**Table of content**

[1 Introduction 2](#_Toc369899742)

[2 Requirements 3](#_Toc369899743)

[2.1 Wholesaler 3](#_Toc369899744)

[2.2 Manufacturer 3](#_Toc369899745)

[2.3 Web-shop 3](#_Toc369899746)

[2.4 Web-shop customer 4](#_Toc369899747)

[2.5 Scenarios 4](#_Toc369899748)

[2.5.1 Purchasing from manufactorer 4](#_Toc369899749)

[2.5.2 Customer purchasing product 4](#_Toc369899750)

[3 Design 5](#_Toc369899751)

[3.1 Choosing a database type 5](#_Toc369899752)

[3.2 Who is going to use it 6](#_Toc369899753)

[3.3 How is it going to be used 6](#_Toc369899754)

[3.4 Relational design / ER-diagram 7](#_Toc369899755)

[3.4.1 Overall ER-diagram 8](#_Toc369899756)

[3.4.2 Customer relationship diagram 10](#_Toc369899757)

[3.4.3 Manufacturer relationship diagram 12](#_Toc369899758)

[3.4.4 Product attribute diagram 14](#_Toc369899759)

[3.4.5 Web-shop attribute diagram 14](#_Toc369899760)

[3.4.6 Pricing plan attribute diagram 15](#_Toc369899761)

[3.5 Implementation 15](#_Toc369899762)

[3.5.1 Physical design 16](#_Toc369899763)

[3.6 Functional dependencies and normal form 16](#_Toc369899764)

[4 Requirement traceability 19](#_Toc369899765)

[5 Queries and views 20](#_Toc369899766)

[5.1 Best selling product 20](#_Toc369899767)

[5.1.1 Example result 20](#_Toc369899768)

[5.2 Best selling product with name 20](#_Toc369899769)

[5.2.1 Example result 20](#_Toc369899770)

[5.3 Most purchasing customer (money, no quantity discount) 21](#_Toc369899771)

[5.3.1 Example result 21](#_Toc369899772)

[5.4 Biggest Wholesaler profit margin product (one unit purchased) 21](#_Toc369899773)

[5.5 Biggest Web-shop profit margin product (no individual discount) 21](#_Toc369899774)

[5.5.1 Example result 21](#_Toc369899775)

[5.6 Products not sold for over 1 month 22](#_Toc369899776)

[5.7 Wholesaler profit 22](#_Toc369899777)

[5.8 Invoice view 22](#_Toc369899778)

[6 Performance and indexes 22](#_Toc369899779)

[7 Transactions 23](#_Toc369899780)

[8 Conclusion 23](#_Toc369899781)

[9 Appendix 24](#_Toc369899782)

[9.1 Files 24](#_Toc369899783)

[9.2 Large database installation manual 24](#_Toc369899784)

[9.3 Tool 25](#_Toc369899785)

# Introduction

This report details the mini-project of Group 2. The report details the work which has been done to arrive at our final product. The focus of the report is on Database management System, and the formal semantics for a complete project has not been followed. There is no traceability for the requirements and design considerations, just like an acceptance test has not been devised.

The project takes its origin in a larger wholesaler who sells low end to mid-range cloth to different retailers. The wholesaler carries both their own brands (no-name) as well as brand names. The company is having a problem with falling sales as more and more clothing purchases are moving to the internet.

A third party person has contacted the company with a project proposal. If the wholesaler allows third party web-shops access to their distribution channels and product portfolio as well as makes it easy for the third-party web-shops to integrate with the wholesaler, then it is believed that it will be possible to boost sales considerably.

For this reason the wholesaler has contracted us to specify and design a database which allows multiple web-shops to integrate directly with the wholesaler’s product portfolio as well as allow for handling online payments and order tracking. At the same time the wholesaler would like their existing warehouse database replaced to integrate with the new system.

# Requirements

The database is used by a very easily defined group of people, each with their own requirements to the system:

* Wholesaler
* Manufacturer
* Web-shop
* Web-shop customer

## Wholesaler

1. The wholesaler must have unlimited access to the entire database
2. The wholesaler must be able to keep track of warehouse stock.
3. The wholesaler must be able to keep track of sales and extract assorted business statistics.
4. The wholesaler has only a single warehouse and the database should not support multiple warehouses.

## Manufacturer

1. A manufacturer will produce products to the wholesaler’s warehouse.
2. It must be possible for a wholesaler to agree on a price for a product in advance.
3. The price of a product may be dependent on the amount or a specific discount etc.
4. The wholesaler must be able to track the progress of a delivery from a manufacturer from order until it arrives at the warehouse.
5. It is not required to be able to trace a piece of clothing back to the manufacturer once it has been accepted at the warehouse.
6. Multiple manufacturers may produce the same product.
7. It must be possible to have a product which exists in an assortment of variants (colour, pattern, ...)
8. It must be possible for a manufacturer to assign assorted attributes to the products like brand, colour, ...

## Web-shop

1. A third party must be able to build a web-shop around the interface provided by the wholesaler (the database).
2. The database must be able to handle online payments (DIBS/NETS).
3. It must be possible to track the progress of a delivery to a customer from order until it is shipped from the warehouse.
4. A web-shop may carry only a subset of the wholesaler’s portfolio.
5. It must be possible for the wholesaler to arrange different prices for different products with different web-shops
6. It must be possible for the wholesaler to set up different discount for the web-shop based on purchase amount, etc.
7. A web-shop may only see the products it carries.
8. A web-shop may only see the customers created inside the given web-shop.
9. A web-shop may only see order data relating to its own purchases and its own customers purchases.
10. It must be possible for the web-shop to define custom attributes which are stored with the customer.

## Web-shop customer

1. A web-shop must be able to create a customer with name, address, age, ...
2. It must be possible for the web-shop to arrange different prices for different products.
3. It must be possible for the web-shop to set up different discount for its customers based on purchase amount, etc.
4. It must be possible for a customer to use online payment (DIBS/PayPal/NETS/...).
5. It must be possible for a customer to pay via invoice.
6. It must be possible to handle customers returning purchased products.
7. It must be possible to handle customer complaints.
8. It must only be possible for a customer to see the order information relation to him- or herself.
9. The customer must never experience a delay of a database query of more than 1 second 99,98% of the time (from DB received the SQL until the response is ready to be sent).

## Scenarios

This section details different common scenarios of use.

### Purchasing from manufactorer

1. The wholesaler places an order with the manufactorer supplying a reference number and enters the order in ManufactorerOrders.
2. The manufactorer sends an order confirmation to the wholesaler who verifies it and enters it in the ManufactorerOrderConfirmations.
3. The manufacturer sends an invoice to the Wholesaler who verifies it and enters it in the ManufactorerInvoices.
4. When the order is received the shipping manifest is compared to the order and the products are added to the ManufactorerReceptions
5. When the invoice is to be paid the payment is made and the ManufactorerInvoices is updated to reflect that the payment is done.

### Customer purchasing product

1. The customer places an order with the web-shop and an entry is added to the CustomerOrders.
   1. If the customer pays with a credit card a NETS interface is used to validate the card and an ID is generated. This ID is used later when the money is to be transferred (may also be used in case the order is returned.
2. An order confirmation is generated and sent to the customer.
3. The order is packaged and shipped and the shipping manifest is added to CustomerDeliveries
4. An invoice is sent to the customer and it is entered in the CustomerInvoices
5. If the customer paid via credit card the money is withdrawn from the customer’s accont and the CustomerInvoices is updated accordingly.
6. If the customer pays the invoice manually the CustomerInvoices is updated when the money is received.

# Design

When designing the database it is important to not only look at the requirements, but also to consider the domain in which the database is expected to be used.

Before delving into the database design some of the more overall considerations will be considered.

1. Who is going to use it?
2. How is it going to be used?
3. What guarantees can we have about the input?
4. What performance requirements are there?
5. What other layers are there apart from the database?

## Choosing a database type

Most traditional databases are normalized. This means that data preferably only exist once and are linked via keys. This is referred to as normalized data. The advantage of this format is that it has a low footprint as duplicate data only exist in form of keys. To retrieve the data the different tables must be joined. Unfortunately the joining operations are costly, and if the throughput is very large, normalized data is actually not preferable.

Another important consideration is consistency. Traditional databases use ACID (Atomicity, Consistency, Isolation, Durability), which ensures that when data is written to the database subsequent requests is guaranteed to retrieve the new update. It also ensures that simultaneous updates are protected from each other and that transactions are supported. Unfortunately this form of access is also expensive, and sometimes not needed. An alternative is BASE (Basically Available, Soft state, Eventual consistency). Sometimes it is OK that we simply know that an update will eventually be completed, and that inconsistencies are OK for a limited time. This form of database is also used for very large project, and often combined with a distributed database. It often relies on NoSQL (Not Only SQL) as opposed to SQL.

As we are designing a database for a larger cloth wholesaler which includes payments, it is beneficial to have ACID ensured, however it would be quite possible to implement it on BASE. Also, since the database is to be used by a limited number of simultaneous users (< 10000000) an ACID DB should be sufficient, and normalized data will also be acceptable, and the databases indexing and performance optimization (keeping some tables joined in RAM to improve performance) should be sufficient.

## Who is going to use it

If the database is only used by fully trusted personnel then there is no need to place limitations on the access, however if the database is to be accessed by people we do not trust, then it is important to ensure that the users are not able to access more than he or she is allowed to.

Access to the database is not directly part of the SQL standard, but most database implementation works with Users, permissions and possibly Roles. The normal implementation allows for limiting a user’s permissions to one or more specific tables or views as well as whether the user is allowed to insert, update or select. Unfortunately this is insufficient when data from multiple individuals exist in the same table and it is therefore necessary to add another layer of protection. This could be on the form of a REST service with its own layer of authentication and exposing a limited interface to the user.

An alternative is the database principle of Row Level Security (RLS), which basically allows a database to limit access to a given row of a database based on the content of an attribute.

RLS is a very interesting, but though there is an implementation in a patch to postgresql, it has been decided to move the limitation on the row level to a layer above the database. The design of the database will however be in such a way that it make it simple to create a layer on top for row level authentication.

In the tables were RLS is required a GUID attribute is added to each row. This GUID is linked to a user, and the higher level authentication ensures that all quires includes a “WHERE GUID=”...”. If e.g. a 128bit GUID is used then it is infeasible that anyone can guess the correct GUID, and the limitation on the upper level authentication is simply to ensure that the where-clause is included, as that is not possible to enforce using standard SQL users, roles and privileges.

There is a special work-around which involves having all requests to the database go through functions, as functions may enforce a WHERE-clause, but this use of transactional SQL one every query has a high performance penalty and is not very “pretty” from a design perspective.

As a minor note, if the users are allowed to insert arbitrary data it is important to encode the text to prevent them from inserting SQL-statements inside the text.

## How is it going to be used

As mentioned before if the database is used through a predefined interface,e it is simple to limit the access based on authentication. We are going to attempt to create the database so it may be access through the normal postgresql access and authentication, so the database may simply be exposed “online”, yet with the simple pre-processing of validating that the WHERE-clause with the GUID is include for the shared tables.

The actual implementation of the layer above the database which enforces this WHERE-clause will not be a part of the project, and it will simply be assumes that it is included where appropriate.

## Relational design / ER-diagram

The relational design may be done from the requirements and the overall design considerations. A way of representing the design at an overall level is using an ER-diagram.

The ER-diagram suffers from the same problem as UML, that the diagrams very quickly become extremely large and detailed.

For this reason we have decided to split the ER diagrams up into a collection of diagrams focussing on a specific subsection of the diagrams, and increasing in detail.

* Overall ER-diagram
* Customer relationship diagram
* Manufacturer relationship diagram
* Product relationship diagram
* Web-shop relationship diagram
* PricingPlan relationship diagram.

There are many different proposed notations for the ER-diagram, but in this report the one presented by Katja Hose will be used. This website also seems to include the same notation as the one used by Katja, and we have used it as a reference: <http://jcsites.juniata.edu/faculty/rhodes/dbms/ermodel.htm> . This site do not include the indication of 1-1, 1-N and N-M which is used in the diagrams in this report. We will solely use the notation of a single number or letter and not the N..M notation to avoid confusion. We have not found any way to indicate optional attributes, so this may only be seen in the implementation.

### Overall ER-diagram



In the above may be found the following entities. The PricingPlan entity is represented three times, even though it is the same entity, just involved in three different relationships. It has been included all three times for clarity, instead of reusing the same entity and just have more lines.

* Product

*The product represents a specific product which is sold by the Wholesaler. The Product is supplied by one or more Manufacturers and is carried by zero or more web-shops.*

* Product Attributes

*Special attributes associated with one or more products, e.g. product line, colour, size, ...*

* Manufacturer

*The Manufacturer sells one or more products to the Wholesaler.*

* Manufactorer PricingPlan

*Indicates the pricing plan as negotiated with the Manufacturer. It includes price, terms of delivery, discounts, ...*

* Web-shop

*The web-shop carries one or more products from the Wholesalers assortment and offers the products to their customers.*

* Web-shop PricingPlan

*For each product that the web-shop carries a price, discount, etc. must be agreed on and this is encompassed in the WPricingPlan. It is the price and conditions at which the Wholesaler sells the products to the Webshop.*

* Customer

*The Customer is the end user purchasing products from the Web-shop.*

* Product PricingPlan

*For each product carried by the Web-shop a default price, discount, etc. must exist which details the price the Web-shop offers the given product to its customers. It is the price and conditions at which the Web-shop sells the products to the Customer, if the customer does not have individual conditions.*

* Customer PricingPlan

*A Customer may be negotiated special conditions, discounts or prices with the Web-shop, and this is expressed in the Customer PricingPlan. The Customer PricingPlan supersedes the Product PricingPlan, unless the Product PricingPlan is better (maybe due to a special offer). The Customer ProcingPlan may also have a special case where it does not have a price, in which case it means a discount compared to the Product PricingPlan. Another special case is if it has a reference to no products, which means it is valid for all products.*

#### Notes

1. It has been considered whether it should be possible to have a group of products which are priced together. This may be used if the same T-shirt exists in 8 different sizes and 12 colours yet all having the same price. In that case it might make sense to be able to group these 96 products and assign a fixed price. It should naturally still be possible to identify a specific product (colour and size) and put it on sale, but reversely if the Manufacturer changes the price for this T-shirt it would be required to update 96 rows. Naturally this may be done by in a single query by using the attribute. This grouping of products using a dedicated entity has been postponed for now.
2. It has been considered whether there should be a Warehouse entity, even though there can be only one.
3. The disadvantage of a generic attribute design is that there is no strong type safety (Green vs. Green), just like duplicate attribute types may exist (Colour vs. Color). By predefining these attributes they may be strongly typed and duplicates may more easily be avoided, but at the same time it is impossible to predict all possible attributes. This generic approach with non type-safe attributes goes against the relational design principle, and should be avoided, but it is sometimes necessary. It is much more suited for the NoSQL design using documents, but that is a different story.
4. Customer pricing plan is presently not implemented

### Customer relationship diagram



In the above may be found the following entities not already described.

* CustomerAttributes

*Custom key-value pair which may be added to the customer by the web-shop.*

* CustomerOrders

*The CustomerOrders is a collection of the customer orders. When a Customer places an order with a Web-shop an entry in CustomerOrders is created detailing the order including which products and at what price (including the pricing plan so it may be detailed on the invoice why the price is as it is).*

* PricingPlan

*The PricingPlan is the price, conditions, etc. that the given product was purchased at (snapshot of the Customer PricingPlan or Product PricingPlan, as it should not update with these).*

* NETS Payment

*The NETS Payment contains the payment details if the customer chooses to play with a credit card. This is important to cache, as the money may not the drawn from the Customer account until the product has been shipped from the warehouse.*

* CustomerOrderConfirmation

*The Customer Order Confirmation is generated after the Customer Order has been validated and confirms to the Customer that the order has been accepted and is also used as a packing list for the warehouse.*

* CustomerInvoices

*The CustomerInvoice details the monetary part of the order, including whether the order has been paid (credit card), and if not the payment conditions.*

* CustomerDeliveries

*The CustomerDeliveries is updated when the product is shipped from the warehouse. This should also trigger the payment of the NETS registration (if payment by credit-card was chosen) as well as the generation of the invoice. This should also trigger an automatic update of the instock attribute of the product.*

#### Notes

1. In some situations it may make sense to perform a partial shipment so the customer may receive some of the order first and the rest later. This option is not supported in this design, and here the order is always shipped together.
2. If the PricingPlan contains information about delivery and these are conflicting then the most beneficial for the customer is chosen.

### Manufacturer relationship diagram



In the above may be found the following entities not already described.

* ProductsOrders

*The ProductsOrders is a collection of the manufactorer orders. When the Wholesaler places an order with a Manufactorer an entry in ProductsOrders is created detailing the order including which products and at what price (including the pricing plan so it may be detailed on the invoice why the price is as it is).*

* PricingPlan

*The PricingPlan is the price, conditions, etc. that the given product was ordered at (snapshot of the Manufactorer PricingPlan, as it should not update with this).*

* ManufactorerOrderConfirmation

*The Manufactorer Order Confirmation is generated after the Manufactorer Order has been validated and confirms from the Manufactorer.*

* ManufactorerInvoices

*The ManufactorerInvoices details the monetary part of the order, including whether the order has been paid, and if not the payment conditions.*

* ManufactorerDeliveries

*The ManufactorerDeliveries is updated when the products are received at the warehouse. This should also trigger an automatic update of the instock attribute of the product.*

#### Note

We must consider how the delivery cost is determined. Perhaps the delivery terms, which are a value now, should be an entity? This allows for delivery price and time estimates and price per kg.

### Product attribute diagram



#### Notes

1. The instock attribute…

### Web-shop attribute diagram



#### Notes

1. When a customer purchases a product then so does the Web-shop (automatically). This purchase, or rather the PricingPlan at the time of the purchase is not recorded anywhere.

### Pricing plan attribute diagram



#### Notes

1. As the pricing plans are shared by multiple products and web-shops it is imperative that the PricingPlan table is immutable, meaning that it is impossible to update a row, and delete may only be performed when there are no more references. This can be done by disallowing update and ensuring by foreign key constraint that the row is deleted prematurely.
2. The PricingPlan is very special as it is reused. It therefore relates to zero or more products, and either a customer, webshop, manufacturer, customerorder and/or manufactorerorder.
3. The special case with zero product references is only used if price is 0 (discount only), and indicates that the PricingPlan is a discount only, and may only be used in the relation with a customer and indicates a discount compared to the normal webshop price. If the PricingPlan also has no product relations it means all products. This special case is not implemented
4. QuantityDiscount makes the immutable part impossible to fully enforce, as it will always be possible to add extra quantity discounts to an existing pricing plan, but this should not be done.

## Implementation

The above design is implemented in Postgresql by simply mapping the relations and relationships to tables. The complex attributes are flattened and included in the table (see e.g. customer address attribute) and multi-valued attributes are implemented using a separate table (see e.g. customerphone). Weak entity types can either be included in the other entity or the primary key from the other entity can be added to the weak entity. We have decided to keep the entities as separate tables add the primary key from the real entity to the weak entity. An inner join may then be used to generate the combined table.

In the definition there is one table with a special functionality and that is customerpricingplan. This table should ideally have three attributes; productid,customerid,pricingplanid with the superkey being the primary key, as the table may contain multiple customers, and each customer has the same or different pricing plan and the pricing plan may be valid for some or all products, where all is indicated by productid=NULL. This last statement means that productid cannot be part of the primary key, but at the same time customerid,pricingplanid is not a candidate key, so it is not possible. We can create special unique indexes for the situation where productid NULL and where it IS NOT NULL (as may be seen in the file), but this does not give us a key, and therefore a simple SERIAL primary key is created, yet it is never referenced. Please refer to the msdd document for a discussion of why multiple UNIQUE indexes are required.

We have not had the need for checking uniqueness on multiple columns, mainly because the primary key constraint on multiple attributes also enforce the UNIQUE check, and we have extensively used combined primary keys, and have not had any non-key dependencies between attributes.

We strive to achieve Boyce-Codd normal form in the implementation, and please refer to the functional dependencies below for details. TODO add why we want this

Please refer to the file miniproject\_mdd.sql for a complete list of the tables, functions, triggers, roles, indexes and users as well as an example of a complete customer order view. This file also contains comments about interesting aspects of the implementation.

### Physical design

A database naturally must be stored somewhere. Based on the type of application it is recommended to use a hosting service with a high bandwidth access. Whether an IaaS, PaaS or SaaS is used is irrelevant, but all must guarantee a minimum of a RAID setup with a remote nightly backup, or better. Most high end hosting services uses virtual servers with a continuous mirror which ensures that if the service fails, then the mirror can take over seamlessly. Naturally the HDD access is also very important as the database will be too big to hold in memory. Many hosting services can offer a very fast disk on a dedicated SAN.

## Functional dependencies and normal form

As the design allows for the user to add arbitrary attributes and values, it is possibly to argue that the user may then added non-atomic values in a single row, thereby violating the 1st normal form. However, as this would be considered a misuse of the design, and since technically it is still the same number of columns, it is not considered a violation.

* products
  + pid -> name,instock,weight
* productattributes
  + name-> productattributetype
* productattributerelations
  + product,attributename -> value
  + product -> products.pid
  + attributename -> productattributes.name
* pricingplans
  + id -> price, discount, deliveryconditions
* manufacturer
  + vatno -> name, paymentcurrency
* manufactorerorderconfirmations
  + manufactorerid,cocno -> cocDate
  + manufactorerid -> manufacturer.vatno
* manufactorerinvoices
  + manufactorerid, invoiceno -> invoicedate , paybefore , paid
  + manufactorerid -> manufacturer.vatno
* manufactorerdeliveries
  + manufactorerid,freightno -> deliverydate
  + manufactorerid -> manufacturer.vatno
* manufactorerorders
  + orderid -> manufactorerid, orderdate, cocid, invoiceid, freightno
  + manufactorerid -> manufacturer.vatno
  + cocid -> manufactorerorderconfirmations.cocno
  + invoiceid -> manufactorerinvoices.invoiceno
  + freightno -> manufactorerdeliveries.freightno
* manufactorerorderedproducts
  + orderid,productid -> priceingplanid, count
  + orderid -> manufactorerorders.orderid
  + productid -> products.pid
  + priceingplanid -> pricingplans.id
* manufactorerproducts
  + manufactorerid, productid -> priceingplanid
  + manufactorerid -> manufacturer.vatno
  + productid -> products.pid
  + priceingplanid -> pricingplans.id
* webshops
  + id -> vatno, name, paymentcurrency, invoiceaddress, paymentconditions
* customers
  + id -> webshopid, firstname, middlename, sirname, tvmfth, floor, streetletter, streetnumber, streetname, postalcode, region, country, paymentconditions
  + webshopid -> webshops.id
* customerphones
  + cid,phone -> value
  + cid -> customers.id
* customerattributes
  + customerid,name -> value
  + customerid -> customers.id
* customerpricingplan
  + id -> customerid,pricingplanid,productid
  + Candidate key: customerid,pricingplanid,productid
  + customerid -> customers.id
  + priceingplanid -> pricingplans.id
  + productid -> products.pid
* netspayments
  + netsid -> transactionhash
* customerorderconfirmations
  + cocno -> cocdate
* customerinvoices
  + invoiceno -> invoicedate, paybefore, paid
* customerdeliveries
  + deliveryid -> deliverydate, freightno
* customerorders
  + orderid -> customerid, orderdate, netsid, cocid, invoiceid, deliveryid
  + customerid -> customers.id
  + netsid -> netspayments.netsid
  + cocid -> customerorderconfirmations.cocno
  + invoiceid -> customerinvoices.invoiceno
  + deliveryid -> customerdeliveries.deliveryid
* quantitydiscounts
  + pricingplanid, count -> discount
  + priceingplanid -> pricingplans.id
* webshopcarries
  + webshopid, productid -> wpricingplanid, ppricingplanid
  + webshopid -> webshops.id
  + productid -> products.pid
  + wpriceingplanid -> pricingplans.id
  + ppriceingplanid -> pricingplans.id
* customerorderproducts
  + orderid, productid -> priceingplanid, count
  + orderid -> customerorders.orderid
  + productid -> products.pid
  + priceingplanid -> pricingplans.id

We have not included the ENUM types in the functional dependencies as they are always leaf nodes and cannot violate any constraints.

As it may be seen ...

2nd normal form deals with normalization. A non-key field should never be a fact about a subset of a key, i.e. no redundant data. There are many good reasons for this, as redundant data leads to possible inconsistencies, it wastes space and it leads to excessive updates.

Our design does contain redundant data (see instock in products), but as described this is a performance tweak and it does not reference any part of any key, and therefore do not violate 2nd normal form.

3rd normal form states that no non-key field may be a fact about another non-key field (in the same table). As mentioned above the instock attribute in products relates to the sum of all purchased and sold products, but it is in multiple tables. A place where this problem could have existed is in the complex address attribute of customers, where the postal code could have been followed up by a town attribute, which would relate these two non-key fields. This is however not the case, as the town is found through an online service outside the DB if the town is required, and therefore there is no need for a town attribute.

Boyce-Codd normal form is very like the 3rd normal form and it is only in a very special situation that it is possible to achieve 3rd normal form, but not Boyce-Codd normal form, and that is if there is a reference from a ...

# Requirement traceability

* + 1. Wholesaler access is controlled by the wholesaler
    2. The Wholesaler has access to the instock attribute in products which is automatically updated.
    3. All sales and purchasing history is kept and may be extracted.
    4. Single warehouse is what has been implemented
    5. That is what the manufacturer does
    6. A PricingPlan is assigned to each combination of manufacturer and product specifying the agreed on price.
    7. Along with the price may be assigned discounts possibly dependent on amounts.
    8. The order, order confirmation, order invoice (including payment) and order delivery may be tracked in the DB.
    9. And it also is not.
    10. Yes, there is a many to many relationship between manufacturer and product.
    11. Yes, this can be done by assigning special attributes to the products linking them together and then assigning them to the same PricingPlan to have them cost the same.
    12. Yes, this is also possible through the assigning of product attributes.
    13. Yes, the webshop role has access to this.
    14. Yes, this is done through the netspayments relation
    15. The order, order confirmation, order invoice (including payment) and order delivery may be tracked in the DB.
    16. There is a webshopcarries relation specifying which products are carried by the web-shop.
    17. The webshopcarries also specify the price that the given web-shop buys the products from the wholesaler for.
    18. Along with the price may be assigned discounts possibly dependent on amounts
    19. This is not fully implemented. Limiting what the web-shops may see through Views can take us some of the way, and limiting where the web-shop may insert, update or delete also help, but without RLS or a layer outside the DB, this is not possible.
    20. Also requires RLS or a layer outside the DB
    21. Also requires RLS or a layer outside the DB
    22. And it is in customerattributes
    23. Yes, the WebShopRole has this right and the customers relation contains this information.
    24. The webshopcarries allows the web-shop to specify the selling price of a given product.
    25. The webshopcarries PricingPlan also allow for discounts possibly dependent on amounts, and a special Customer PricingPlan may offer further discounts, both general and at the product level (Customer PricingPlan not fully implemented).
    26. The netspayments relation allows for this
    27. The customerinvoices allows for this
    28. This has not been implemented, and is left for future implementation, along with handling discrepancies in the periodic manual warehouse count.
    29. This has not been implemented.
    30. Yes the CustomerRole has this right and all sales are logged in the DB.
    31. Please see below under performance and indexes, but this requirement cannot always be met.

# Queries and views

TODO Add EXPLAIN ANALYZE to get exact execution time and build example execution tree.

## Best selling product

|  |
| --- |
| SELECT productid, SUM(count) FROM customerorderproducts GROUP BY productid order by SUM(count) DESC limit 1; |

### Example result

|  |  |
| --- | --- |
| **productid** | **sum** |
| 46195 | 503 |

## Best selling product with name

|  |
| --- |
| SELECT name, productid, count FROM (SELECT productid, SUM(count) AS count FROM customerorderproducts GROUP BY productid ORDER BY SUM(count) DESC limit 1) temp INNER JOIN products ON temp.productid=products.pid; |

### Example result

|  |  |  |
| --- | --- | --- |
| **name** | **productid** | **count** |
| Product 45195 | 46195 | 503 |

This code also makes it easy to find the 10 best selling products (just change limit) or worst selling products (just change ORDER BY to ASC).

## Most purchasing customer (money, no quantity discount)

|  |
| --- |
| SELECT customerid, SUM(price - price\*discount/100) FROM customerorderproducts INNER JOIN customerorders ON customerorderproducts.orderid= customerorders.orderid INNER JOIN pricingplans ON customerorderproducts.priceingplanid=pricingplans.id GROUP BY customerorders.customerid ORDER BY sum(price - price\*discount/100) DESC LIMIT 1; |

### Example result

|  |  |
| --- | --- |
| **customerid** | **SUM(price - price\*discount/100)** |
| 13695 | 62057.20 |

## Biggest Wholesaler profit margin product (one unit purchased)

|  |
| --- |
|  |

## Biggest Web-shop profit margin product (no individual discount)

|  |
| --- |
| WITH tempdata AS (SELECT webshopid, productid, (((p1.price - p1.price\*p1.discount/100) - (p2.price - p2.price\*p2.discount/100))/(p2.price - p2.price\*p2.discount/100))\*100 AS profit FROM webshopcarries INNER JOIN pricingplans p1 ON p1.id=webshopcarries.wpricingplanid INNER JOIN pricingplans p2 ON p2.id=webshopcarries.ppricingplanid)  SELECT tempdata.webshopid, tempdata.productid, to\_char(tempdata.profit,'S999999D99') || '%' AS profitmargin FROM (SELECT webshopid, MAX(profit) AS maxprofit FROM tempdata GROUP BY webshopid) groupdata INNER JOIN tempdata ON groupdata.webshopid=tempdata.webshopid AND groupdata.maxprofit=tempdata.profit LIMIT 10; |

Only 10 first found is show to reduce result. Remove Limit 10 to see all

### Example result

|  |  |  |
| --- | --- | --- |
| **webshopid** | **productid** | **profitmargin** |
| 1000 | 83185 | +1094.35% |
| 1008 | 96850 | +1127.39% |
| 1009 | 51799 | +1096.54% |
| 1010 | 5879 | +1096.55% |
| 1011 | 61877 | +1106.51% |
| 1016 | 36092 | +956.32% |
| 1017 | 59553 | +1081.33% |
| 1021 | 26243 | +1182.40% |
| 1022 | 11165 | +795.63% |
| 1023 | 11483 | +1196.48% |

First suggestion was based on a temporary table, which was populated with the INTO keyword, but then we were made aware of the WITH statement, and it gives a nicer query.

## Products not sold for over 1 month

|  |
| --- |
|  |

## Wholesaler profit

|  |
| --- |
|  |

## Invoice view

The invoice does not have an amount, so to view an invoice we need to calculate it from the order information.

|  |
| --- |
| CREATE VIEW customerinvoicewithamount AS  SELECT invoiceno, invoicedate, paybefore, paid, op.price as amount FROM (SELECT orderid, SUM(price - price\*discount/100) price FROM customerorderproducts INNER JOIN pricingplans ON pricingplans.id=customerorderproducts.priceingplanid GROUP BY orderid) op INNER JOIN customerorders ON op.orderid=customerorders.orderid INNER JOIN customerinvoices ON customerorders.invoiceid=customerinvoices.invoiceno; |

This can then be used as

|  |
| --- |
| SELECT invoicedate, paybefore, paid, amount FROM customerinvoicewithamount WHERE invoiceno=145; |

A simple addition of more JOINs to add the tables for order confirmation and delivery would allow the retrieval of all order information in one select, and as this is often done when presenting a customer with their previous purchases this is a useful view. This view may be found in the MDD file.

# Performance and indexes

The PostgreSQL automatically generates indexes for the primary keys when the tables are created, and this makes a lot of sense because of all the joining with the primary keys. When manually creating indexes it is important to consider which queries must execute quickly. Naturally the calculation of profit margins and validation of instock values are not required to run fast, as this is a maintenance query most likely run automatically at off-peak hours.

The customer address is a good example. This contains a postal code, region and country as well as paymentconditions, which are all suitable for indexing. If the web-shops wants to continuously display the purchases of a given postal code this must run often and fast.

To test this we first run a simple query using the postal code:

|  |
| --- |
| EXPLAIN ANALYZE SELECT \* FROM customers WHERE postalcode=’1010’; |

Total runtime: 6.38ms

If we then create an index of the postal codes:

|  |
| --- |
| CREATE INDEX customer\_postalcode\_idx ON customers(postalcode); |

Total runtime SELECT \* FROM customers WHERE postalcode=’1010’: 0.125ms

An improvement of more than a factor of 50, which is a very good improvement.

Another very obvious place to create an index is for the productattributes. There is already an index for the primary key, which is the combination of product id and attribute name, but a common lookup will be find all products with a given attribute name and value, which the index does not help with.

|  |
| --- |
| CREATE INDEX productattributerelations\_name\_value\_idx ON productattributerelations (attributename, value); |

EXPLAIN ANALYZE SELECT \* FROM productattributerelations WHERE attributename=’brand’ AND value=’Brand 12’;

This goes from an execution time of 65ms to 3.2ms – an improvement of a factor of 20.

Naturally there are many other candidates, and as the database matures it is also possible to make the required indexes as it becomes apparent which queries are important and which are allowed to be slow.

# Transactions

Transactions are important when multiple updates of the database and where it is important that either all updates or no updates occur. There are two situations where this is very important in the database; when creating a manufacturer order and when creating a customer order.

In both situations when the order is created the order is inserted into the orders table, and the products are inserted into the orderedproducts table. This must be done as an atomic action, and this is done via the use of transactions. This may also be seen in the file miniproject\_create1.sql.

Transactions can have different levels of protection, from pure ACID, where the transaction blocks all affected tables until the transaction is finalized. An alternative is the slightly more lenient read uncommitted, where others may read the transaction data, even though it is not committed, but any attempt to update the same section would not be allowed. This more lenient version is believed sufficient for us.

# Conclusion

# Appendix

## Files

This list contains all files which is a part of this report, along with a short description of what they contain.

* Report miniproject.pdf

*This report*

* miniproject\_mdd.sql

*Contains the database definitions in SQL (compatible with postgresql) including views, users, roles and triggers, etc. The file also contain SQL comments for anything noteworthy.*

* miniproject\_clear.sql

*Contains DROP statements for the database, users and roles in miniproject\_mdd.sql. This is used if one wishes start fresh, or go from the single entry instance to the large database.*

* miniproject\_create1.sql

*Contains SQL statements for populating the database with a single instance of product, web-shop, manufacturer, customer, and orders. This uses the realistic approach for the order procedure. The file also contain SQL comments for anything noteworthy.*

* execute\_db.bat

*Batch utility file to assist in the execution of .sql files in postgresql.*

* execute\_db\_silent.bat

*Batch utility file to assist in the execution of very large .sql files in postgresql.*

* DatabaseContentGenerator.exe

*Executable for generating a file which will populate a database created with miniproject\_mdd.sql*

*with a bunch of test data. See section tool below for further details.*

## Large database installation manual

1. Download the needed files:
   1. /DatabaseContentGenerator/executable/DatabaseContentGenerator.exe
   2. miniproject\_mdd.sql
   3. execute\_db.bat
   4. execute\_db\_silent.bat
2. Execute DatabaseContentGenerator.exe and choose a starting point (e.g. 1000), press Generate and select a location for the result file (e.g. webshoptest1.sql)
   1. The execution may take several minutes (though less than 5 on our computer) and will generate a file with between 100 and 200MB of insert data.
   2. If an error is generated please try again (will happen in less than 10% of the executions).
3. Ensure that the psql.exe is in the Environment search path
4. Open a command prompt in the directory containing execute\_db.bat and execute\_db\_silent.bat
5. If a database called “webshoptest1” already exist please open the PSQL prompt and DROP it.
6. Execute “execute\_db.bat <path to miniproject\_mdd.sql>” where <path to miniproject\_mdd.sql> is replaced with the path to the actual file.
   1. Enter PSQL password when prompted
7. Verify that the execution was successful.
8. Execute “execute\_db\_silent.bat <path to webshoptest1.sql>” where <path to webshoptest1.sql> is replaced with the path to the file generated by DatabaseContentGenerator.exe.
   1. Enter PSQL password when prompted
   2. This may take several minutes (though less than 5 on our computer).
   3. No output will be generated to the console, but will be output to a file (dumpfile.txt) in the same directory that the execute\_db\_silent.bat was executed from.
   4. After execution verify that a file called dumpfile.txt exists, and that it is slightly bigger than the webshoptest1.sql (about 1kb bigger, so hardly anything). If you have a large text file viewer you are welcome to verify that file content.
9. Open a PSQL command prompt and execute “\c webshoptest1” to change the database.
10. Copy the SQL queries from the report into the PSQL command prompt to verify the results above
11. Invent you own fun queries to play around with the data, and please remember that the sales and purchase prices are generated at random, so some of the products are not good business.

## Tool

The tool is a windows .NET application based on version 3.5, and will generate a text file with insert statements containing the following:

* 100000 products
* 5000 productlines + 1 no-line
* 20 brands + 1 no-brand
* All products has size 0 - 79 (XXS - XXXL)
* All products hsa a colour ("red", "white", "green", "blue", "lightblue", "baige", "black", "multicoloured")
* 10 manufactorers with random currency
* 10000 pricing plans without discount
* 10000 pricing plans with discount
* 2000 of the pricing plans has quantity discounts.
* Each product is produced by 1 or 2 manufactorers with random pricing plans.
* Each product has been belivered between 1 and 15 times and in each order is between 1 and 10 products
* There are 60 webshops
* Each webshop carries between 1000 and 10000 products
* Each webshop has between 50 and 1000 customers
* Each customer has purchased between 0 and 100 products.
* Each order from the customer consists of between 1 and 10 products.
* Each product is purchased in a quantity of 1 to 3 units

The file is approximately 130MB, and may naturally be moved to a non-windows PC via e.g. an USB stick.

The file generates specific insert statements, and thereby bypasses the two triggers used to maintain the instock attribute of products. This is important for performance reasons, as the creation of the database would otherwise take much longer. An early experiment shows that importing the data takes more than 1 hour, if the same principle as miniproject\_create1.sql is used.