**Table of content**

[1 Introduction 2](#_Toc370196502)

[2 Requirements 3](#_Toc370196503)

[2.1 Wholesaler 3](#_Toc370196504)

[2.2 Manufacturer 3](#_Toc370196505)

[2.3 Web-shop 3](#_Toc370196506)

[2.4 Web-shop customer 4](#_Toc370196507)

[2.5 Scenarios 4](#_Toc370196508)

[2.5.1 Purchasing from manufactorer 4](#_Toc370196509)

[2.5.2 Customer purchasing product 4](#_Toc370196510)

[3 Design 5](#_Toc370196511)

[3.1 Choosing a database type 5](#_Toc370196512)

[3.2 Who is going to use it 6](#_Toc370196513)

[3.3 How is it going to be used 6](#_Toc370196514)

[3.4 Relational design / ER-diagram 7](#_Toc370196515)

[3.4.1 Overall ER-diagram 8](#_Toc370196516)

[3.4.2 Customer relationship diagram 10](#_Toc370196517)

[3.4.3 Manufacturer relationship diagram 12](#_Toc370196518)

[3.4.4 Product attribute diagram 14](#_Toc370196519)

[3.4.5 Web-shop attribute diagram 14](#_Toc370196520)

[3.4.6 Pricing plan attribute diagram 15](#_Toc370196521)

[3.5 Implementation 15](#_Toc370196522)

[3.5.1 Physical design 21](#_Toc370196523)

[3.6 Functional dependencies and normal form 22](#_Toc370196524)

[4 Requirement traceability 23](#_Toc370196525)

[5 Queries and views 24](#_Toc370196526)

[5.1 Single purchase and sale 24](#_Toc370196527)

[5.2 Best selling product 26](#_Toc370196528)

[5.2.1 Example result 26](#_Toc370196529)

[5.3 Best selling product with name 26](#_Toc370196530)

[5.3.1 Example result 26](#_Toc370196531)

[5.4 Most purchasing customer (money, no quantity discount) 26](#_Toc370196532)

[5.4.1 Example result 26](#_Toc370196533)

[5.5 Biggest Wholesaler profit margin product (one unit purchased) 26](#_Toc370196534)

[5.6 Biggest Web-shop profit margin product (no individual discount) 26](#_Toc370196535)

[5.6.1 Example result 27](#_Toc370196536)

[5.7 Products not sold for over 1 month 27](#_Toc370196537)

[5.8 Wholesaler profit 27](#_Toc370196538)

[5.9 Invoice view 27](#_Toc370196539)

[6 Performance and indexes 27](#_Toc370196540)

[7 Transactions 29](#_Toc370196541)

[8 Conclusion 29](#_Toc370196542)

[9 Appendix 30](#_Toc370196543)

[9.1 Files 30](#_Toc370196544)

[9.2 Large database installation manual 30](#_Toc370196545)

[9.3 Tool 31](#_Toc370196546)

# 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



### 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 of the owning entity (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.

Below is a selection of CREATE statements defining the database schema, for a complete list please refer to the file miniproject\_mdd.sql. The shown CREATE statements are the ones we feel is the most important. The notation ... will be used to indicate that some CREATE statements has been removed.

|  |
| --- |
| CREATE DATABASE webshoptest1;  \c webshoptest1  CREATE TYPE productattributetype AS ENUM ('string', 'integer', 'float');  -- weight is in grams so integer is fine.  CREATE TABLE products (  pid SERIAL PRIMARY KEY,  name VARCHAR(128) NOT NULL,  instock INTEGER NOT NULL DEFAULT 0,  weight INTEGER NOT NULL  );  CREATE TABLE productattributes (  name VARCHAR(128) PRIMARY KEY,  type productattributetype NOT NULL  );  CREATE TABLE productattributerelations (  product INTEGER,  attributename VARCHAR(128),  value VARCHAR(128) NOT NULL,  PRIMARY KEY (product, attributename),  FOREIGN KEY (product) REFERENCES products(pid),  FOREIGN KEY (attributename) REFERENCES productattributes(name)  );  CREATE TYPE termsofdelivery AS ENUM ('abLager', 'SideOfShip', '3month', '14dg', '7dg', '1dg');  -- if the pricing plans are to be shared it is important that they are immutable (cannot be gauranteed by the DB)  -- serial is a pseudonum for integer/big so it is possible to use an integer to refecerence it in a foreign key.  -- should the pricing plan include a currency??? - or do we default to DKK?  CREATE TABLE pricingplans (  id SERIAL PRIMARY KEY,  price FLOAT NOT NULL,  discount FLOAT NULL,  deliveryconditions termsofdelivery NULL  );  CREATE TYPE currency AS ENUM ('DKK', 'EUR', 'USD', 'GBP', 'AUD', 'INR', 'AED', 'CAD', 'CHF', 'CNY');  CREATE TABLE manufactorer (  vatno VARCHAR(128) PRIMARY KEY,  name VARCHAR(128) NOT NULL,  paymentcurrency currency NOT NULL  );  ...  CREATE TABLE manufactorerinvoices (  manufactorerid VARCHAR(128),  invoiceno VARCHAR(128) NOT NULL,  invoicedate DATE NOT NULL,  paybefore DATE NOT NULL,  paid BOOLEAN NOT NULL DEFAULT false,  PRIMARY KEY(manufactorerid,invoiceno),  FOREIGN KEY (manufactorerid) REFERENCES manufactorer(vatno)  );  ...  CREATE TABLE manufactorerorders (  orderid SERIAL PRIMARY KEY,  manufactorerid VARCHAR(128),  orderdate DATE NOT NULL,  cocid VARCHAR(128) NULL,  invoiceid VARCHAR(128) NULL,  freightno VARCHAR(128) NULL,  FOREIGN KEY(manufactorerid) REFERENCES manufactorer(vatno),  FOREIGN KEY(manufactorerid,cocid) REFERENCES manufactorerorderconfirmations(manufactorerid,cocno),  FOREIGN KEY(manufactorerid,invoiceid) REFERENCES manufactorerinvoices(manufactorerid,invoiceno),  FOREIGN KEY(manufactorerid,freightno) REFERENCES manufactorerdeliveries(manufactorerid,freightno)  );  -- Orriginally we had the manufactorercoc, invoice and deliveires reference the order, having order be the single primary key, but this was not possible, As we could not ensure that the cocno and invoiceno would be unique for the manufactorer. Create unique index as no two orders from the same manufactorer may ever have the same cocno - not possible.  -- how do we werify that any product in there for a given manufactorer is also in manufactorerproducts? If manufcatorerid and productid was here it would be possible, but it is not???  CREATE TABLE manufactorerorderedproducts (  orderid INTEGER,  productid INTEGER,  priceingplanid INTEGER NOT NULL,  count INTEGER NOT NULL CHECK (count > 0),  PRIMARY KEY (orderid,productid),  FOREIGN KEY (orderid) REFERENCES manufactorerorders(orderid),  FOREIGN KEY (productid) REFERENCES products(pid),  FOREIGN KEY (priceingplanid) REFERENCES pricingplans(id)  );  CREATE TABLE manufactorerproducts (  manufactorerid VARCHAR(128),  productid INTEGER,  priceingplanid INTEGER NOT NULL,  PRIMARY KEY (manufactorerid,productid),  FOREIGN KEY (manufactorerid) REFERENCES manufactorer(vatno),  FOREIGN KEY (productid) REFERENCES products(pid),  FOREIGN KEY (priceingplanid) REFERENCES pricingplans(id)  );  CREATE TYPE termsofpayment AS ENUM ('prepay', '10dgNet', '14dgNet', '30dgNet', 'LbMntPlus15dg');  CREATE TABLE webshops (  id SERIAL PRIMARY KEY,  vatno VARCHAR(64) NOT NULL,  name VARCHAR(128) NOT NULL,  paymentcurrency currency NOT NULL,  invoiceaddress VARCHAR(256) NOT NULL,  paymentconditions termsofpayment NOT NULL DEFAULT '30dgNet'  );  CREATE TYPE apartmentlocations AS ENUM ('left', 'middle', 'right');  CREATE TYPE countries AS ENUM ('Denmark', 'England', 'USA');  -- We do not store the city as it may be derived form the postal code and there are lots of online services for that, and that way we do not risk an inconsistency.  -- The unique below will not catch all the combinations where some of the values are NULL (see customerpricingplan), but due to multiple NULLable attributes creating a full UNIQUE coverage is very complex, and has been postponed (requires a WHERE for all combinations of possible NULL attributes.  CREATE TABLE customers (  id SERIAL PRIMARY KEY,  webshopid INTEGER NOT NULL,  firstname VARCHAR(128) NOT NULL,  middlename VARCHAR(128) NULL,  sirname VARCHAR(128) NOT NULL,  tvmfth apartmentlocations NULL,  floor INTEGER NULL,  streetletter CHAR(2) NULL,  streetnumber INTEGER NOT NULL,  streetname VARCHAR(128) NOT NULL,  postalcode VARCHAR(32) NOT NULL,  region VARCHAR(64) NULL,  country countries NOT NULL,  paymentconditions termsofpayment NOT NULL DEFAULT 'prepay',  UNIQUE (webshopid, firstname, middlename, surname, tvmfth, floor, streetletter, streetnumber, streetname, postalcode, region, country),  FOREIGN KEY(webshopid) REFERENCES webshops(id)  );  ...  CREATE TABLE customerattributes (  customerid INTEGER,  name VARCHAR(128),  value VARCHAR(128) NOT NULL,  PRIMARY KEY (customerid,name),  FOREIGN KEY(customerid) REFERENCES customers(id)  );  -- Special customer-specific discount. The functionality is not used at present, and it wil remain empty.  -- This uses a unused primary key and custom unique index trick. This is because the unique-ness of the  -- columns: customerid,pricingplanid,productid should form the primary key, but as productid may be null  -- it cannot be part of the primary key, but it can be part of a unique check. However in postgresql  -- the following is allowed: CREATE TABLE test (t INTEGER NULL, UNIQUE(t)); INSERT INTO test (NULL); INSERT INTO test (NULL);  -- This is because NULL is not nothing, and NULL != NULL (the latter may seem a little counter-intuitive, but it is because NULL  -- is neither nothing nor something).  -- To bypass this we create two seperate unique indexes depending on whether the productid is NULL or not, and then it works.  CREATE TABLE customerpricingplan (  id SERIAL PRIMARY KEY,  customerid INTEGER NOT NULL,  pricingplanid INTEGER NOT NULL,  productid INTEGER NULL,  FOREIGN KEY(customerid) REFERENCES customers(id),  FOREIGN KEY (productid) REFERENCES products(pid)  );  CREATE UNIQUE INDEX customerpricingplan\_3col\_uni\_idx  ON customerpricingplan (customerid, pricingplanid, productid)  WHERE productid IS NOT NULL;  CREATE UNIQUE INDEX customerpricingplan\_2col\_uni\_idx  ON customerpricingplan (customerid, pricingplanid)  WHERE productid IS NULL;  -- partly for fun and partly because we control the IDs we reverse the dependency and make the ids unique.  ...  -- a null in paybefore is used for creditcard payments (immediate)  CREATE TABLE customerinvoices (  invoiceno SERIAL PRIMARY KEY,  invoicedate DATE NOT NULL,  paybefore DATE NULL,  paid BOOLEAN NOT NULL DEFAULT false  );  ...  -- the orderid can be left unique only in conjunction with the customer, but that makes the relationship to purchased products more complicated.  CREATE TABLE customerorders (  orderid SERIAL PRIMARY KEY,  customerid INTEGER NOT NULL,  orderdate DATE NOT NULL,  netsid BIGINT NULL,  cocid INTEGER NULL,  invoiceid INTEGER NULL,  deliveryid INTEGER NULL,  FOREIGN KEY(customerid) REFERENCES customers(id),  FOREIGN KEY(netsid) REFERENCES netspayments(netsid),  FOREIGN KEY(cocid) REFERENCES customerorderconfirmations(cocno),  FOREIGN KEY(invoiceid) REFERENCES customerinvoices(invoiceno),  FOREIGN KEY(deliveryid) REFERENCES customerdeliveries(deliveryid)  );  -- be very very careful. Inserting into this relation will change an existing pricing plan - can it be prevented?  -- At first we have a quantitydiscountrelationships and quantitydiscounts, but we realised that count would never be the same for the same pricing plan so we could save a relation.  -- discount is a percentage  CREATE TABLE quantitydiscounts(  pricingplanid INTEGER,  count INTEGER NOT NULL CHECK (count > 0),  discount FLOAT NOT NULL CHECK (discount > 0 AND discount < 1),  PRIMARY KEY (pricingplanid, count),  FOREIGN KEY (pricingplanid) REFERENCES pricingplans(id)  );  CREATE TABLE webshopcarries (  webshopid INTEGER NOT NULL,  productid INTEGER NOT NULL,  wpricingplanid INTEGER NOT NULL,  ppricingplanid INTEGER NOT NULL,  PRIMARY KEY (webshopid,productid),  FOREIGN KEY (webshopid) REFERENCES webshops(id),  FOREIGN KEY (productid) REFERENCES products(pid),  FOREIGN KEY (wpricingplanid) REFERENCES pricingplans(id),  FOREIGN KEY (ppricingplanid) REFERENCES pricingplans(id)  );  CREATE TABLE customerorderproducts (  orderid INTEGER,  productid INTEGER,  priceingplanid INTEGER NOT NULL,  count INTEGER NOT NULL CHECK (count > 0),  PRIMARY KEY (orderid,productid),  FOREIGN KEY (orderid) REFERENCES customerorders(orderid),  FOREIGN KEY (productid) REFERENCES products(pid),  FOREIGN KEY (priceingplanid) REFERENCES pricingplans(id)  );  CREATE FUNCTION trigfunc\_manufactorer\_order() RETURNS trigger AS $$  BEGIN  UPDATE products SET instock=p.instock+mop.count FROM manufactorerorders mo INNER JOIN manufactorerorderedproducts mop ON mo.orderid=mop.orderid INNER JOIN products p ON mop.productid=p.pid WHERE mo.orderid=NEW.orderid;  return NEW;  END  $$ LANGUAGE plpgsql;  -- This trigger is only for debugging as we would like to insert an order directly (not via update)  CREATE TRIGGER update\_in\_stock\_manufactorer  AFTER UPDATE ON manufactorerorders  FOR EACH ROW  WHEN (NEW.freightno IS NOT NULL AND OLD.freightno IS NULL)  EXECUTE PROCEDURE trigfunc\_manufactorer\_order();  CREATE FUNCTION trigfunc\_customer\_order() RETURNS trigger AS $$  BEGIN  UPDATE products SET instock=p.instock-cop.count FROM customerorders co INNER JOIN customerorderproducts cop ON co.orderid=cop.orderid INNER JOIN products p ON cop.productid=p.pid WHERE co.orderid=NEW.orderid;  return NEW;  END  $$ LANGUAGE plpgsql;  -- This trigger is only for debugging as we would like to insert an order directly (not via update)  CREATE TRIGGER update\_in\_stock\_customer  AFTER UPDATE ON customerorders  FOR EACH ROW  WHEN (NEW.deliveryid IS NOT NULL AND OLD.deliveryid IS NULL)  EXECUTE PROCEDURE trigfunc\_customer\_order();  -- View detailing everything about the customer orders  -- Left join is used as some tables may not contain any entry yet (invoice, delivery) or may never have a matching row (netspayments).  CREATE VIEW customerorderview AS  SELECT customerid, customerorders.orderid AS orderid, orderdate, cocno, cocdate, invoiceno, invoicedate, paybefore, paid, op.price as amount, customerorders.deliveryid AS deliveryid, deliverydate, freightno, customerorders.netsid AS netsid 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 LEFT JOIN customerinvoices ON customerorders.invoiceid=customerinvoices.invoiceno LEFT JOIN customerorderconfirmations ON customerorders.cocid=customerorderconfirmations.cocno LEFT JOIN customerdeliveries ON customerorders.deliveryid=customerdeliveries.deliveryid LEFT JOIN netspayments ON customerorders.netsid=netspayments.netsid;  CREATE VIEW customerorderproductview AS  SELECT customerid, customerorderview.orderid AS orderid, orderdate, cocno, cocdate, invoiceno, invoicedate, paybefore, paid, amount, deliveryid, deliverydate, freightno, netsid, productid, count, price, discount FROM customerorderview INNER JOIN customerorderproducts ON customerorderview.orderid=customerorderproducts.orderid INNER JOIN pricingplans ON pricingplans.id=customerorderproducts.priceingplanid;    CREATE ROLE WebshopRole;  CREATE ROLE CustomerRole;  CREATE ROLE ManufactorerRole;  CREATE ROLE WholesalerRole;  -- The webshops must be able to see its own data, but not change it.  GRANT SELECT ON webshops TO GROUP WebshopRole;  -- The webshops must be able to see which products it carries, and update one of the pricing plans  GRANT SELECT,UPDATE ON webshopcarries TO GROUP WebshopRole;  -- The webshops must be able to create, modify and destroy new customers  ...  -- The Customer must be able to see its own data but not change it  GRANT SELECT ON customers TO GROUP CustomerRole;  GRANT SELECT ON customerphones TO GROUP CustomerRole;  ...  -- The Manufactorer must be able to see its own data and  GRANT SELECT ON manufactorer TO GROUP ManufactorerRole;  GRANT SELECT ON manufactorerproducts TO GROUP ManufactorerRole;  GRANT SELECT ON manufactorerorders TO GROUP ManufactorerRole;  GRANT SELECT ON manufactorerorderedproducts TO GROUP ManufactorerRole;  GRANT SELECT,INSERT ON manufactorerorderconfirmations TO GROUP ManufactorerRole;  ...  -- The Wholesaler has permission to create and update products  GRANT SELECT,INSERT,UPDATE,DELETE ON products TO GROUP WholesalerRole;  GRANT SELECT,INSERT,UPDATE,DELETE ON productattributes TO GROUP WholesalerRole;  ...  -- A simple way to limit some of what the different people may see, is to use a VIEW. It is then possible to limit what a given user may see to only what the VIEW exposes, and not the actual tables. E.g. the user can only see the pricing plans that relates to webshopcarries, but not the pricing plans which relates only to manufactorerorders  -- Create dummy users  CREATE USER WebShop1 WITH PASSWORD 'ws1' IN ROLE WebshopRole;  CREATE USER WebShop2 WITH PASSWORD 'ws2' IN ROLE WebshopRole;  ... |

There are several important aspects to note here:

1. The manufacturer orders are generated by the manufacturer and there may therefore e.g. be two identical invoice numbers. If the invoice number is to be used as a key then it must be combined with an id identifying the manufacturer that sent it. An invoice id must be unique from the given manufacturer, and it is therefore acceptable to use a combined key with manufacturer id and invoice id, which is what we have done. The same principle is used for order confirmation and delivery. In practice this is done by including both manufacturer id and invoice id as primary key for the manufactorerinvoices table.
2. The Customer invoice (and order confirmation and delivery description) is generated solely by the wholesaler, and it is therefore possible to guarantee a unique invoice id for all web-shops. As this offers a simpler design, we have chosen to make the invoice id primary key, allowing us to not include the web-shop as part of the customerinvoices table. Naturally the web-shop may still be found vie the customer which specifies the web-shop id (customer invoice <- customer order -> customer -> web-shop).
3. The customerpricingplan table is special 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 mdd document for a discussion of why multiple UNIQUE indexes are required. Some notes next to the CREATE statement gives more details about why multiple UNIQUE statements are required.
4. In order to automatically keep the instock attribute of products up to date two triggers are created. One which reduces instock whenever a product is shipped (delivery added to customerorders) and one which increases instock whenever a product is received from a manufacturer (freightno added to manufactorerorders).
5. Multiple roles are created to limit access to the DB depending on who the user is, and two views are show to give an example of how to return all data about an order. The views may also be used to further limit the access, but only giving access to a collection of views, instead of the tables which are used to generate the views. This will e.g. prevent a web-shop from seeing what other pricing plans there are (pricing plans not used by any web-shops). This limitation through views has not been implemented, but is a possibility.
6. The quantity discount and pricing plan tables are very special. As they are shared by many products, web-shops, manufacturers and customers, great care must naturally be taken when modifying one. To prevent one web-shop from changing the price of another web-shop’s prices, the pricing plan entries should be considered immutable, and to change the price of a product a new PricingPlan should be created and the references of the products to change prices for should be updated to this new PricingPlan. This is also true for the quantity discount, but unfortunately here there is not easy way to prevent modifications. Naturally unused pricing plans must routinely be cleaned up.

We strive to achieve Boyce-Codd normal form in the implementation, and please refer to the functional dependencies below for details.

### 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.

A database adhering to the ACID properties must ensure both Consistency and Durability, which relates to the physical design. When the user instructs the database to perform an update, it is important that the user can feel confident that, if the system returns a status that the update has been performed successfully, then the update will also exist even if the system fails. This is normally done by writing a log file where all actions are written possibly along with their transaction. This file is persisted before the status is returned to the user, then the database can recover from a crash by simply re-executing the log-file. This approach allows for the log-file to be completed very fast, as it can be written continuously during the transaction, and then completed in a single page.

## 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. An example where a specific design was used to avoid violating 1NF is the customer phone. Here e.g. a comma separated string would violate 1NF, and instead a separate table is used.

Functional dependencies are only relevant within a single table. For this reason only a few tables has any form of functional dependency. We would like to skip the customerpricingplan, as this is a special case where the functional dependency *id -> customerid,pricingplanid,productid* and *customerid,pricingplanid,productid -> id* only exist because productid=NULL prevents it from being part of the PRIMARY KEY (see earlier).

* customers
  + id -> webshopid, firstname, middlename, sirname, tvmfth, floor, streetletter, streetnumber, streetname, postalcode, region, country, paymentconditions
  + webshopid, firstname, middlename, sirname, tvmfth, floor, streetletter, streetnumber, streetname, postalcode, region, country -> id, paymentconditions

The reason everything is needed is because two people living at the same address may both have an account (Address not Candidate key). Also the same street name may exist in different postal codes in the same country (there are several stationsvej in Denmark), and it is therefore not possible to obtain the postal code from the street name and number (number might have been needed for a very long street). Region is just an optional extension (like state in USA) and can therefore not be used to obtain the postal code, but may be needed as two regions in the same country may have two areas with the same postal code. It is also not possible to obtain the country from the region as this may be NULL, or may have the same name in multiple countries, just like postal codes are not unique.

The reason the customers table has an id attribute is for two reasons; firstly to avoid having to use all 12 attributes in name and address whenever we want to reference a customer, and second because several of the attributes may be NULL and can therefore not be part of a PRIMARY KEY.

2nd is aimed at removing partial key dependencies. “*A table is in 2NF if and only if it is in 1NF and no non-prime attribute is dependent on any proper subset of any candidate key of the table*“. This prevents redundancies and fixes the problem with update abnormalities. 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 not within the same table. An example where 2NF could have been a problem if a different design had been chosen, had been if the town had been added to the customer address, as the postalcode -> town, and that would break 2NF (a new relation with postalcode-> town would have to be created).

3rd normal form removes dependencies on non-key attributes. “*A table is in 3NF if and only if it is in 2NF and every non-prime attribute of R is non-transitively dependent (i.e. directly dependent) on every superkey of R*”. paymentconditions are dependent on both candidate keys, but as 3NF only prohibits dependency on non-prime attributes (attributes not in any candidate key), this is OK.

Boyce-Codd normal form is very like the 3rd normal form, but also removes anomalies from relations with multiple candidate keys. “*A table is in BCNF if and only if it is in 3NF and every determinant is a candidate key (determinant is any attribute (simple or composite) on which some other attribute is fully functional dependent)*”. The anomalies in 3NF arise from overlapping candidate keys (candidate keys with at least one attribute in common). In customers there are no common attributes in the two candidate keys, and all dependent attributes (paymentconditions) is dependent on a candidate key. Therefore the relation meets the BCNF.

# 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

## Single purchase and sale

This is only a subsection of the complete execution. Please refer to the file miniproject\_create1.sql for a complete example. The notation ... has been used to indicate that something has been removed.

|  |
| --- |
| -- Be adviced that temporary tables are used to ensure that we have the auto generated IDs. In practice these IDs would exist in application and not in temporary tables.  CREATE TEMPORARY TABLE tempidcollection (productid INTEGER NULL, mpricingplanid INTEGER NULL, wpricingplanid INTEGER NULL, ppricingplanid INTEGER NULL, cpricingplanid INTEGER NULL, morderid INTEGER NULL, webshopid INTEGER NULL, customerid INTEGER NULL, corderid INTEGER NULL, ccocid INTEGER NULL, cinvoiceid INTEGER NULL, cdeliveryid INTEGER NULL);  INSERT INTO tempidcollection VALUES (NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);  -- Create a product  INSERT INTO products (name,weight) VALUES ('XB T-Shirt',156);  UPDATE tempidcollection SET productid=LASTVAL();  -- Create product attribute types  INSERT INTO productattributes VALUES ('productline','string');  INSERT INTO productattributes VALUES ('brand','string');  INSERT INTO productattributes VALUES ('size','string');  INSERT INTO productattributes VALUES ('sizemeasurement','integer');  INSERT INTO productattributes VALUES ('colour','string');  -- Create product attributes  INSERT INTO productattributerelations SELECT productid, 'productline','XB T-shirt' FROM tempidcollection;  INSERT INTO productattributerelations SELECT productid,'brand','XB' FROM tempidcollection;  INSERT INTO productattributerelations SELECT productid,'size','M' FROM tempidcollection;  INSERT INTO productattributerelations SELECT productid,'sizemeasurement',42 FROM tempidcollection;  INSERT INTO productattributerelations SELECT productid,'colour','green' FROM tempidcollection;  -- Create a manufactorer  INSERT INTO manufactorer VALUES ('CN34554345','ChinaProductionExcelence','CNY');  -- Create a pricing plan  BEGIN TRANSACTION;  INSERT INTO pricingplans (price,discount,deliveryconditions) VALUES (129.56,0.0,'SideOfShip');  UPDATE tempidcollection SET mpricingplanid=LASTVAL();  -- abLager does not really make sense here, as the websop never directly receive a shipment, so we just choose the minimum one. We could make an NA  INSERT INTO pricingplans (price,discount,deliveryconditions) VALUES (270,0.0,'abLager');  UPDATE tempidcollection SET wpricingplanid=LASTVAL();  INSERT INTO quantitydiscounts SELECT wpricingplanid,10,0.1 FROM tempidcollection;  INSERT INTO quantitydiscounts SELECT wpricingplanid,50,0.2 FROM tempidcollection;  INSERT INTO quantitydiscounts SELECT wpricingplanid,100,0.3 FROM tempidcollection;  INSERT INTO quantitydiscounts SELECT wpricingplanid,500,0.4 FROM tempidcollection;  INSERT INTO quantitydiscounts SELECT wpricingplanid,1000,0.5 FROM tempidcollection;  INSERT INTO pricingplans (price,discount,deliveryconditions) VALUES (599.00,0.0,'7dg');  UPDATE tempidcollection SET ppricingplanid=LASTVAL();  END TRANSACTION;  -- Create a manufactorer product  INSERT INTO manufactorerproducts SELECT 'CN34554345',productid,mpricingplanid FROM tempidcollection;  -- Place order with manufactorer  BEGIN TRANSACTION;  INSERT INTO manufactorerorders (manufactorerid,orderdate) VALUES ('CN34554345',CURRENT\_DATE);  -- This is a trick to the code can be executed in sequence, but in reality it would be an insert with a select of the id followed by another INSERT controlled by the application  UPDATE tempidcollection SET morderid=LASTVAL();  INSERT INTO manufactorerorderedproducts (orderid,productid,priceingplanid,count) SELECT morderid, productid, mpricingplanid, 1200 FROM tempidcollection;  END TRANSACTION;  -- After the COC has been received it is verified manually and entered. The order id will be on the COC.  BEGIN TRANSACTION;  INSERT INTO manufactorerorderconfirmations VALUES ('CN34554345','CPE00012',CURRENT\_DATE + integer '2');  UPDATE manufactorerorders SET cocid='CPE00012' WHERE orderid IN (SELECT morderid FROM tempidcollection);  -- The last where is not really needed  END TRANSACTION;  -- After the invoice is received it is verified manually and entered. The order id will be on the invoice.  -- The payment date may be calculated from the invoice date and the terms of payment, but we will postpone that for now.  BEGIN TRANSACTION;  -- Date format '2013-10-16'  INSERT INTO manufactorerinvoices (manufactorerid,invoiceno,invoicedate,paybefore) VALUES ('CN34554345', 'CPE-INV-00007',CURRENT\_DATE + integer '10',CURRENT\_DATE + integer '40');  UPDATE manufactorerorders SET invoiceid='CPE-INV-00007' WHERE orderid IN (SELECT morderid AS orderid FROM tempidcollection);  END TRANSACTION;  -- After the product is delivered  -- the 1200 from the order products are automatically added to the instock value  BEGIN TRANSACTION;  INSERT INTO manufactorerdeliveries VALUES ('CN34554345', 'DHL-274622', CURRENT\_DATE + integer '13');  UPDATE manufactorerorders SET freightno='DHL-274622' WHERE orderid IN (SELECT morderid FROM tempidcollection);  END TRANSACTION;  -- create web-shop  BEGIN TRANSACTION;  INSERT INTO webshops (vatno, name, paymentcurrency, invoiceaddress, paymentconditions) VALUES ('DK12765899','Exciting Webshops Inc.','DKK','Nyhavnsgade 46, 2300 København S','LbMntPlus15dg');  UPDATE tempidcollection SET webshopid=LASTVAL();  END TRANSACTION;  -- Add product to their portfolio  INSERT INTO webshopcarries SELECT webshopid, productid,wpricingplanid,ppricingplanid FROM tempidcollection;  ...  DROP TABLE tempidcollection; |

## 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 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.

Due to space limitations the an example of the query plan has been moved to the appendix.

## 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.

The reason the improvement is so great is that the index makes it much faster to find the entries with the given postal code. The index created is a B-tree dense non-clustering index by default, however as the postalcode has a limited number of combinations Postgresql uses a Bitmap fetch to further improve performance. This allows for the traversal of the index with much less page fetching.



In this example only Danish postalcodes are shown, but any text may be used. As it is text the database will have to have a way of comparing two strings. This could e.g. be done alphabetically one letter at a time using the ASCII value for the string. This allows for very fast look-up of a given value or range of values. The dense index is larger than the sparse index, but allows for fast determination if a value exist, which a sparse index does not. The non-clustering means that a list of pointers must be kept, where the clustering index keeps the matching values together and just a pointer to the start of the cluster is needed. This may also be faster to load from dist as more values will be on the same page, but it requires more reorganizing on insert/update/delete.

As it may be seen by the result of EXPLAIN ANALYZE, Postgresql uses bitmap-fetch to load the entire index in one go (the Recheck is because the entire index would not fit and a reduced version is used). The Index cond: ((postalcode)::text = '1010'::text) is the WHERE.

|  |
| --- |
| Bitmap Heap Scan on customers (cost=4.55..125.16 rows=38 width=103)  Recheck Cond: ((postalcode)::text = '1010'::text)  -> Bitmap Index Scan on customer\_postalcode\_idx (cost=0.00..4.54 rows=38 width=0)  Index Cond: ((postalcode)::text = '1010'::text) |

Without the filter it can be seen here that postgresql goes through all entries and filters out the not matching rows and removing them from the result (here 34935 rows was removed out of 34972).

|  |
| --- |
| Seq Scan on customers (cost=0.00..1024.15 rows=38 width=103)  Filter: ((postalcode)::text = '1010'::text)  Rows Removed by Filter: 34935 |

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 an integral part of database use. It is needed to ensure two properties of ACID; Isolation and Atomicity. Atomicity ensures that a group of updates can be done atomically, meaning that either all updates are done or none of them are. Isolation ensures that a group of actions can be performed in isolation, meaning that it may be guaranteed that one ongoing transaction cannot affect another ongoing transaction.

A database can have a more or less conservative approach to transactions and ACID. In its pure form an incomplete transaction is simply non-existent, and any attempt to access the uncommitted data should either wait for the transaction to complete. This is the safest solution, but also the ones that hurt performance the most.

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. It does allow for breading consistency in rare cases, as a read of the uncommitted data may be used as input for an update to another table. If the DB crashes after the first transaction completes, but before the uncommitted data is committed, then the restore from the log-files will only restore the data which was based on the uncommitted data, which is not restored. This is however a very unlikely scenario, but possible.

There are two situations in this database where transactions are very important; 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.

# Conclusion

Before the making of this project we had all worked with databases before on practical level. During the development of this database and the writing of the report we got to work with the database in a more theoretical level, and have been able to try out the different database design principles and working with indexes and performances more specifically. In the industry it is rare that you have the opportunity to work with the different project this thoroughly, which has given us a very good understanding of database design and the different methods and tools employed.

During the design and implementation of the database many realizations were reached; is the flexibility of the per-product pricing plan necessary, or would it be better to have a product group entity which groups products around a pricing plan? This would prevent sales on specific sub products in the product group, but would simplify the design. Also the immutable nature of the Pricing plan gives cause for concern, as it is imperative that a pricing plan is not changed, as it might be shared by other products/webshops/customers. However, adding a new pricing plan may lead to unreferenced pricing plans (wasting space). It is also important to consider the level of normalization. If many joins are performed very often, then some level of denormalization may be preferred to obtain higher performance.

Many of these considerations can best be determined after the design and implementation has been field tested for a while, so it is possible to accurately determine the load and use of the system and naturally this design refactoring is an integral part of a database design, but will be left for a late time.

# 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.

## Query plan example

...