosystem of microservices.

Independently hosted microservices often still need to interact with each other and exchange data to enable more complex customer journeys. For that they need to have a robust way of communicating and sending requests to one another. There are two main ways of communication, one being HTTP protocol and other being event messaging. HTTP protocol has the advantage of being more reliable on when certain actions are done on each service, but in turn this kind of communication takes out the point of having separately hosted services as they cannot really act independently and depend on response from other services, at least most of the time. Events on the other hand are much more distributed and function as an indication for other services that host service have done something and needs others to do something on their side. This results in system acting as separate workers that work towards common goal without blocking its colleagues or knowing what happens inside of others, although it’s not always the best thing because some domains require immediate persistency all across the system. [AWC19]

1.3. Challenges of distributed system

Distributed system comes with its own unique challenges like availability, even if a single component of the system is not performing, whole system might be considered dysfunctional, especially if there are no attention given to ensuring additional counter measures, aiming to remove single point of failures. Reliability is a similar aspect that should be given extra attention as having number of small components interacting with each other is not something that is achieved

effortlessly, as distributed system can only be considered reliable if the bugs in components won't bring down the entire application [KM19]. There is an acknowledged theory about availability, consistency and partition tolerance balance and compromises in distributed system called CAP theorem, sometimes also mentioned as Brewer's theorem. This theory emphasis that in distributed systems, there will always need to be sacrifices of some properties and metrics, and that system should be designed with that in mind [GL12].

More often than not microservice based system is hosted in unified private network, but there might be cases where microservices are hosted in different hosts or locations. Under those circumstances performance and security are challenges that arise naturally because in case of communication between components happening across the network, it gets hard to sustain fast but secure data transfer [DGL+17]. Further ahead we can see that if there are these kind of challenges on business layer, what challenges might this mean for persistence (database) layer? Most common include data consistency problems, unreliable source of truth and failure handling [LZS+21]. These are challenges that occur when microservices are already deployed and running, there is a whole other set of problems that come up during the actual migration?

2. Data ownership and consistency

2.1. Who owns what data in microservice architecture

Most business functionality that can be found in IT projects requires data, as data is representation of business entities in physical domain. A question arises when migrating monolithic systems (that usually has all the data available in one place, one database engine), which services should own the data? It is agreed that a single service, which most of the time has one business responsibility, should be the owner of data, that belongs to its domain [New15]. E.g., payment service should most certainly be the owner of payment information. It might seem straight forward at first sight but who should own data when more than one service strongly depends on it?

For example, businesses must split up business units and ownership of certain data among them. If several business functions require the same information, for instance customer data, it shouldn't suddenly become public domain that everyone can tinker with, observed in figure 1. Having couple of units working directly on customer data directly can lead to some unexpected results even when whole business resides in one location.

Business units from different locations tinkering on the same data is bound to have catastrophic consequences. For instance, department responsible for sales might be in middle of auditing their sale data might run in to a situation where customer onboarding department offboards customer and deletes his data without informing any other department, resulting in mismatching records across the system and possible huge fines.

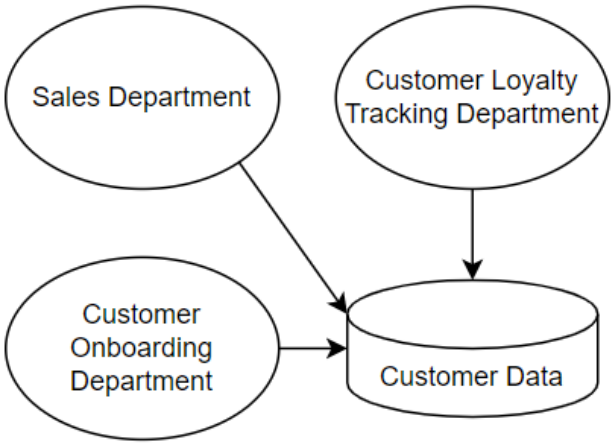

Fig 1. Illustration of several departments interacting with shared data

It's quite easy to transfer the same logic to context of monoliths and microservices. Trying to establish data ownership and clear bounded contexts in very early stages of designing is one of the

backbones of truly consistent and reliable microservice based system [New19]. Although this is only possible with a very good understanding of the domain and of the product that is being developed, that's why microservice architecture is not a recommended approach for businesses in early stages, as the domain boundaries are not yet identified and can rapidly change [Ric+18].

2.2. Data consistency

Data inconsistency is a fundamental problem in distributed systems because data required for business process might be hosted in several databases and involve several services [LZS+21]. This is not a problem in monolithic system as for separate actions, service still operates on the same database and can trust it to be most up to date data need for whole process to be done, as the transactions happen in the same database engine and there are often convenient tools to manage them [KM19]. That's not the case in microservices because individual parts of business process might be executed in separate services in parallel and asynchronously. This in turn means that if some service fails to finish it's part of work, we need to make sure that other services do not finish theirs as well, as that might lead to data/state inconsistency across the system [DGL+17].

This can be solved by treating business process as one distributed transaction, in which all individual actions must be completed successfully for whole process to be deemed successful. This is most successfully implemented by having two phase commit approach [Rud18]. This is called strong consistency and the use of it really depends on the needs of total consistency in certain domain or particular business process (e.g., banking or money transaction). In cases where total, real-time consistency is not that important it might be worth going with eventual consistency [FFZ+17]. This is a system design where transaction state is not required to be consistent across all affected services and databases. For example, it's required to keep payment state for certain order consistency across payment and order services, but it might be tolerable to not wait for response from order history service to arrive before telling the customer that order was finished successfully.

It's worth looking into what business processes need to have total consistency and which are okay with eventual one when designing your system [FFZ+17]. Having atomic transactions happening across as few services as possible, is bound to result in more logically divided system, of course without forgetting domain boundaries and data ownership. This challenge is further described in pattern 4 of data migration challenges, methods and patterns chapter.

2.3. Data migration

Figuring out data ownership and data consistency issues might put up quite a challenge even in existing microservice systems but keeping data consistent across old monolith and new services when migrating existing system is a staggering task. Freshly extracted microservices are rarely trusted to become a replacement for same functionality and data ownership in monolith immediately, as they are bound to have some lingering issues that will only be observed after some monitoring period.

Data migration should be a gradual process that ensures that no data is lost because of unexpected behavior of new microservice or because of flawed infrastructure. For example, there might be a case where monolith is actively used by data heavy business operations, for example banking system. These operations rely on data to perform any and all actions to achieve their objectives. Splitting off new microservice and transforming all operations to use it, won't be straight forward as any inconsistency or missing data will result in certain operations not being executed properly. That's why gradual data migration is a must when transformation and testing is still in progress and that is the case until new microservice can take the full responsibility of the data and functionality. Certain strategies and patterns can be used as tools during migration that help in keeping the consistency and transferring of ownership. Patterns 2 and 3 provided in 4th section of this paper dive deeper in detail how to achieve gradual transition.

3. Analysis of data migration challenges, methods and patterns

We are going to go through some of the potential monolith splitting and migration challenges and some viable solutions to them.

3.1. Database schema is used by other apps

More often than not, old and complex monoliths become source of data for some other applications in company's ecosystem, meaning that splitting and exporting existing database might not be straightforward. The exact challenge in this case is what to do with database schemas that are currently being used by external applications and any changes to structure of it would break the functionality of exterior services.

There are couple of solutions provided by Sam Newman in his work on Monolith to Microservices [New19]. First one being Pattern of Database View, this is to use in cases other services are directly accessing monolith's database. The essence of this pattern is to provide dedicated public schema that would consist of all views that might be needed by consumers of the database. This allows migration team to make changes to database schema, as long as they are able to keep public views intact.

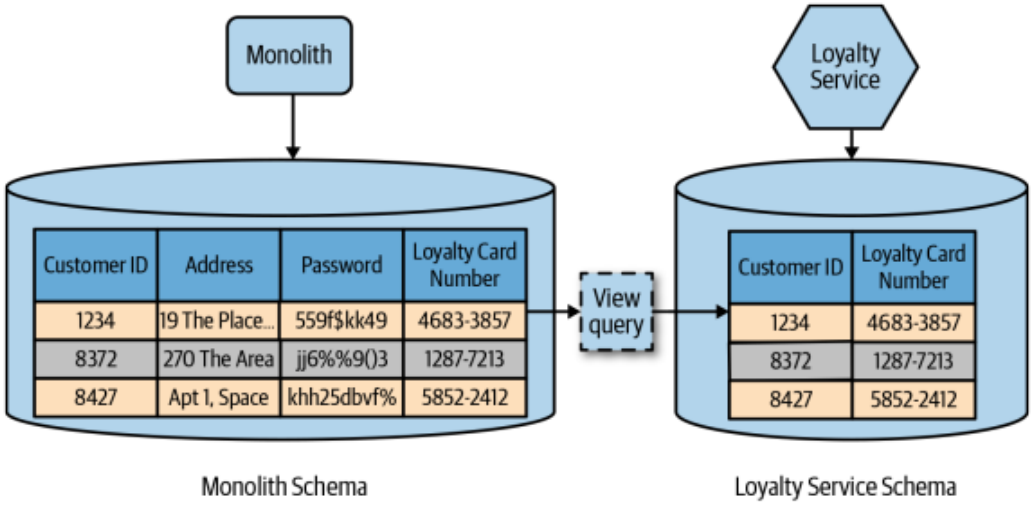

Fig 2. A database view projecting a subset of an underlying schema [New19]

In figure 2 we can observe an example where Loyalty Service was depended on some data from monolith's database Customer ID and Loyalty Card Number. Having exposed these fields as a view, we can now split and extract remaining fields of Address and Password to a separate schema without breaking the contract with Loyalty Service.

Having other applications access database directly is not ideal to say the least. It is sometimes much smarter to hide it behind an API or a service. Database Wrapping Service is a pattern of hiding database behind a service that will give us some room to breathe while untangling database schema itself, because external database users will not be able to hack with our schema anymore and will have to rely on exposed data.

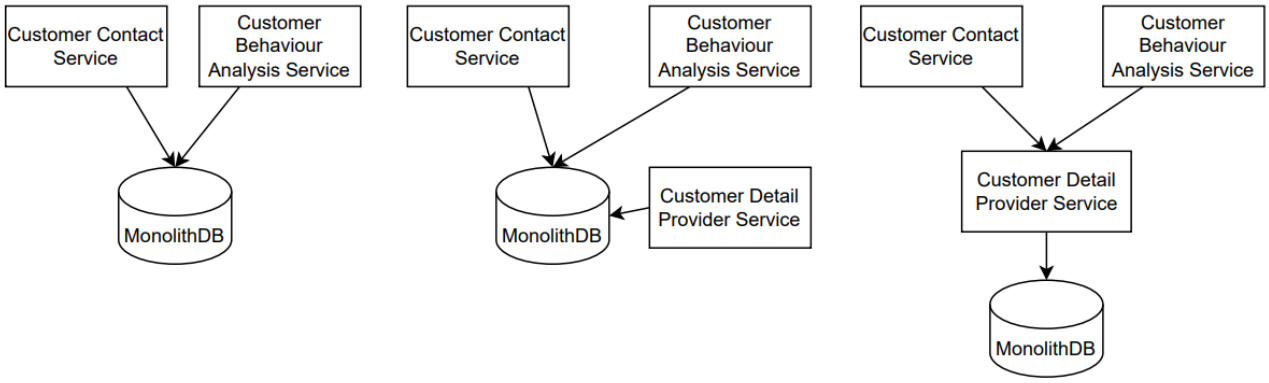

Fig 3. Database Wrapping Service pseudo-schema illustration

In figure 3 it is shown how Database Wrapping Service would be applied in a system where we have two services Customer Contact Service and Customer Behaviour Analysis Service depending on data from Monolith database. Additional Customer Details Provider Service (CDPS) is created that will access the database and provide required customer information to any consumers that will interact with it. Then the CDPS is placed as a middleman for database and services that previously accessed database directly. Having this pattern in place it is easier to adjust and tinker with the Monolith database schema itself without breaking any external applications.

3.2. Splitting the schema and unsplittable tables

Second step after ensuring that schema can be tinkered with without breaking any external dependencies is to split the actual database schema to really get a good understanding of how database of newly split of microservice will look like. In this step it should be clear, which data and functionality belongs to the new microservice, all though it’s not yet worth thinking about how the actual data synchronization and ownership will be handled. It might just be that you will face a point where determined domains do not make sense in regards of the database tables themselves, and this is a good place to reflect on design decisions. It should be noted that existing table schemas should not be a deciding factor, only an additional thing to take into consideration when making design decisions.

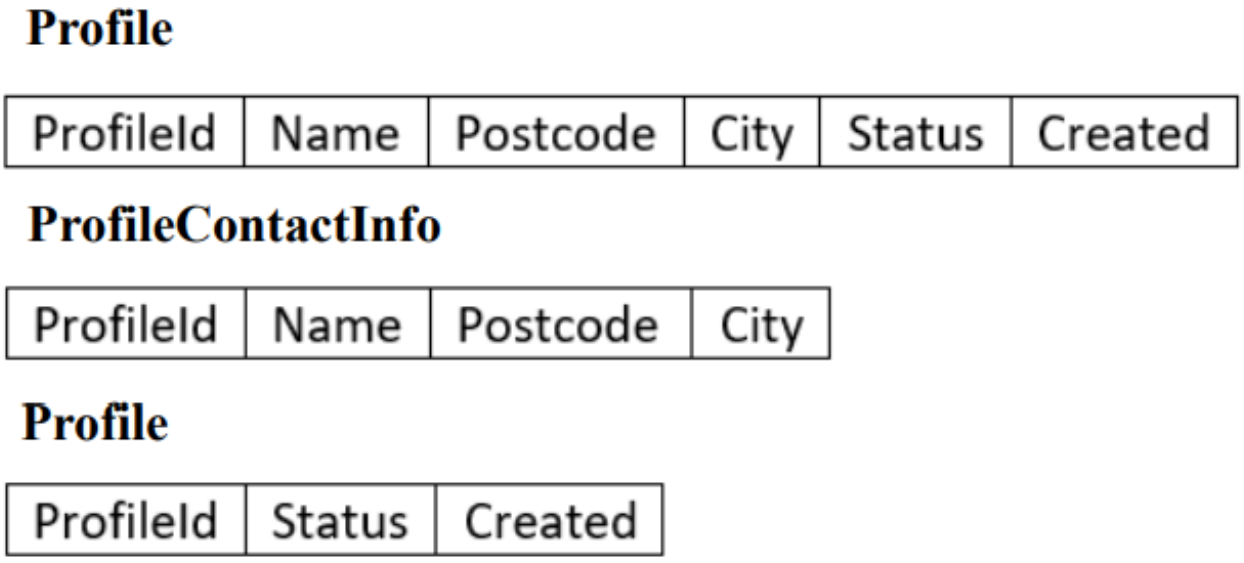

Fig 4. Database table splitting before and after structure

In the example provided in figure 4, Profile table after the split only have couple of fields beside primary key and it might be counterintuitive to do the split as one of the entities will be left with very little fields, but this is the moment we need to remember that we are trying to achieve exactly that: microservices with single responsibility and splitting down tables until it makes sense to do so (until we are left with entities that still have an actual meaning) is what we should strive for [New15].

3.3. How to transfer ownership of data, data synchronization

Following method describes a combination of patterns Change Data Ownership and Synchronize Data in Application in Newman's book [New19].

Having done database schema changes in the monolith database it is time to transfer the actual data ownership to a separate service. It might be smart to have new service up and running before creating new database and transferring the actual data to it, see figure 5. Having new microservice deployed and connected to old monolith database allows development team to inspect if the system behaves as expected before performing the actual physical split of database.

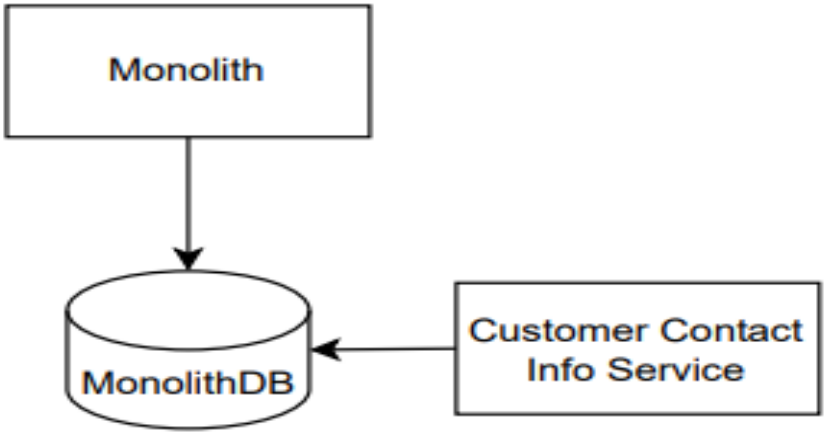

Fig 5. New service accessing data in monolith's database

After ensuring that newly deployed service indeed behaves as expected, the first steps of data ownership transfer can begin. Observed in figure 6, at first new database is required to be created with schema that will be used by the microservice. Initial data synchronization might as well simple be data copying from monolith database to new one and having our microservice still reading the data from monolith database to operate, while also synchronizing any changes to new database.

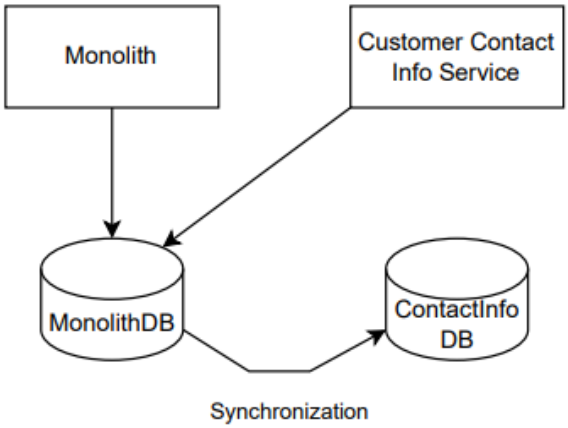

Fig 6. Synchronizing data to new database

Moreover, our new service should become responsible for application wise synchronization of data by making write changes to new database instead of relying on synchronization to do it for it, as seen in figure 7.

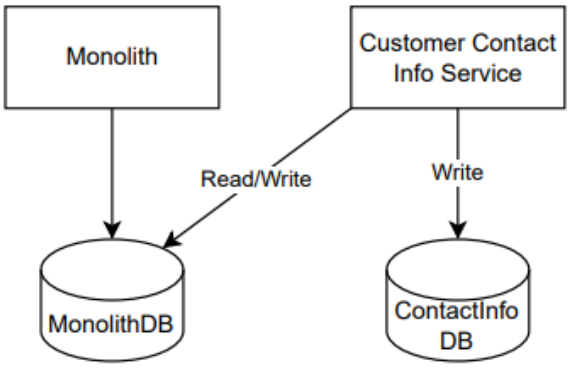

Fig 7. Service taking responsibility of making write operations

In a stable system, transfer of data ownership can be initiated, microservice should start performing read operations on its own database as revealed in figure 8. Freshly deployed independent service is has become fully responsible for the data that was previously stored, distributed and altered in old, consolidated database. At this point monolith should start using service API to reach previously available data in case it still needs the data for any business functionality, even though synchronization between databases is still happening in the background. Only after some observation period can the data synchronization no longer be performed and split off microservice fully trusted with being the new source of truth.

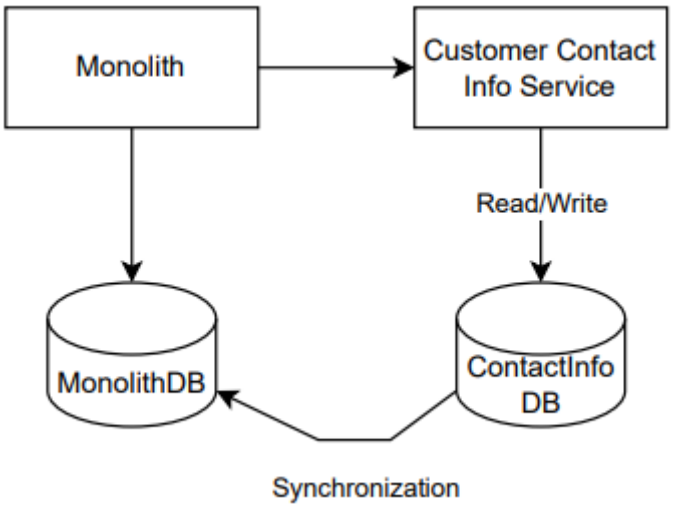

Fig. 8 Service takes full responsibility of its data

3.4. Data consistency and how to achieve it.

As described before data consistency is a challenge even in a well-established microservice architecture-based system. During the migration it might be worth taking extra time to consider which entities or tables might cause data consistency problems if they are split down. Essentially at this point development team along with business experts should discuss which business functionalities and transactions that tinker with entities, that might be affected by the migration, should be left together. As this decision is more convenient to make in the beginning of the migration when it's still easy to establish which transactions should be left as atomic ones and which ones can be considered valid even with eventual consistency.

Popular and mature pattern to achieve and maintain data consistency in microservice architecture is the Saga pattern. Saga is a way to distribute cross-service transaction into sequence of local transactions. If there is a failure or violation of business rule in any of local transactions, whole saga rolls back, in sense that compensating operations are executed to revert changes that were done by any previous local transactions. Some local transactions might not need compensating transaction as they might not do any critical changes to any of the data as shown in figure 9.

Step	Service	Transaction	Compensating transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	—
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	—
5	Kitchen Service	approveTicket()	—
6	Order Service	approveOrder()	—

Fig. 9 Service, transaction and compensating transaction table [Ric+18]

There are two ways of managing sagas, one is choreography and other one is orchestration. Choreography coordination is achieved by distributing decision making between saga participants, services that are involved in execution of saga are responsible for communicating with each other, mostly by events. Orchestration is exactly as it sounds, there is a delegated saga orchestrator that is responsible for sending messages to services and controlling the whole flow of transaction execution.

An example of orchestration-base saga can be observed in figure 10. All of the communication happens via messaging broker, using commands. In this example saga orchestrator is hosted in Order Service, it initiates all steps of the saga, including verifying customer, creating ticket/order, authorizing card, approving restaurant order and finally approving order. Orchestrator also receives all of the updates from saga participants and further instructs other services to perform their actions. This way the state of saga is controlled in one place and is easier to manage.

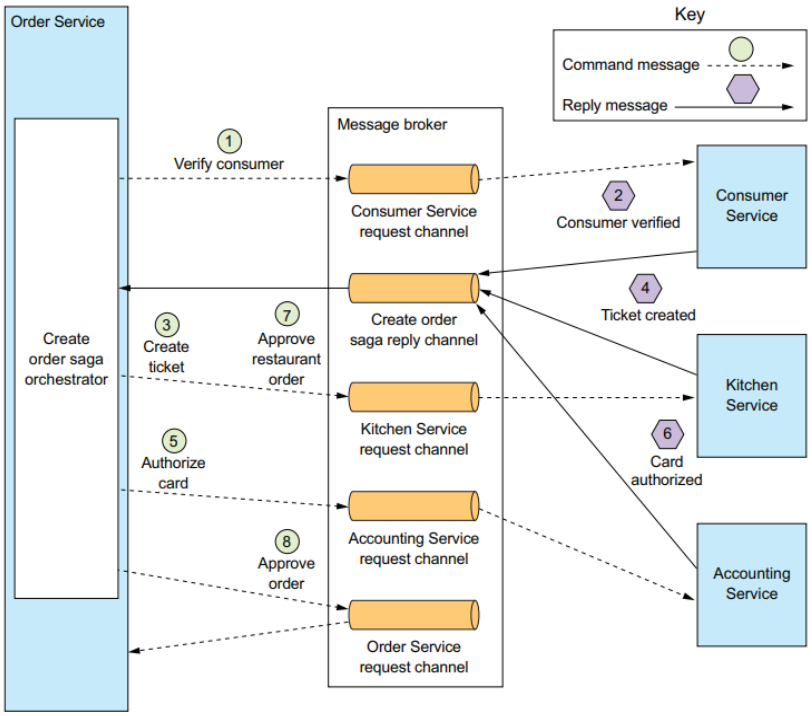

Fig. 10 Create order saga, implemented via orchestration [Ric+18]

4. Practical illustration

Main focus of practical illustration is to showcase suggested methods and evaluate if they are indeed useful in the migration, particularly in determining the most optimal data model. Usefulness of methods will be measured qualitatively: undergoing analysis on how usage or guidance of a strategy or method helped in overcoming particular challenges and answering any questions that arise. Another goal of the simulation will be to illustrate how having knowledge of data consistency and ownership can assist us in making rational decisions when establishing domain boundaries and resolving design flaws. Designed data model will be evaluated using normal form analysis and data storage efficiency measurement. Table names and columns will written be in upper case to improve readability.

4.1. System under analysis

Database schema used in the practical illustration was provided by external company and it's an actual schema used in production environment of system. It suffers from various poor architectural solutions, which we will be trying to analyze and solve. First step will be analyzing the current status of schema and various entity relationships in it. Possible improvements will be defined and taken while also considering business requirements. This analysis and restructuring will happen without considering technical migration itself, as we first need to prepare the schema to actually be migrated to separate services. This first stage of restructuring will help us to get a clearer view on what are the data boundaries and assist us in identifying possible microservices. High level schema we will be analyzing and trying to refactor is shown below in diagram:

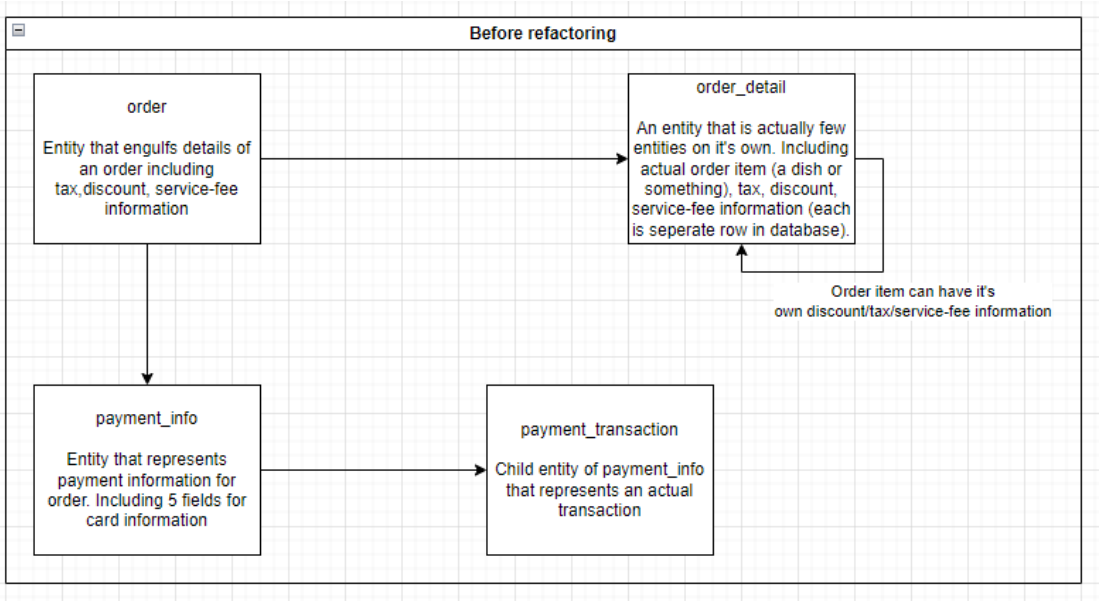

Fig. 11 POS system before refactoring

The full database schema, including all tables and their columns can be found in appendix 1. The database schema was taken from a point of sale (POS) system, to be more concrete, the part of auditing various orders and their details, including tax, discount and service fee information. POS system is hardware and software used in businesses to manage sales transactions. Additionally couple of database tables were provided that illustrate payment auditing. Business logic behind these tables is the following (only mentioning the logic relevant for our illustration):

- Order entity is created when POS operator initializes new order to start tracking information about the transaction between customer and merchant. Order entity is populated with technical information like creation times, merchant and store information and given an initial status.
- During the process of transaction, be it: scanning of goods, delivery of the services or catering at a restaurant, order details are created and order entity itself is updated with data about total of tax, discounts or service fees.
- Each item in the order itself also has additional tax, discount or service fee information that is applied directly to the order item and not the order.
- Finally, order is finalized, all information is updated and payment can be processed, payment information tracking is initialized.
- During the finalization of payment, payment information is saved, including payment date and time, optional card information (in case of card payment) and also any audit related data.

Upon first glance it might seem that there is nothing complicated with the processes, but digging deeper into how this was implemented on schema level, flaws of past inferior design and poor technical decisions begin to surface. Main effort of this illustration will be solving design flaws, including refactoring of some tables while trying to identify any possible data boundary improvements. It’s good to first try to identify any obvious technical defects that might make our migration troublesome.

4.2. Technical flaws

This is an old database schema with tables that had seen many changes made due new business requirements or simply due new developers trying to improve it throughout the years. Main purpose of this sub-section is to identify any technical limitations (caused by poor technical

solutions) that would cause us problems while migrating the system to a more distributed one. After analysis of existing tables few technical flaws are identified:

- Couple of tables (PAYMENT_INFO and ORDER) have CREATION_TIME_ZONE field that is used to identify in which time zone was this payment or order created. This is not actually needed as for each of these entities we already track STORE_ID and the date and time localization should be done based on merchant's location, in UTC, instead, because of its reliability and the main purpose of these tables – auditing [PA19]. Furthermore, having least amount of date and time complexity is preferred when dealing with distributed systems, as keeping data consistent is easier done with unified time zones.
- ORDER table has a TAX_PERCENTAGE field, which initially might make sense to have, but after looking further into relationship of ORDER_DETAIL and order it becomes apparent that it's not so useful to have. Tax percentage rate is determined by the actual type of order item in the ORDER_DETAIL table, for example, in case of restaurant order, different menu items might have different tax rates, like alcohol and food. Without even looking into application code, it is safe to say that TAX_PERCENTAGE field either gives away inaccurate information or is not used at all and should not be included in this schema. Having this type of relationships cut is subject to help us have a more distributed system, because of smaller amount of dependencies between tables (possible microservices in the future).
- ORDER table contains an IS_DELETED column, which essentially represents order status. ORDER table also contains an Enum of ORDER_STATUS which clearly should be appended with status that represents deleted state for the order, and IS_DELETED column would be dropped.
- PAYMENT_INFO table has a field that represents its type: PAYMENT_TYPE. In the application level it's probably treated as an enumeration but that is not the case in database. Fields that represent a limited set of possible values should be treated as an enumeration and the data type of the PAYMENT_TYPE field should be changed to tinyint(4) (which represents Enum type in this particular schema). Accommodating these kinds of values as an enumeration is helpful for avoiding easily preventable interpretation issues the communication between different services in distributed system.

List of database schema changes according to this section, can be found in appendix 2.

4.3. Design flaws and their solutions

As mentioned earlier, simple technical flaws are not the only thing that our database schema is affected by. While obvious technical flaws are relatively easy to identify, design flaws require a better understanding of the system and the business that is powered by it. Higher level overview of the processes is necessary in addition to more throughout analysis of what the tables in this schema actually represent and what are they used for.

4.3.1. ORDER_DETAIL table and it's multiple responsibilities.

Initially ORDER_DETAIL table appeared to be quite logical, just a table that represents an entity used to track any items in the order entity (ORDER to ORDER_DETAIL, one to many relationship), but after deeper look at its composing parts, it becomes clear that, that is not the case. Order detail entity on its own represents essentially what the name stands for, a detail. Detail being anything from the following:

- Order item, like a dish in restaurant or a bar of soap from a supermarket, including its name, category, price, quantity, discount/tax/entity amount for this item.
- Discount details: name, rate, type, value and category. Additionally, this discount detail can be applied both on order item and the order itself.
- Tax details: name, rate, type, inclusiveness (included in price of the item or not). This also can be applied to the order item or the order itself.
- Service fee details: name, rate, type. Shares same order item/order application.

ORDER_DETAIL table represents four totally different entities in its rows depending on which order details needs to be stored. Additionally possible application of both another order detail entity or the order itself creates recursive data structure, making the querying for full information of the order that much more complex. Data boundaries should be clearly defined even in a system that is only taking the first steps in migration journey and that can be achieved by determining who owns (or would like to own) which order detail entity. Having four entities in the same table is bound to create unnecessary confusion and difficult data ownership questions, as the systems that might need to tinker with tax information would need to adjust rows in a shared table. Furthermore, having these kinds of tables is difficult to manage and operate on as they have to be prefilled with considerable number of null values because most of the fields are not used for a single scenario.

Further, in this sub-section we will be trying to solve this design flaw by splitting the table up into few smaller ones, as well identifying and reducing any redundant dependencies between entities. By solving this design flaw, we will not only prepare this database schema for migration but also greatly reduce complexity of the monolith system by introducing unambiguous entity relationships.

4.3.1.1. Theoretical split

We are creating tables that in their data model will hold only relevant information for their entity, as mentioned in the order detail list previously. Along with structuring entities in a way that they can be as reusable as possible, while still retaining all relevant information needed for business processes. Reusability of entities will lead to optimizing our data storage, the impact on it will be measured and evaluated at the end of this section.

Discount, tax and entity (service-fee) order details look like great candidates to be treated as reusable, enumeration-like, entities. These order details in themselves only hold information that are used to calculate total discount, tax or service-fee, meaning that they are not unique for each order item and can be reused as needed (even for the order itself). Order item entity is a different case, this entity needs to hold information about all of the order details and their totals, meaning that in addition to holding calculated totals of discount, tax and service-fee, it will also hold foreign keys to rows that were used to generate the totals. Moving out calculated discount, tax and service-fee information to be held in ORDER_ITEM enables us to have more unambiguous entity relationship, as we treat additional order detail entities as additional information for the order item and not as a unique order detail of the order itself. Order item entity will also hold information on the item itself and the connection to order and payment information (as splitting of a bill at restaurant is not an uncommon scenario).

This allows us to have a clearer data ownership boundaries. Teams that are interested in historical tax, discount or service-fee information can own their corresponding tables without any fear that ORDER_DETAIL legacy table changes will affect their systems. Not to mention, having additional order detail information accessible as reusable entities allows us to easily connect order with any order-wide information, without the use of confusing ORDER_DETAIL table rows, as well as improving data utilization and reducing data redundancy.

The ORDER_DETAIL table is split into four tables, each having some fields that are mandatory for all of them like: MERCHANT_ID and STORE_ID. Discount, tax and service-fee

tables hold the order detail information used to calculate totals in ORDER and ORDER_ITEM tables, like rates, types and historical information such as discount name. ORDER_ITEM table holds all the information about particular order item, like: category, price, name, type. ORDER_ITEM also hold information about totals of discount, tax and service-fee and foreign keys to more detailed information: DISCOUNT_ID, DISCOUNT_AMT, TAX_ID, TAX, ENTITY_ID, ENTITY_VALUE. Tables that were created in process of this exercise and their normal formal analysis can be found in appendix 3.

4.3.1.2. Normalization and data storage optimization

Having done the restructuring of ORDER_DETAIL table, effectiveness of data model has to be measured. Evaluations will be done by performing normalization analysis of new tables, trying to understand which normal form we managed to achieve, to identify if we did not create any overly complex data models [Fag81]. Additionally, data storage optimization analysis will be conducted to measure how our changes impact data storage efficiency in this particular database schema.

Database normalization is a process of minimizing data redundancy and improving data integrity. It is most commonly achieved by breaking down big tables into smaller, single purpose tables that efficient relationships between them. The main goal is to ensure that data is not duplicated and that data anomalies are eliminated. Level of normalization is measured by normal forms of tables, higher form given to those tables that successfully solve complex dependencies and efficiently reduce data redundancy. Additionally, normalization is beneficial to improving data consistency and simplifying data maintenance. Although too harsh normalization might result in database schema that is very difficult to navigate and query data from [Fag81] [WDL10]. Following is analysis of newly created order detail tables for tax, discount and service-fee.

Tables reach first normal form (1NF) because:

- Each column only contains one single value (no multiple values present in single attribute)
- Each column only contains single data type
- Each column has unique name that represents the value it holds
- Order does not matter as the values are retrieved via ID, and the meaning of row does not change even if it's in different order each time you query it (for example by creation time)

Tables reaches 2NF because:

- It is in 1NF
- Because there is no partial dependency and that is because primary key is a single column ID

Tables reaches 3NF because:

- It is in 2NF
- All values depend only on the primary key, there is no additional columns that does not depend on ID

Tables reaches Boyce-Codd Normal Form (BCNF) because:

- It is in 3NF
- There are no values that depend only on one part of primary key as the primary key itself is only one value

Tables reaches 4NF because:

- It is in BCNF
- There are no possible rows that can have the same primary key and different values to it (multi valued dependency is not possible) as every single row represents unique tax order detail information.

After normalization analysis of new tables, it is determined that they are all at least in fourth normal form, main reason for that being their simplified design and being more of a single responsibility entities, especially compared to legacy ORDER_DETAIL table. Main outcome of having our new tables normalized are those of data redundancy decrease and optimization of storage efficiency, that we will further investigate in next paragraph [WDL10].

Data storage optimization is also observed, by having duplicate order detail information moved out to a separate tables, we are able to reuse them when order item requires any additional information. Particularly tax information is an order detail that is used for every single order item, as all items have to be taxed. Previously for each order item, there was a new row being created in ORDER_DETAIL table, with abundance of unused NULL fields, and in general, duplication of tax information that does not serve a meaningful purpose. Overall size of ORDER_DETAIL table is 1.4 TB, with approximately 10.347% of the data being tax information order details. From those

10.347% tax information rows 0.534 % are of unique TAX_ID, meaning that there are only so little rows that represent unique information. Other 9.813% tax rows are duplicate ones, and having those include duplication of the following fields: STORE_ID int(11), MERCHANT_ID bigint(20), CATEGORY_ID bigint(20), CREATED_ON timestamp, CREATED_BY bigint(20), TAX_NAME varchar(45), TAX_RATE decimal(12,4), TAX_TYPE tinyint(2), IS_INCLUSIVE tinyint(1), PARENT_ORDER_DETAIL_ID bigint(20), ORDER_DETAIL_TYPE tinyint(4). Overall space that is occupied by duplicated fields is estimated to be around 95 bytes, without taking into account NULL fields that still require some storage to track, meaning that by restructuring them to be reusable, and reducing duplication we already save estimate of 6.908% storage space, and that is only from tax order detail information. Numbers used to calculate these estimates can be found in appendix 4. Less drastic (because of optional usage), but still significant storage utilization improvements are to be observed for discount and service-fee information. Taking all of that into account, we are confident to say that this kind of restructuring is greatly beneficial to data storage optimization, because of the pure storage utilization improvements and reduce of data redundancy.

4.3.1.3. Challenges of physical split

Having done the theoretical restructuring work, before actual physical splitting of the table can occur, we must first examine how our changes might affect any external systems. Those systems that directly use our database table of ORDER_DETAIL to enable any business functionality, for example, generating reports. These systems depend on current implementation of how order detail information is provided to their processes and undergoing needed adjustments is not a viable option at the given point in time. We will try to overcome this problem using the first strategy, which describes challenges of other applications using database schema, mentioned in third section about analysis of strategies and methods. This method suggests to use database views or database as a service pattern to retain contract between our system and any external one. We will be going with database views and try to recreate legacy contract of ORDER_DETAIL table to provide to external system, while also analyzing if this method is performant in this particular case.

Creating a database view for usage of external systems is not a complex task on technical level. The main challenge is trying to map entities that are now living in separate tables, being connected in a different way than they were before, to a legacy ORDER_DETAIL table. The approach we will try to take is to go top to bottom through entity relationships and try to recreate the rows in database view through joins.

First of all, fields of ORDER_DETAIL table needs to be setup to be filled in, that can be done easily by just taking the legacy ORDER_DETAIL creation SQL command. Subsequently we need to remember how ORDER_DETAIL row of order item was connected to any of its additional order details, that being via PARENT_ORDER_DETAIL_ID on additional order detail and on one of DISCOUNT_ID, TAX_ID and ENTITY_ID on order item. Next, another field that we dropped (as it became useless) needs to be filled in, ORDER_DETAIL_TYPE, this we will fill in SQL query itself, by recognizing from which table we get the information from. In our exercise of splitting the table we did not drop any other information so it can be fetched either from order item or from discount, tax or service-fee entity tables to be filled in and ready to use for external systems. Having these questions figured out we are ready to write pseudo code that will go through each ORDER_ITEM table row and generate us parent ORDER_DETAIL row with addition to recursively adding it's discount, tax, service-fee information entities as ORDER_DETAIL rows and connecting them via PARENT_ORDER_DETAIL_ID.

At first look it might look like we are taking quite a performance hit by having to recursively recreate all of the ORDER_DETAIL rows (which there is a huge amount of), including discount, tax and service-fee information that now lives in separate tables. This would be truth if we were to operate on this table frequently and rows on any of the new tables were modified on frequently, but previous knowledge of business processes allows us to take this performance hit because of the nature of business requirement and stored data. This database schema is used for auditing and having that in mind we can estimate that discount, tax or service-fee entities should not be modified ever, as they are used for auditing purposes, and in case of new discounts, taxes or service-fees new rows should be added. Order item entity should also normally not be modified once it is registered in the system, even though some changes might be expected in case of human mistakes. Furthermore, audits are not performed daily, and end of month data is most frequently used. Earlier statements, combined with clustered indexing of foreign keys to derived tables, grant us freedom with going with materialized views to have ORDER_DETAIL view ready for usage whenever end of month comes [Muk19] [CY12]. Database schema after refactoring:

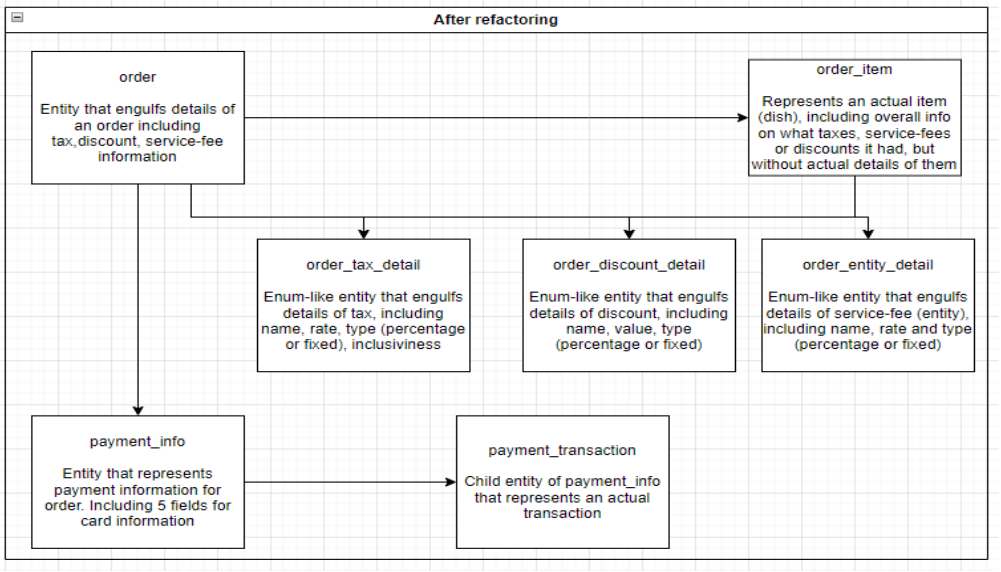

Fig. 12 POS system after refactoring (ORDER_DETAIL table split)

4.3.2. Suspected overburden of card details in PAYMENT_INFO table

Second method mentioned in analysis of data migration challenges, methods and patterns section talks about splitting tables. More explicitly second method suggest to split table until tables that we acquire after splitting, still make sense and represent an actual entity. PAYMENT_INFO table on its own represents information about payment:

- Information about what this payment pays for: ORDER_ID, AMOUNT.
- Technical information about payment itself: PAYMENT_TRANSACTION_ID, TRANSACTION_CODE, AUTH_CODE, PAYMENT_PROCESSOR_TRANS_CODE.
- Information about card details: CARD_DATA, CARD_TYPE, CARD_LAST_4DIGIT, CARD_HOLDER_NAME, CARD_HOLDER_LAST_NAME.

It’s apparent at first sight that there is overburden of card details in this table, it seems like it’s a whole separate entity inside another entity (payment information). Method suggests to split it off to a separate table that would include card details and just join it with payment information entity when it’s needed. Second method also suggest that if we can split, we should split table until it makes sense, and card detail table would make total sense. It seems obvious that we should split it off and proceed with our exercise, but there is a catch.

Having a separate table for card details sounds reasonable and very clean, but it should first be determined if there is a business need for that. In previous design flaw we emphasized that this is audit system where we store information regarding orders, order details and their payments, and

those are not frequently modified. We should raise a question, if some team inside auditing segment has usage for this separate card details table? Further system analysis points to a negative answer, even though card details might make sense to have on its own from logical standpoint, it does not from business point of view.

Fundamentally card details have no real usage on their own, even if it represents a unique entity, it has no function and is never used solely without payment information. Having card details as a separate table would make sense in case of payment part of the system, as we could then track any updates to the card itself, or would have a service that would use card detail information to make verifications and process the transactions themselves. In case of auditing system, it would create an additional table that would always need to be joined together with data in PAYMENT_INFO table to be useful, this would in turn make the queries and analysis tool underperforming and have unnecessary complexity, especially given the huge number of payments being done in the system. There might be an urge to follow reusability stand point mentioned in previous design flaw, but in this scenario, cards are unique most of the time and having a separate table to match up existing card to a payment info is not ideal. This could also add additional unnecessary issues and their solutions because data retention laws in case of any retention rules applied to PAYMENT_INFO table, but not card details table.

Information about payment information and card details in this distinct scenario belongs to the same business flow and have no usage on their own. In this particular scenario we've decided to not split off card details to a separate table, debunking second method and also suggesting an improvement to it. The improvement being that the exercise of splitting tables into smaller ones should be carefully thought through considering business requirements and data ownership from business flow point of view, considering the usefulness of having separate entities.

4.4. Discount order detail migration plan

This section illustrates preliminary gradual migration plan including pseudo-API contracts and messaging event description. Marketing team came up with new requirement of having a tool to update their discount campaigns, including, automatically adding new possible discounts into auditing system. User story they've created for us:

- As a marketing team,
- We want to have ownership of discount order details used for auditing, and be the source of truth for it,

- So that we would be assured that audit information regarding discounts is tightly managed.

Our responsibility as auditing team is to physically transfer discount order detail information handling to their eco system. Currently discount order detail is a table on our monolith database engine, meaning that to implement this new business requirement we are required to physically migrate the table to their database engine. Furthermore, we still require this information to generate our database view for reporting team usage, meaning that we will still have this information synchronized to our database, but will no longer be the source of truth for it. This particular scenario poses a threat to data consistency issue in a distributed system and to overcome it, third method from section 3 of this paper will be used.

First a microservice will be created and deployed, it will engulf logic to manage discount order details used for auditing, including adding discounts, notifying our monolith about any updates, endpoint to fetch possible discounts and their details. Following third method, the microservice will first connect to monoliths database engine to update discounts, this is used to test if the microservice itself can be trusted with management of discounts. Only after ensuring that there are no anomalies and that microservice is behaving as expected, we proceed with creation of a discount order detail table on their database. To achieve data consistency, we put manual data synchronization job into usage, that manually synchronizes current state of table on our monolith to theirs. At this point eventual consistency is sufficient, but that will not be the case when data ownership will be transferred. Current state of the migration can be seen in a diagram below:

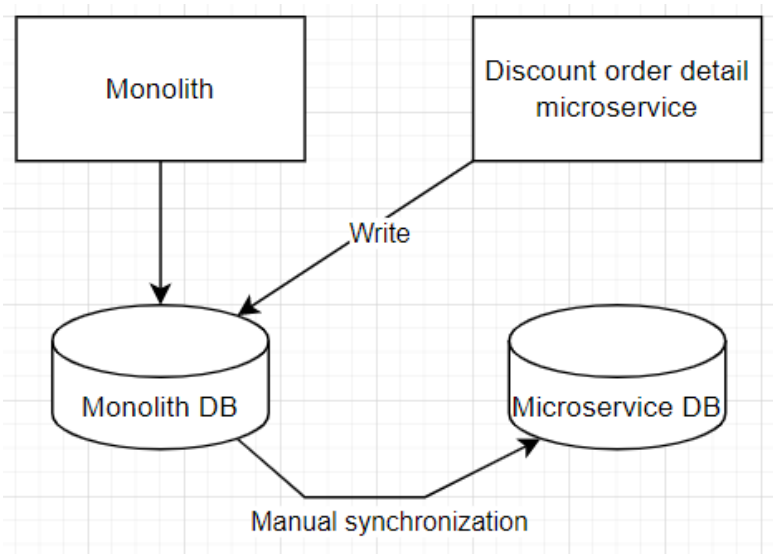

Fig. 13 State of migration when microservice is already responsible for updating monolith DB

Further steps in third method instructs us to start relying on microservice to update both our monolith's and their own instance of discounts. Manual data synchronization is not executed anymore, we transfer responsibility of managing discount information to external microservice. Microservice itself thus far depends on our database to be the source of truth and uses it for reading operations.

After sufficient grace period and testing we are ready to fully transfer data ownership to the microservice and underlying database table. Absolute consistency is the goal of this step, for that process of notifying our monolith about any changes is implemented, in addition to exposing endpoint for our monolith to actually fetch these changes and update discount information on our database. Pseudo API contracts and messaging event used to achieve these responsibilities can be found in appendix 5. After the migration, process of adding/updating discounts is presented in following sequence diagram:

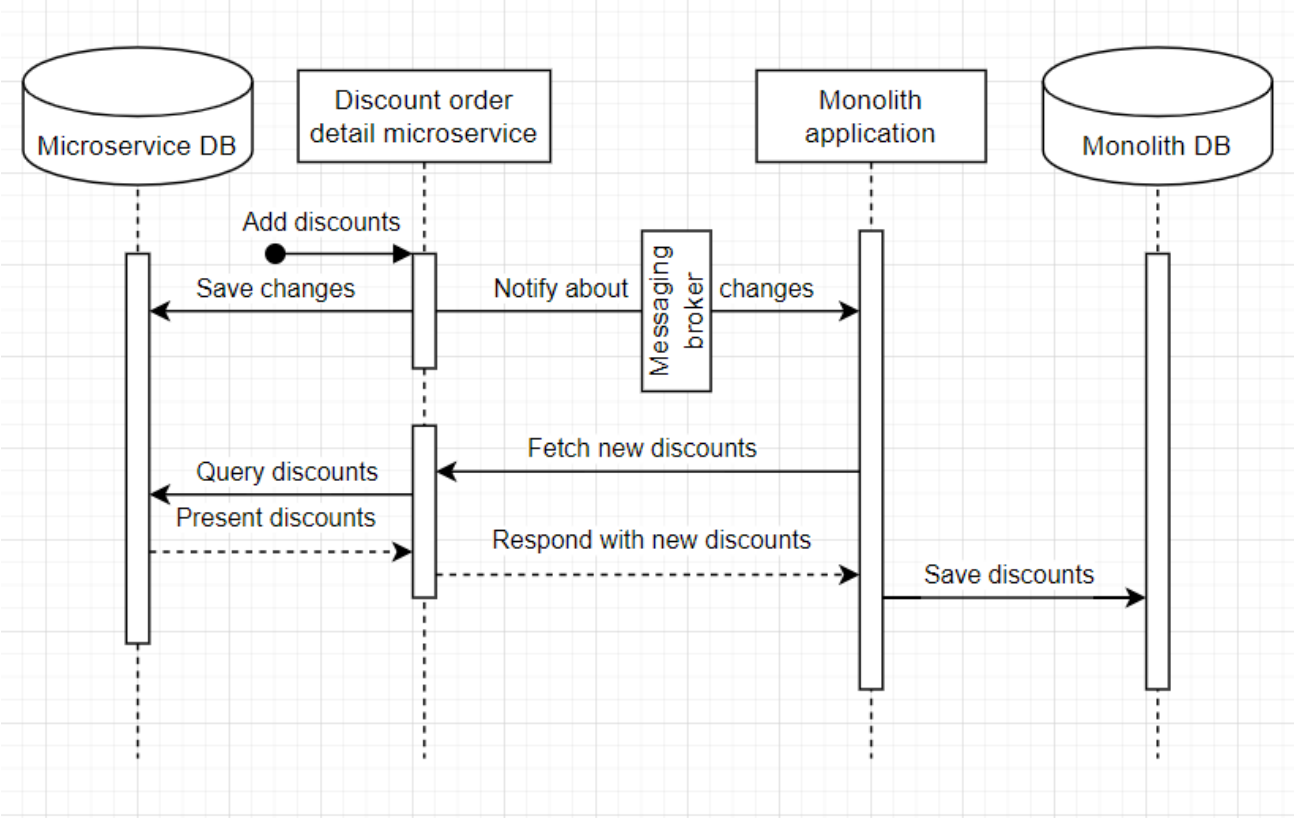

Fig. 14 Process of adding new discounts and ensuring data consistency in a sequence diagram

Third method of transferring data ownership helped us to plan out a migration plan that would ensure smooth transition. Even though the method itself does not mention the actual technical details

of how data consistency should be ensured after the migration, it provides useful guidelines on the general idea of gradual transferring of data ownership. That being said it's important to take into account the business flows of the system, as of this particular example we decided to proceed with having a copy of discount order details in the monolith itself, this might not be the case for other systems. For systems that cannot simply have same data in multiple databases this migration means that performance of existing business flows might degrade if no additional measures are taken.

5. Conclusions and results

5.1. Results

Having analyzed fundamental problems of distributed systems: data consistency and ownership, it becomes apparent that figuring out the right way to slice up persistence layer is one of key challenges in the migration from monolith to microservices.

Establishing clear domain boundaries, assists in finding logical service boundaries that will result in easier work of figuring out the actual functionality of the service as well. Deciding where we want absolute consistency can help to group or merge services that otherwise would have been split for the benefit of conforming to theoretical model.

Additionally, after using analyzed theory and situating it in various data migration patterns and strategies, we determined that different patterns are a great help at minimizing the work needed to figure out data ownership and consistency. Even though there is no solution that fits all situations. After evaluating different approaches, we came to conclusion that methods, analyzed in third section of work, concerning database schema and data tables are of a great assistance when figuring out data ownership and splitting of the database. While methods regarding the actual transfer of ownership and establishing data synchronization, especially Sagas, are a much-needed help in solving data consistency problems.

Practical illustration enabled us to assess examined theory about data ownership and consistency in conjunction with methods and strategies that were supposed to help us work with illustrative database schema. Few technical flaws were identified and mitigated, with evaluation on how these flaws would have impacted functionality of a distributed system.

Pattern of database views was put into practice while trying to reverse engineer legacy table from derived ones, it was identified that the pattern is indeed useful, but effectiveness of the suggested approach revolves around nature of the system and its constraints.

Principle of splitting tables until they make sense to do so, was carried out while contemplating decision to split off card details from payment information table, it was determined that the method does not hold up in certain scenarios and that even though it made sense from a logical standpoint, it did not make sense to apply it in this particular system.

Strategy of gradual transferring of data ownership was put into action when simulating new business requirement to transfer ownership of discount order details to marketing team. Usage of the

method enabled us to define steps for gradual transferring of data ownership, which really eased up the exercise, but in the end did not mention technical details of how data consistency should be established, as that strongly depends on the system itself.

5.2. Conclusions

Finally, after throughout analysis, it becomes clear that vast persistence layer knowledge is indeed a key factor when trying to achieve a successful monolith to microservices migration. This is reinforced by challenges that were studied both from theoretical and from practical standpoint. It becomes apparent that they are a bound to come up during the migration and having extensive knowledge of patterns and strategies is a must to deal with them. That being said, patterns and strategies should be applied carefully, taking into consideration business requirements and characteristics of data before using them, as their effectiveness highly relies on the system context.

Main objective of this paper was achieved, the optimal data model was designed and evaluated via effectiveness of normalization, including high normal forms of new tables, additionally optimization of data storage was measured and great improvements were reached, identified by significant enhancement of data storage utilization.

Additional work is encouraged, especially to investigate usage of saga pattern in practical scenarios. Data consistency challenges and solutions can be tested further with an illustrative system that is more demanding on the availability and consistency of data across the microservices. Data modification problematics of split database schema can also be further analyzed and methods tested.

6. References

[AWC19] Smid Antonin, Ruolin Wang, and Tomas Cerny. "Case study on data communication in microservice architecture." Proceedings of the Conference on Research in Adaptive and Convergent Systems. 2019. https://www.researchgate.net/profile/Tom-Cerny/publication/337200845_Case_study_on_data_communication_in_microservice_architecture/links/5ee2960b92851ce9e7dca840/Case-study-on-data-communication-in-microservice-architecture.pdf

[CY12] Rada Chirkova, Jun Yang. "Materialized views." in *Foundations and Trends® in Databases Volume 4 Issue 4.* Now Publishers, Boston, 2012, pp. 295-405.

[DGL+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara Fabrizio Montesi, Ruslan Mustafin, Larisa Safina. "Microservices: yesterday, today, and tomorrow." in Mazzara, Manuel, and Bertrand Meyer, eds. *Present and ulterior software engineering*. Springer International Publishing, Cham, 2017, pp. 195–216.

[Fag81] Ronald Fagin. "A normal form for relational databases that is based on domains and keys." in *Transactions on Database Systems (TODS) Volume 6(3),* ACM, New York, 1981, pp. 387-415.

[FBZ+19] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, Stefan Wagner. "From monolith to microservices: A classification of refactoring approaches." in J.-M. Bruel, M. Mazzara, and B. Meyer, eds. *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer International Publishing, Cham, 2019, pp. 128–141.

[FFZ+17] Andrei Furda, Colin Fidge, Olaf Zimmermann, Wayne Kelly and Alistair Barros. "Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency." in *IEEE Software 35.3.* 2017, pp. 63-72.

[KM19] Justas Kazanavičius, and Dalius Mažeika. "Migrating legacy software to microservices architecture." in *Open Conference of Electrical, Electronic and Information Sciences (eStream).* IEEE, 2019, pp. 1-5.

[LZS+21] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, Marcos Kalinowski. "Data management in microservices: State of the practice, challenges, and research directions." in *Proc. VLDB Endow., 14 (13).* 2021, pp. 3348-3361.

[Muk19] Sourav Mukherjee. "Indexes in Microsoft SQL Server" arXiv preprint arXiv:1903.08334. 2019. https://arxiv.org/ftp/arxiv/papers/1903/1903.08334.pdf

[NAE16] Alshuqayran Nuha, Nour Ali, and Roger Evans. "A systematic mapping study in microservice architecture." in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2016, pp. 44-51.

[New15] Sam Newman. Building microservices. "O'Reilly Media, Inc.", 2015. Preface xiii-xiv, pp. 1, 34

[New19] Sam Newman. Monolith to microservices: evolutionary patterns to transform your monolith. "O'Reilly Media, Inc.", 2019. pp. 1, 28-32, 128-135.

[PA19] Gianna Panfilo & Carlos F. Arias. “The coordinated universal time (UTC)” in *Metrologia, Volume 56(4).* IOP Publishing, Bristol, 2019, pp. 042001.

[Ric+18] Chris Richardson. Microservices patterns: with examples in Java. Simon and Schuster, 2018. pp. 18-19, 114 - 125

[Rud18] Chaitanya K. Rudrabhatla. "Comparison of event choreography and orchestration techniques in microservice architecture." in *International Journal of Advanced Computer Science and Applications 9.8*. 2018, pp. 18-22.

[GK7] Ganek, Alan, and Kristof Kloeckner. "An overview of IBM service management." in *IBM Systems Journal 46.3*. 2007, pp. 375-385.

[GL12] Seth Gilbert, and Nancy Lynch. "Perspectives on the CAP Theorem." in *Computer 45.2*, Institute of Electrical and Electronics Engineers, New Jersey, 2012, pp. 30-36.

[WDL10] Ting J. Wang, Hui Du, and Constance M. Lehmann. "Accounting for the benefits of database normalization." in *American Journal of Business Education (AJBE) Volume 3(1).* The Clute Institute, Littleton, 2010, pp. 41-52.

7. Appendices

7.1. Appendix 1. Initial POS auditing database schema

```
CREATE TABLE order_detail (
    ID bigint(20) NOT NULL,
    ORDER_ID bigint(20) DEFAULT NULL,
    PRODUCT_ID bigint(20) DEFAULT NULL,
    DISCOUNT_ID bigint(20) DEFAULT NULL,
    DISCOUNT_AMT decimal(14,6) DEFAULT '0.000000',
    QUANTITY decimal(12,4) DEFAULT '0.0000',
    PRICE decimal(14,6) DEFAULT '0.000000',
    TAX decimal(14,6) DEFAULT '0.000000',
    IS_DELETED tinyint(2) DEFAULT '0',
    DISCOUNT_TYPE varchar(2) DEFAULT NULL,
    DISCOUNT_VALUE decimal(7,2) DEFAULT '0.00',
    MERCHANT_ID bigint(20) NOT NULL,
    STORE_ID int(11) DEFAULT '1',
    PARENT_ORDER_DETAIL_ID bigint(20) DEFAULT NULL,
    PRODUCT_NAME varchar(100) DEFAULT NULL,
    CATEGORY_ID bigint(20) DEFAULT NULL,
    ORDER_DETAIL_TYPE tinyint(4) DEFAULT '0',
    TAX_ID bigint(20) DEFAULT NULL,
    IS_INCLUSIVE tinyint(1) DEFAULT '0',
    TAX_NAME varchar(45) DEFAULT NULL,
    DISCOUNT_NAME varchar(45) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    PRODUCT_TYPE tinyint(2) DEFAULT '0',
    PAYMENT_INFO_ID bigint(20) DEFAULT NULL,
    DISCOUNT_CATEGORY tinyint(2) DEFAULT NULL COMMENT '1 - Discount\n2 - Loyalty
Discount',
    MODIFIED_ON timestamp NULL DEFAULT NULL,
    MODIFIED_BY bigint(20) DEFAULT NULL,
    ENTITY_VALUE decimal(10,4) DEFAULT '0.0000',
    ENTITY_ID bigint(20) DEFAULT NULL,
    ENTITY_RATE decimal(10,4) DEFAULT '0.0000',
    ENTITY_NAME varchar(200) DEFAULT NULL,
    PRIMARY KEY (ID),
    KEY ORDER_ID_FK (ORDER_ID),
    KEY PRODUCT_ID_FK (PRODUCT_ID),
    KEY DISCOUNT_ID_FK (DISCOUNT_ID),
    KEY PARENT_ORDER_DETAIL_ID (PARENT_ORDER_DETAIL_ID),
    KEY OD_M_PD_IDX (MERCHANT_ID,PRODUCT_NAME),
    KEY OD_TAX_IDX (TAX_ID),
    KEY OD_MODIFIED_ON_IDX (MODIFIED_ON)
)
```

```
CREATE TABLE orders (
    ID bigint(20) NOT NULL,
    TRANSACTION_CODE varchar(20) DEFAULT NULL,
    CUSTOMER_ID bigint(20) DEFAULT NULL,
    ORDER_AMOUNT decimal(12,4) DEFAULT '0.0000',
    MERCHANT_ID int(11) DEFAULT NULL,
    STORE_ID int(11) DEFAULT NULL,
    TAX_AMOUNT decimal(12,4) DEFAULT '0.0000',
    DISCOUNT_ID bigint(20) DEFAULT NULL,
    ORDER_SUBTOTAL decimal(12,4) DEFAULT '0.0000',
    GLOBAL_DISCOUNT_AMT decimal(12,4) DEFAULT '0.0000',
    ORDER_STATUS tinyint(4) DEFAULT NULL COMMENT '1- PENDING\n2 - PARTIAL PAID\n3 -
PAID',
    IS_DELETED varchar(1) DEFAULT '0',
    TAX_PERCENTAGE decimal(12,4) DEFAULT '0.0000',
    ORDER_CREATED_BY bigint(20) DEFAULT NULL,
    ORDER_LAST_MODIFIED_BY bigint(20) DEFAULT NULL,
    ORDER_CREATION_TIME timestamp NULL DEFAULT NULL,
    ORDER_LAST_UPDATE_TIME timestamp NULL DEFAULT NULL,
    CREATION_TIME_ZONE varchar(75) DEFAULT NULL,
    ORDER_TYPE varchar(10) DEFAULT NULL,
    ORDER_CLOSED_BY bigint(20) DEFAULT NULL,
    ORDER_CLOSED_DATE timestamp NULL DEFAULT NULL,
    PAID_AMOUNT decimal(12,4) DEFAULT '0.0000',
    ITEM_DISCOUNT_AMT decimal(14,6) DEFAULT '0.000000',
    TAXABLE_REVENUE decimal(14,6) DEFAULT '0.000000',
    CHANGE_NUM bigint(20) DEFAULT NULL,
    ORDER_TYPE_ID bigint(20) DEFAULT NULL,
    GRATUITY_AMT decimal(10,4) DEFAULT '0.0000',
    PRIMARY KEY (ID),
    KEY CUSTOMER_ID_FK (CUSTOMER_ID),
    KEY MERCHANT_ID_FK (MERCHANT_ID),
    KEY EMPLOYEE_ID_FK (ORDER_CREATED_BY,ORDER_LAST_MODIFIED_BY),
    KEY DISCOUNT_ID_FK (DISCOUNT_ID),
    KEY O_MID_OCT_IDX (MERCHANT_ID,ORDER_CREATION_TIME),
    KEY O_MID_TC_IDX (MERCHANT_ID,TRANSACTION_CODE),
    KEY O_OCT_IDX (ORDER_CREATION_TIME),
    KEY O_CUST_IDX (CUSTOMER_ID),
    KEY OD_MIDOCD_IDX (MERCHANT_ID,ORDER_CLOSED_DATE),
    KEY O_M_LUD_IDX (MERCHANT_ID,ORDER_LAST_UPDATE_TIME),
    KEY IDX_O_MD (MERCHANT_ID,DISCOUNT_ID),
    KEY IDX_ORDERS_LUTS (ORDER_LAST_UPDATE_TIME),
    KEY O_CLOSEDBY_IDX (ORDER_CLOSED_BY),
    KEY IDX_OSTATUS_IDX (MERCHANT_ID,ORDER_STATUS),
    KEY O_STATUS_IDX (ORDER_STATUS)
)
```

```
CREATE TABLE payment_info (
    ID bigint(20) NOT NULL,
    payment_type varchar(10) DEFAULT NULL,
    CARD_DATA varchar(45) DEFAULT NULL,
    PAYMENT_TRANSACTION_ID bigint(20) DEFAULT '0',
    AMOUNT decimal(12,4) DEFAULT '0.0000',
    MERCHANT_ID bigint(20) DEFAULT NULL,
    ORDER_ID bigint(20) NOT NULL,
    PAYMENT_STATUS tinyint(4) DEFAULT NULL COMMENT '0 - SUCCESS\n1 - FAILED\n2 -
PENDING',
    TRANSACTION_CODE varchar(100) DEFAULT NULL,
    AUTH_CODE varchar(45) DEFAULT NULL,
    PAYMENT_DATE timestamp NULL DEFAULT NULL,
    TRANSACTION_TYPE tinyint(2) DEFAULT NULL,
    CREATION_TIME_ZONE varchar(75) DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    CHANGE_DUE decimal(12,4) DEFAULT '0.0000',
    STORE_ID int(11) DEFAULT '1',
    PAYMENT_PROCESSOR_TRANS_CODE varchar(300) DEFAULT NULL,
    CARD_TYPE varchar(20) DEFAULT NULL,
    CARD_LAST_4DIGIT varchar(7) DEFAULT NULL,
    MODIFIED_BY bigint(20) DEFAULT NULL,
    MODIFIED_ON timestamp NULL DEFAULT NULL,
    CARD_HOLDER_NAME varchar(300) DEFAULT NULL,
    CARD_HOLDER_LAST_NAME varchar(150) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    PRIMARY KEY (ID),
    KEY ORDER_ID_FK (ORDER_ID),
    KEY PAYMENT_TRANSACTION_ID_FK (PAYMENT_TRANSACTION_ID),
    KEY MERCHANT_ID_FK (MERCHANT_ID),
    KEY P_MID_AUTH_IDX (MERCHANT_ID,AUTH_CODE),
    KEY PPID_IDX (PARENT_PAYMENT_INFO_ID),
    KEY PAY_PD_IDX (PAYMENT_DATE),
    KEY PAY_MPD_IDX (MERCHANT_ID,PAYMENT_DATE),
    KEY PAY_MCHN_IDX (MERCHANT_ID,CARD_HOLDER_NAME(255)),
    KEY PAY_MCHLN_IDX (MERCHANT_ID,CARD_HOLDER_LAST_NAME),
    KEY PAY_MODIFIED_ON_IDX (MODIFIED_ON)
)

CREATE TABLE payment_transaction (
    ID bigint(20) NOT NULL AUTO_INCREMENT,
    MERCHANT_ID bigint(20) NOT NULL,
    PAYMENT_PROCESSOR_ID bigint(20) DEFAULT NULL,
    PAYMENT_PROCESSOR_TRANS_CODE varchar(100) DEFAULT NULL,
    STORE_ID bigint(20) DEFAULT NULL,
    TRANSACTION_TIME timestamp NULL DEFAULT NULL,
    TRANACTION_STATUS tinyint(4) DEFAULT NULL COMMENT '0 - SUCCESS\n1 - FAILED',
    USER_ID bigint(20) DEFAULT NULL,
    PAYMENT_TRANSACTION_REQUEST_ID bigint(20) DEFAULT NULL,
    PAYMENT_TRANSACTION_RESPONSE_ID bigint(20) DEFAULT NULL,
    PRIMARY KEY (ID)
)
```

7.2. Appendix 2. Database schema after technical flaw eradication

```
CREATE TABLE order_detail (
    ID bigint(20) NOT NULL,
    ORDER_ID bigint(20) DEFAULT NULL,
    PRODUCT_ID bigint(20) DEFAULT NULL,
    DISCOUNT_ID bigint(20) DEFAULT NULL,
    DISCOUNT_AMT decimal(14,6) DEFAULT '0.000000',
    QUANTITY decimal(12,4) DEFAULT '0.0000',
    PRICE decimal(14,6) DEFAULT '0.000000',
    TAX decimal(14,6) DEFAULT '0.000000',
    IS_DELETED tinyint(2) DEFAULT '0',
    DISCOUNT_TYPE varchar(2) DEFAULT NULL,
    DISCOUNT_VALUE decimal(7,2) DEFAULT '0.00',
    MERCHANT_ID bigint(20) NOT NULL,
    STORE_ID int(11) DEFAULT '1',
    PARENT_ORDER_DETAIL_ID bigint(20) DEFAULT NULL,
    PRODUCT_NAME varchar(100) DEFAULT NULL,
    CATEGORY_ID bigint(20) DEFAULT NULL,
    ORDER_DETAIL_TYPE tinyint(4) DEFAULT '0',
    TAX_ID bigint(20) DEFAULT NULL,
    IS_INCLUSIVE tinyint(1) DEFAULT '0',
    TAX_NAME varchar(45) DEFAULT NULL,
    DISCOUNT_NAME varchar(45) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    PRODUCT_TYPE tinyint(2) DEFAULT '0',
    PAYMENT_INFO_ID bigint(20) DEFAULT NULL,
    DISCOUNT_CATEGORY tinyint(2) DEFAULT NULL COMMENT '1 - Discount\n2 - Loyalty
Discount',
    MODIFIED_ON timestamp NULL DEFAULT NULL,
    MODIFIED_BY bigint(20) DEFAULT NULL,
    ENTITY_VALUE decimal(10,4) DEFAULT '0.0000',
    ENTITY_ID bigint(20) DEFAULT NULL,
    ENTITY_RATE decimal(10,4) DEFAULT '0.0000',
    ENTITY_NAME varchar(200) DEFAULT NULL,
    PRIMARY KEY (ID),
    KEY ORDER_ID_FK (ORDER_ID),
    KEY PRODUCT_ID_FK (PRODUCT_ID),
    KEY DISCOUNT_ID_FK (DISCOUNT_ID),
    KEY PARENT_ORDER_DETAIL_ID (PARENT_ORDER_DETAIL_ID),
    KEY OD_M_PD_IDX (MERCHANT_ID,PRODUCT_NAME),
    KEY OD_TAX_IDX (TAX_ID),
    KEY OD_MODIFIED_ON_IDX (MODIFIED_ON)
)
```

```
CREATE TABLE orders (
    ID bigint(20) NOT NULL,
    TRANSACTION_CODE varchar(20) DEFAULT NULL,
    CUSTOMER_ID bigint(20) DEFAULT NULL,
    ORDER_AMOUNT decimal(12,4) DEFAULT '0.0000',
    MERCHANT_ID int(11) DEFAULT NULL,
    STORE_ID int(11) DEFAULT NULL,
    TAX_AMOUNT decimal(12,4) DEFAULT '0.0000',
    DISCOUNT_ID bigint(20) DEFAULT NULL,
    ORDER_SUBTOTAL decimal(12,4) DEFAULT '0.0000',
    GLOBAL_DISCOUNT_AMT decimal(12,4) DEFAULT '0.0000',
    ORDER_STATUS tinyint(4) DEFAULT NULL COMMENT '1- PENDING\n2 - PARTIAL PAID\n3 -
PAID\n4 - DELETED',
    ORDER_CREATED_BY bigint(20) DEFAULT NULL,
    ORDER_LAST_MODIFIED_BY bigint(20) DEFAULT NULL,
    ORDER_CREATION_TIME timestamp NULL DEFAULT NULL,
    ORDER_LAST_UPDATE_TIME timestamp NULL DEFAULT NULL,
    ORDER_TYPE varchar(10) DEFAULT NULL,
    ORDER_CLOSED_BY bigint(20) DEFAULT NULL,
    ORDER_CLOSED_DATE timestamp NULL DEFAULT NULL,
    PAID_AMOUNT decimal(12,4) DEFAULT '0.0000',
    ITEM_DISCOUNT_AMT decimal(14,6) DEFAULT '0.000000',
    TAXABLE_REVENUE decimal(14,6) DEFAULT '0.000000',
    CHANGE_NUM bigint(20) DEFAULT NULL,
    ORDER_TYPE_ID bigint(20) DEFAULT NULL,
    GRATUITY_AMT decimal(10,4) DEFAULT '0.0000',
    PRIMARY KEY (ID),
    KEY CUSTOMER_ID_FK (CUSTOMER_ID),
    KEY MERCHANT_ID_FK (MERCHANT_ID),
    KEY EMPLOYEE_ID_FK (ORDER_CREATED_BY,ORDER_LAST_MODIFIED_BY),
    KEY DISCOUNT_ID_FK (DISCOUNT_ID),
    KEY O_MID_OCT_IDX (MERCHANT_ID,ORDER_CREATION_TIME),
    KEY O_MID_TC_IDX (MERCHANT_ID,TRANSACTION_CODE),
    KEY O_OCT_IDX (ORDER_CREATION_TIME),
    KEY O_CUST_IDX (CUSTOMER_ID),
    KEY OD_MIDOCD_IDX (MERCHANT_ID,ORDER_CLOSED_DATE),
    KEY O_M_LUD_IDX (MERCHANT_ID,ORDER_LAST_UPDATE_TIME),
    KEY IDX_O_MD (MERCHANT_ID,DISCOUNT_ID),
    KEY IDX_ORDERS_LUTS (ORDER_LAST_UPDATE_TIME),
    KEY O_CLOSEDBY_IDX (ORDER_CLOSED_BY),
    KEY IDX_OSTATUS_IDX (MERCHANT_ID,ORDER_STATUS),
    KEY O_STATUS_IDX (ORDER_STATUS)
)
```

```
CREATE TABLE payment_info (
    ID bigint(20) NOT NULL,
    payment_type tinyint(4) DEFAULT NULL,
    CARD_DATA varchar(45) DEFAULT NULL,
    PAYMENT_TRANSACTION_ID bigint(20) DEFAULT '0',
    AMOUNT decimal(12,4) DEFAULT '0.0000',
    MERCHANT_ID bigint(20) DEFAULT NULL,
    ORDER_ID bigint(20) NOT NULL,
    PAYMENT_STATUS tinyint(4) DEFAULT NULL COMMENT '0 - SUCCESS\n1 - FAILED\n2 -
PENDING',
    TRANSACTION_CODE varchar(100) DEFAULT NULL,
    AUTH_CODE varchar(45) DEFAULT NULL,
    PAYMENT_DATE timestamp NULL DEFAULT NULL,
    TRANSACTION_TYPE tinyint(2) DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    CHANGE_DUE decimal(12,4) DEFAULT '0.0000',
    STORE_ID int(11) DEFAULT '1',
    PAYMENT_PROCESSOR_TRANS_CODE varchar(300) DEFAULT NULL,
    CARD_TYPE varchar(20) DEFAULT NULL,
    CARD_LAST_4DIGIT varchar(7) DEFAULT NULL,
    MODIFIED_BY bigint(20) DEFAULT NULL,
    MODIFIED_ON timestamp NULL DEFAULT NULL,
    CARD_HOLDER_NAME varchar(300) DEFAULT NULL,
    CARD_HOLDER_LAST_NAME varchar(150) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    PRIMARY KEY (ID),
    KEY ORDER_ID_FK (ORDER_ID),
    KEY PAYMENT_TRANSACTION_ID_FK (PAYMENT_TRANSACTION_ID),
    KEY MERCHANT_ID_FK (MERCHANT_ID),
    KEY P_MID_AUTH_IDX (MERCHANT_ID,AUTH_CODE),
    KEY PPID_IDX (PARENT_PAYMENT_INFO_ID),
    KEY PAY_PD_IDX (PAYMENT_DATE),
    KEY PAY_MPD_IDX (MERCHANT_ID,PAYMENT_DATE),
    KEY PAY_MCHN_IDX (MERCHANT_ID,CARD_HOLDER_NAME(255)),
    KEY PAY_MCHLN_IDX (MERCHANT_ID,CARD_HOLDER_LAST_NAME),
    KEY PAY_MODIFIED_ON_IDX (MODIFIED_ON)
)

CREATE TABLE payment_transaction (
    ID bigint(20) NOT NULL AUTO_INCREMENT,
    MERCHANT_ID bigint(20) NOT NULL,
    PAYMENT_PROCESSOR_ID bigint(20) DEFAULT NULL,
    PAYMENT_PROCESSOR_TRANS_CODE varchar(100) DEFAULT NULL,
    STORE_ID bigint(20) DEFAULT NULL,
    TRANSACTION_TIME timestamp NULL DEFAULT NULL,
    TRANACTION_STATUS tinyint(4) DEFAULT NULL COMMENT '0 - SUCCESS\n1 - FAILED',
    USER_ID bigint(20) DEFAULT NULL,
    PAYMENT_TRANSACTION_REQUEST_ID bigint(20) DEFAULT NULL,
    PAYMENT_TRANSACTION_RESPONSE_ID bigint(20) DEFAULT NULL,
    PRIMARY KEY (ID)
)
```

7.3. Appendix 3. Database schema after ORDER_DETAIL split

```
CREATE TABLE order_discount_detail (
    ID bigint(20) NOT NULL,
    IS_DELETED tinyint(2) DEFAULT '0',
    MERCHANT_ID bigint(20) NOT NULL,
    STORE_ID int(11) DEFAULT '1',
    CATEGORY_ID bigint(20) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    DISCOUNT_TYPE varchar(2) DEFAULT NULL, COMMENT '(FIXED (amount), PERCENTAGE, BOGO
(buyonegetone))'
    DISCOUNT_VALUE decimal(7, 2) DEFAULT '0.00',
    DISCOUNT_NAME varchar(45) DEFAULT NULL,
    DISCOUNT_CATEGORY tinyint(2) DEFAULT NULL COMMENT '1 - Discount\n2 - Loyalty
Discount',
    PRIMARY KEY (ID)
)

CREATE TABLE order_entity_detail (
    ID bigint(20) NOT NULL,
    IS_DELETED tinyint(2) DEFAULT '0',
    MERCHANT_ID bigint(20) NOT NULL,
    STORE_ID int(11) DEFAULT '1',
    CATEGORY_ID bigint(20) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    ENTITY_NAME varchar(200) DEFAULT NULL,
    ENTITY_RATE decimal(10, 4) DEFAULT '0.0000',
    ENTITY_TYPE tinyint(2) DEFAULT '1' COMMENT 'enum type: 0: NONE, 1: Percentage, 2:
FIXED AMOUNT',
    PRIMARY KEY (ID)
)

CREATE TABLE order_tax_detail (
    ID bigint(20) NOT NULL,
    IS_DELETED tinyint(2) DEFAULT '0',
    MERCHANT_ID bigint(20) NOT NULL,
    STORE_ID int(11) DEFAULT '1',
    CATEGORY_ID bigint(20) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    TAX_NAME varchar(45) DEFAULT NULL,
    TAX_RATE decimal(12,4) NOT NULL DEFAULT '0.0000',
    TAX_TYPE tinyint(2) DEFAULT '1' COMMENT 'enum type: 0: NONE, 1: Percentage, 2: FIXED
AMOUNT',
    IS_INCLUSIVE tinyint(1) DEFAULT '0', COMMENT '0: not included in price\n1: price
include tax',
    PRIMARY KEY (ID)
)

CREATE TABLE order_item (
    ID bigint(20) NOT NULL,
    ORDER_ID bigint(20) DEFAULT NULL,
    IS_DELETED tinyint(2) DEFAULT '0',
    MERCHANT_ID bigint(20) NOT NULL,
```

```
    STORE_ID int(11) DEFAULT '1',
    CATEGORY_ID bigint(20) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    PAYMENT_INFO_ID bigint(20) DEFAULT NULL,
    MODIFIED_ON timestamp NULL DEFAULT NULL,
    MODIFIED_BY bigint(20) DEFAULT NULL,
    QUANTITY decimal(12, 4) DEFAULT '0.0000',
    PRICE decimal(14, 6) DEFAULT '0.000000',
    PRODUCT_NAME varchar(100) DEFAULT NULL,
    PRODUCT_TYPE tinyint(2) DEFAULT '0',
    PRODUCT_ID bigint(20) DEFAULT NULL,
    DISCOUNT_ID bigint(20) DEFAULT NULL,
    DISCOUNT_AMT decimal(14, 6) DEFAULT '0.000000',
    TAX_ID bigint(20) DEFAULT NULL,
    TAX decimal(14, 6) DEFAULT '0.000000',
    ENTITY_ID bigint(20) DEFAULT NULL,
    ENTITY_VALUE decimal(10, 4) DEFAULT '0.0000',
    PRIMARY KEY (ID),
    KEY ORDER_ID_FK (ORDER_ID),
    KEY ORDER_ID_IDX (ORDER_ID),
    KEY PRODUCT_ID_FK (PRODUCT_ID),
    KEY DISCOUNT_ID_FK (DISCOUNT_ID),
    KEY TAX_ID_FK (TAX_ID),
    KEY ENTITY_ID_FK (ENTITY_ID),
    KEY OD_MODIFIED_ON_IDX (MODIFIED_ON),
    KEY OD_DISCOUNT_IDX (DISCOUNT_ID),
    KEY OD_TAX_IDX (TAX_ID),
    KEY OD_ENTITY_IDX (ENTITY_ID)
)

CREATE TABLE orders (
    ID bigint(20) NOT NULL,
    TRANSACTION_CODE varchar(20) DEFAULT NULL,
    CUSTOMER_ID bigint(20) DEFAULT NULL,
    ORDER_AMOUNT decimal(12,4) DEFAULT '0.0000',
    MERCHANT_ID int(11) DEFAULT NULL,
    STORE_ID int(11) DEFAULT NULL,
    ORDER_SUBTOTAL decimal(12,4) DEFAULT '0.0000',
    ORDER_STATUS tinyint(4) DEFAULT NULL COMMENT '1- PENDING\n2 - PARTIAL PAID\n3 -
PAID\n4 - DELETED',
    ORDER_CREATED_BY bigint(20) DEFAULT NULL,
    ORDER_LAST_MODIFIED_BY bigint(20) DEFAULT NULL,
    ORDER_CREATION_TIME timestamp NULL DEFAULT NULL,
    ORDER_LAST_UPDATE_TIME timestamp NULL DEFAULT NULL,
    ORDER_TYPE varchar(10) DEFAULT NULL,
    ORDER_CLOSED_BY bigint(20) DEFAULT NULL,
    ORDER_CLOSED_DATE timestamp NULL DEFAULT NULL,
    PAID_AMOUNT decimal(12,4) DEFAULT '0.0000',
    ITEM_DISCOUNT_AMT decimal(14,6) DEFAULT '0.000000',
    TAXABLE_REVENUE decimal(14,6) DEFAULT '0.000000',
    CHANGE_NUM bigint(20) DEFAULT NULL,
    ORDER_TYPE_ID bigint(20) DEFAULT NULL,
    GRATUITY_AMT decimal(10,4) DEFAULT '0.0000',
    DISCOUNT_ID bigint(20) DEFAULT NULL,
    GLOBAL_DISCOUNT_AMT decimal(12,4) DEFAULT '0.0000',
    TAX_ID bigint(20) DEFAULT NULL,
```

```
    TAX_AMOUNT decimal(12,4) DEFAULT '0.0000',
    ENTITY_ID bigint(20) DEFAULT NULL,
    ENTITY_VALUE decimal(10, 4) DEFAULT '0.0000',
    PRIMARY KEY (ID),
    KEY CUSTOMER_ID_FK (CUSTOMER_ID),
    KEY MERCHANT_ID_FK (MERCHANT_ID),
    KEY EMPLOYEE_ID_FK (ORDER_CREATED_BY,ORDER_LAST_MODIFIED_BY),
    KEY DISCOUNT_ID_FK (DISCOUNT_ID),
    KEY ENTITY_ID_FK (ENTITY_ID),
    KEY TAX_ID_FK (TAX_ID),
    KEY O_MID_OCT_IDX (MERCHANT_ID,ORDER_CREATION_TIME),
    KEY O_MID_TC_IDX (MERCHANT_ID,TRANSACTION_CODE),
    KEY O_OCT_IDX (ORDER_CREATION_TIME),
    KEY O_CUST_IDX (CUSTOMER_ID),
    KEY OD_MIDOCD_IDX (MERCHANT_ID,ORDER_CLOSED_DATE),
    KEY O_M_LUD_IDX (MERCHANT_ID,ORDER_LAST_UPDATE_TIME),
    KEY IDX_O_MD (MERCHANT_ID,DISCOUNT_ID),
    KEY IDX_ORDERS_LUTS (ORDER_LAST_UPDATE_TIME),
    KEY O_CLOSEDBY_IDX (ORDER_CLOSED_BY),
    KEY IDX_OSTATUS_IDX (MERCHANT_ID,ORDER_STATUS),
    KEY O_STATUS_IDX (ORDER_STATUS)
)

CREATE TABLE payment_info (
    ID bigint(20) NOT NULL,
    payment_type tinyint(4) DEFAULT NULL,
    CARD_DATA varchar(45) DEFAULT NULL,
    PAYMENT_TRANSACTION_ID bigint(20) DEFAULT '0',
    AMOUNT decimal(12,4) DEFAULT '0.0000',
    MERCHANT_ID bigint(20) DEFAULT NULL,
    ORDER_ID bigint(20) NOT NULL,
    PAYMENT_STATUS tinyint(4) DEFAULT NULL COMMENT '0 - SUCCESS\n1 - FAILED\n2 -
PENDING',
    TRANSACTION_CODE varchar(100) DEFAULT NULL,
    AUTH_CODE varchar(45) DEFAULT NULL,
    PAYMENT_DATE timestamp NULL DEFAULT NULL,
    TRANSACTION_TYPE tinyint(2) DEFAULT NULL,
    CREATED_BY bigint(20) DEFAULT NULL,
    CHANGE_DUE decimal(12,4) DEFAULT '0.0000',
    STORE_ID int(11) DEFAULT '1',
    PAYMENT_PROCESSOR_TRANS_CODE varchar(300) DEFAULT NULL,
    CARD_TYPE varchar(20) DEFAULT NULL,
    CARD_LAST_4DIGIT varchar(7) DEFAULT NULL,
    MODIFIED_BY bigint(20) DEFAULT NULL,
    MODIFIED_ON timestamp NULL DEFAULT NULL,
    CARD_HOLDER_NAME varchar(300) DEFAULT NULL,
    CARD_HOLDER_LAST_NAME varchar(150) DEFAULT NULL,
    CREATED_ON timestamp NULL DEFAULT NULL,
    PRIMARY KEY (ID),
    KEY ORDER_ID_FK (ORDER_ID),
    KEY PAYMENT_TRANSACTION_ID_FK (PAYMENT_TRANSACTION_ID),
    KEY MERCHANT_ID_FK (MERCHANT_ID),
    KEY P_MID_AUTH_IDX (MERCHANT_ID,AUTH_CODE),
    KEY PPID_IDX (PARENT_PAYMENT_INFO_ID),
    KEY PAY_PD_IDX (PAYMENT_DATE),
    KEY PAY_MPD_IDX (MERCHANT_ID,PAYMENT_DATE),
    KEY PAY_MCHN_IDX (MERCHANT_ID,CARD_HOLDER_NAME(255)),
```

```
    KEY PAY_MCHLN_IDX (MERCHANT_ID,CARD_HOLDER_LAST_NAME),
    KEY PAY_MODIFIED_ON_IDX (MODIFIED_ON)
)

CREATE TABLE payment_transaction (
    ID bigint(20) NOT NULL AUTO_INCREMENT,
    MERCHANT_ID bigint(20) NOT NULL,
    PAYMENT_PROCESSOR_ID bigint(20) DEFAULT NULL,
    PAYMENT_PROCESSOR_TRANS_CODE varchar(100) DEFAULT NULL,
    STORE_ID bigint(20) DEFAULT NULL,
    TRANSACTION_TIME timestamp NULL DEFAULT NULL,
    TRANACTION_STATUS tinyint(4) DEFAULT NULL COMMENT '0 - SUCCESS\n1 - FAILED',
    USER_ID bigint(20) DEFAULT NULL,
    PAYMENT_TRANSACTION_REQUEST_ID bigint(20) DEFAULT NULL,
    PAYMENT_TRANSACTION_RESPONSE_ID bigint(20) DEFAULT NULL,
    PRIMARY KEY (ID)
)
```

7.4. Appendix 4. Storage space optimization calculations.

Storage space was calculated using total amount of data: 1.4 TB and 10,374,451,263 rows in order_detail table.

Sample of 100000 entries was taken to get approximate amount of total, unique and duplicate tax rows.

SQL used:

```
with all_rows as (
     SELECT TOP 100000, ORDER_DETAIL_TYPE, TAX_ID
     FROM  order_detail
)
SELECT COUNT(*) as total_tax_rows (10347)
FROM all_rows
WHERE ORDER_DETAIL_TYPE = 2 (2 for tax)

SELECT COUNT(*) as unique_tax_rows (534)
FROM all_rows
WHERE ORDER_DETAIL_TYPE = 2
GROUP BY TAX_ID
```

Duplicates were calculated by subtracking total_unique_tax_rows from total_tax_rows, being 9813. With that we calculated approximate percentage of duplicate tax rows, 9813 divided by 100000, giving us 9.813 %.

Estimated kilobytes saved per duplicate tax row is 0.095 KB. Approximate amount of duplicate tax rows is 9.813 % of all rows in order_detail table, 1018044902. Approximate space saved is 96714265.73KB, accounting for 6.908% of total space used to store order_detail table.

7.5. Appendix 5. Pseudo API endpoints and messaging event

```
POST /discount #Add new discount order detail
Request body application/json
{
    "merchant_id": integer,
    "store_id": integer,
    "category_id": integer,
    "created_by": integer,
    "discount_type": string (Enum ['fixed', 'percentage', 'bogo']),
    "discount_value": decimal,
    "discount_name": string,
    "discount_category": int (Enum [1 - 'Discount', 2 - 'Loyalty Discount'])
}
Response body application/json
{
    "id": integer,
    "merchant_id": integer,
    "store_id": integer,
    "category_id": integer,
    "created_on": string (datetime)
    "created_by": integer,
    "discount_type": string (Enum ['fixed', 'percentage', 'bogo']),
    "discount_value": decimal,
    "discount_name": string,
    "discount_category": int (Enum [1 - 'Discount', 2 - 'Loyalty Discount'])
}

GET /discount/{id:integer} #Get discount order detail by id
Response body application/json
{
    "id": integer,
    "merchant_id": integer,
    "store_id": integer,
    "category_id": integer,
    "created_on": string (datetime)
    "created_by": integer,
    "discount_type": string (Enum ['fixed', 'percentage', 'bogo']),
    "discount_value": decimal,
    "discount_name": string,
    "discount_category": int (Enum [1 - 'Discount', 2 - 'Loyalty Discount'])
}
```

```
GET /discounts/{merchant_id:integer}/{store_id:integer}/{fromdate:string (datetime)}
#Get discounts for merchant, store and date created from
[
    {
        "id": integer,
        "merchant_id": integer,
        "store_id": integer,
        "category_id": integer,
        "created_on": string (datetime)
        "created_by": integer,
        "discount_type": string (Enum ['fixed', 'percentage', 'bogo']),
        "discount_value": decimal,
        "discount_name": string,
        "discount_category": int (Enum [1 - 'Discount', 2 - 'Loyalty Discount'])
    }
]

DiscountAddedEvent #Identifies that one new single discount was added
Payload:
{
    "discount_id": integer,
    "merchant_id": integer (default NULL),
    "store_id": integer (default NULL)
}

DiscountsAddedEvent #Identifies that bundle of discounts was added
Payload:
{
    "merchant_id": integer,
    "store_id": integer,
    "created_on": string (datetime)
}
```