Table of content

[Introduction 2](#_Toc369284397)

[Requirements 2](#_Toc369284398)

[Wholesaler 3](#_Toc369284399)

[Manufacturer 3](#_Toc369284400)

[Web-shop 3](#_Toc369284401)

[Web-shop customer 4](#_Toc369284402)

[Scenarios 4](#_Toc369284403)

[Purchasing from manufactorer 4](#_Toc369284404)

[Customer purchasing product 4](#_Toc369284405)

[Setting up a web-shop 5](#_Toc369284406)

[Design 5](#_Toc369284407)

[Choosing a database type 5](#_Toc369284408)

[Who is going to use it 6](#_Toc369284409)

[How is it going to be used 6](#_Toc369284410)

[Relational design / ER-diagram 7](#_Toc369284411)

[Overall ER-diagram 8](#_Toc369284412)

[Customer relationship diagram 10](#_Toc369284413)

[Manufacturer relationship diagram 12](#_Toc369284414)

[Customer attribute diagram 13](#_Toc369284415)

[Product attribute diagram 15](#_Toc369284416)

[Web-shop attribute diagram 15](#_Toc369284417)

[Manufacturer attribute diagram 16](#_Toc369284418)

[Pricing plan attribute diagram 18](#_Toc369284419)

[Transactional dependencies 18](#_Toc369284420)

[Queries and views 18](#_Toc369284421)

[Best selling product 18](#_Toc369284422)

[Example result 18](#_Toc369284423)

[Best selling product with name 19](#_Toc369284424)

[Example result 19](#_Toc369284425)

[Most purchasing customer (money, no quantity discount) 19](#_Toc369284426)

[Attempt one 19](#_Toc369284427)

[Attempt two 19](#_Toc369284428)

[Example result 19](#_Toc369284429)

[Biggest Wholesaler profit margin product (one unit purchased) 19](#_Toc369284430)

[Biggest Web-shop profit margin product (no individual discount) 20](#_Toc369284431)

[Example result 20](#_Toc369284432)

[Products not sold for over 1 month 20](#_Toc369284433)

[Wholesaler profit 20](#_Toc369284434)

[Performance and indexes 20](#_Toc369284435)

[Transactions 21](#_Toc369284436)

[Appendix 21](#_Toc369284437)

[Installation manual 21](#_Toc369284438)

# 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, delivery time, 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 product the same product.
7. It must be possible to have a product which exist 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).
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/...).
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.

### Setting up a web-shop

TBD

# 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
* Manufacturer relationship diagram
* Web-shop relationship diagram
* Customer relationship diagram
* Product attribute diagram
* Manufacturer attribute diagram
* Web-shop attribute diagram
* Customer attribute diagram

### Overall ER-diagram



In the above may be found the following entities

* 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.

* MPricingPlan

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.

* WPricingPlan

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.

* PPricingPlan

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.

* CPricingPlan

A Customer may be negotiated special conditions, discounts or prices with the Web-shop, and this is expressed in the CPricingPlan. The CPricingPlan supersedes the PPricingPlan, unless the PPricingPlan is better (maybe due to a special offer).

#### 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.
4. Customer pricing plan is presently not implemented

### Customer relationship diagram



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

* 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 CPricingPlan or PPricingPlan, 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.

#### 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



### Customer attribute diagram













### Product attribute diagram



### Web-shop attribute diagram



#### Notes

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

### Manufacturer attribute diagram











#### Note

1. 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.

### 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.

## Transactional dependencies

# Queries and views

TODO Add EXPLAIN ANALYZE to get exact execution time and build 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)

### Attempt one

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

This code ran for > 11 hours without completing, and was then terminated. We then noticed an interesting error. The “customerorderproducts INNER JOIN customerorders ON customerorderproducts.orderid=customerorderproducts.orderid” is incorrect, it should be “customerorderproducts INNER JOIN customerorders ON customerorderproducts.orderid= customerorders.orderid”. It could be interesting to consider why this never completes.

### Attempt two

|  |
| --- |
| 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; |

Executes in a matter of seconds

### 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)

|  |
| --- |
| -- Get the values we need to work on into a temp table  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 INTO TEMPORARY tempdata FROM webshopcarries INNER JOIN pricingplans p1 ON p1.id=webshopcarries.wpricingplanid INNER JOIN pricingplans p2 ON p2.id=webshopcarries.ppricingplanid;  -- Get the groupd max values and join with self  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;  -- Drop the temp table  DROP TABLE tempdata; |

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% |

## 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; |

# Performance and indexes

# Transactions

# Appendix

## 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.