Valmart Management System
CS387: Databases and Information Systems Lab

Divyam Bansal  Guna Prasaad  Vipul Venkataraman  C Yeshwanth
110050086     110050082     110050084     110050083
Problem Specification

Name of the Project
Valmart Management System

Application domain
For our CS 387 project, we intend on developing a back end branch management system for an organization like Valmart. In such large organizations that maintain a huge inventory stock there is a need for an integrated system to place orders with vendors, manage and exchange inventories among different branches and maintain customer and employee records. We aim at developing such an integrated system to achieve the aforementioned functionalities.

Specifications
As mentioned above, our system tries to model and keep track of all the aspects of a branch. We also provide information of the company as a whole, where we query and collect data from all the branches and provide statistics regarding their performance. We have a three tier employee system for the company. Namely an employee, a manager and a chief manager. We also have vendors who are responsible for providing branches with inventories on request. These are the people who will use the system. More details regarding specific components of our model are as follows:

- **Operational Hierarchy:** Our system models a network of many branches for a specific organization spread across a region. Hence here we store details of a specific store such as its region, its size, the number of departments the store has etc. We also assign a specific manager to each branch

- **Merchandise Hierarchy:** We model the hierarchy of a merchandise here. We have a merchant group who acts as the manufacturer of any item. We bring in vendors into the system here since each merchandise comes into a store through a specific vendor. All the information regarding a product like its vendor ID, its manufacture data, dimensions are stored here. The retail price of each product that depends on the store, the item and the vendor is also modeled here

- **Point of Sale:** Several components regarding a point of sale transaction are stored here. These involve information like the customer who is buying the inventories, the amount of discount we provide for his purchase, whether he avails additional discount because he may be a frequent shopper and additional information with respect to a transaction like the store ID, the operator who is handling the transaction and the inventory list

- **Inventory:** This is an important component of our system where information regarding all the inventories are maintained. We store the inventories that are present on display at each specific floor of a branch. We therefore maintain the current inventory of a branch and for every floor of the branch. We also maintain the inventory history which will be pretty vital in computing statistics regarding a branch or a specific product

- **Planned Inventory:** We propose an inventory plan for every branch to its branch manager by predicting the inventories that would be required in future depending on current stock and past transactions in that specific branch. We will develop an Artificial Intelligence algorithm that will facilitate the manager into placing the required order to vendors

- **Orders:** Since we provide a planned inventory for each branch manager, he should be able to place order depending on what our system suggests him. We therefore provide an interactive interface to the branch manager that will enable him into placing orders for his branch from vendors. Here we have information regarding the vendor in question, details regarding the
purchase order such as the branch manager and the time, and how the order is distributed within the store. Like how the products are placed in the various floors.

- **Merchandise Summaries**: Here we provide various statistics regarding the sales of a particular merchandise. We haven’t finalized on what all feature we need to provide but we will have information such as the sales of a product over a period of time and within any time frame. This will help a manager to gauge how well a product is selling amongst his customers. We will also provide data regarding the performance of a product across various branches and regions. This information that concerns with the global perspective of a product across the company will be available only to the chief manager.

- **Operational Summaries**: Here we provide statistics regarding the operational performance of a branch as well as an employee. We provide information such as the store sales over a period of time and within any given time frame. We draw comparisons across various branches to find out who is making most business. We also plan to provide various other information such as the best branch and the best selling product.

**Non deliverables**

- Since we have decided to focus more on the back-end management of a branch, we have not taken records as to how a cash transaction was made between a customer and the organization. For example, we can incorporate transactions using cash, credit card, gift coupons and other tenders.

- The inventories that a specific branch can hold has not been restricted. On a realistic point of view, this is not the case because we have a space constraint. Since the dimension of each product then has to be taken into account and implementing this would be cumbersome, we decided to not have an inventory limit for a branch.

**Member Specific Roles**

The exact roles of each team member has not been finalized. We plan to divide our work for this project as we go further along.

**Future Extensions - Other ‘value - added’ facilities**

With the current model of our schema, we have scope for the development of the following functionalities. However, we will take a decision on implementing them depending on how the project unfolds with time.

- Our model supports giving gift coupons to customers depending on the amount of good they have bought. This will induce a customer in visiting the company again, thereby increasing the revenue of the company.

- We can include customer feedback as an extension to the proposed model. For every transaction, a customer can give his/her feedback on how they found the service to be, on whether the store should consider giving more discounts etc.

- We can model discount rates depending upon the season. For instance, we provide discounts on goods during the festive season where we expect a lot of purchase.

**References**

1. Data Warehousing *using the Walmart model* by Paul Westerman
2. Slides on Database Concepts by Prof. Umesh Bellur
Entity-Relationship Diagrams

Introduction

In this report, we have used the Functional Specifications of our project and have come up with Entity Relationship diagrams that models our application.

Salient Features

Since our model logically has different components, we felt providing an E/R diagram for each of the logical units mentioned below would be appropriate. We have therefore included an E/R model for the following components:

- Operational Hierarchy
- Merchandise Hierarchy
- Point of Sale
- Inventory
- Orders

In our functional specification, we have Planned Inventories, Merchandise Summaries and Operational Summaries included as components of our model. But these components deal with statistical inference of the data and hence we have not provided E/R models for the same. We have also implemented the following features in our model

- A number of entity sets
- Relationships between Entity Sets
- Ternary and Quaternary Relationships
- Weak entity sets

Inheritance was one criteria we could have modeled since we have a three tier employee system. But providing roles to employees seemed more natural and hence we limited ourselves to roles.

E/R Models

The Entity Relationship diagram for each of the components is provided below.
Operational Hierarchy

MERCHANDISE HIERARCHY

VENDOR
- VEND_ID
- VEND_NAME
- VEND_ADDR1
- VEND_ADDR2
- VEND_CITY
- VEND_COUNTRY
- VEND_SIP
- VEND_CONTACT_NAME
- VEND_CONTACT_PHONE
- VEND_CONTACT_EMAIL
- VEND_SEC_PHONE

MERCHANDISE
- MGRS_GROUP_ID
- MGRS_GROUP_NAME
- MGRS_GROUP_MGR_NAME
- MGRS_ADDR1
- MGRS_ADDR2
- MGRS_CNY
- MGRS_COUNTRY
- MGRS_ZIP
- MGRS_CONTACT_NAME
- MGRS_CONTACT_PHONE
- MGRS_CONTACT_FAX
- MGRS_CONTACT_EMAIL
- MGRS_SEC_PHONE

PRICE
- SELL

PRODUCT
- ITEM_ID
- ITEM_STATUS
- ITEM_DESC
- CREATE_DATE
- ITEM_COLOR
- CODE
- MIN_RPT_AMT
- MAX_RPT_AMT
- PLASNAME_CODE
- HAZARD_CODE
- WT_PER_UNIT
- X_AXIS_MEASURE
- Y_AXIS_MEASURE
- Z_AXIS_MEASURE

STORE
- ...
Point of Sale

Inventory
Orders

Note: Colored blocks represent weak entity sets and relationships

Acknowledgements and References

1. We used a tool provided by [http://gliffy.com](http://gliffy.com) to draw our E/R diagrams
2. Slides on E/R modelling by Prof. Umesh Bellur
Test Plan

Introduction

In this report we explain our Test Plan. Testing forms an integral part for any application and hence we explain the procedure through which we test the correctness and exception handling functionalities we have implemented in our application. We also intend to test parameter verification and backend functionality of our system.

Methods used

We have used the following testing methods to test our system:

- **Static testing**: This type of testing involves reviews, walkthroughs and inspections made during the designing and implementation of our application. Static testing involves mostly verification that we do back end

- **Dynamic testing**: Here we actually execute programmed code with a given set of test cases to test the functionality of our application. We employ dynamic testing as we develop our application over the various components to test particulate sections and when we integrate all of them to complete our application

Since static testing is done back end and more on the fly, we explain the various dynamic testing methodologies we plan to employ in this report. We also explain how we use each method to test our model in particular.

Structural testing

We test the internal structures and the performance of our program, as opposed to the functionalities exposed to the end user. We design test cases as inputs to our program to exercise paths through the code and determine the appropriate outputs. In specific to our application:

- We will test whether the data is stored according to our Entity Relationship model by making sure the data is consistent
- We will design queries to verify the various functionalities we plan to provide. These include statistical inferences, data with respect to a customer etc We then use the test data to check if the result we obtain is consistent with what is expected

Black box testing

Here, we treat our software as a ”black box” and examine the functionality without any knowledge of internal implementation.

- **Account validation and authentication**: We have a three tier employee system and hence have interfaces for each type of employee. We test the username, password, data provided related to specific fields like the employee ID, make sure the compulsory fields are not empty
- **Sanity checks**: Since we provide interfaces to place orders to vendors, we need to make sure the data entered regarding the inventories to be bought are logical. The quantity for each commodity, for instance should be a natural number
- **Login**: We test for incorrect username, incorrect passwords and empty fields when a user tries to log in through our interface
Security and Accessibility

Making sure an employee does not gain access unwanted information is very essential in our system. The three types of employees we have is the normal employee, managers and a chief manager. The various cases we have considered here are:

- We need to make sure a normal employee does not see information pertaining to managers alone
- The employee should not be able to place orders to vendors
- A manager is specific to a branch, he should not be able to see information that corresponds to a different branch like the inventory list, performance of a different branch
- The chief manager should have access to information regarding all the branches of a company

References
Relational Model

Introduction

In this section, we explain the Relational Model of our application. The relational model at this stage, obtained from our Entity Relationship diagrams would not be normalized. The goal is to deliberately introduce redundancy in data at this stage, so that we can demonstrate the various normalization techniques for the next stage of the project and improve the current design.

Relationships

To have a clear understanding of how we have modeled our system, we have enumerated some relationships of our model along with the entity sets involved

<table>
<thead>
<tr>
<th>Relationship name</th>
<th>Entity Sets involved</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</td>
<td>Store, Floor</td>
<td>One to Many</td>
</tr>
<tr>
<td>Caters</td>
<td>Department, Store_dept, Floor</td>
<td>One to Many</td>
</tr>
<tr>
<td>Location</td>
<td>Store, Region</td>
<td>Many to One</td>
</tr>
<tr>
<td>Serves</td>
<td>Department, Product</td>
<td>One to Many</td>
</tr>
<tr>
<td>Manf</td>
<td>Product, Merchandise</td>
<td>Many to One</td>
</tr>
<tr>
<td>Vendors</td>
<td>Store, Vendor</td>
<td>Many to Many</td>
</tr>
<tr>
<td>Sells</td>
<td>Product, Vendor</td>
<td>Many to One</td>
</tr>
<tr>
<td>Cust_visit</td>
<td>Store, Customer, Employee, Pos_Transaction</td>
<td>Many to Many</td>
</tr>
<tr>
<td>Purchase Order</td>
<td>Store_dept, Vendor</td>
<td>Many to Many</td>
</tr>
</tbody>
</table>

Conversion

Many to One

We first consider the Many to One relationship Sells (price)

**Vendor** (Vndr_ID, Vndr_name, Vndr_addr1, Vndr_addr2, Vndr_city, Vndr_country, Vndr_zip, Vndr_contact_name, Vndr_contact_phone, Vndr_contact_fax, Vndr_contact_email, Vndr_sec_phone)

- Primary Key: Vndr_ID
- Functional Dependencies: Vndr_ID -> Vndr_name, Vndr_addr1, Vndr_addr2, Vndr_city, Vndr_country, Vndr_zip, Vndr_contact_name, Vndr_contact_phone, Vndr_contact_fax, Vndr_contact_email, Vndr_sec_phone

**Product** (Item_ID, Item_status, Item_desc, Create_date, Item_color_code, Item_size_code, min_retail_amnt, max_retail_amount, flammable_code, hazard_code, wt_per_unit, x_axis_measure, y_axis_measure, z_axis_measure)

- Primary Key: Item_ID
- Functional Dependencies: ItemID -> Item_status, Item_desc, Create_date, Item_color_code, Item_size_code, min_retail_amnt, max_retail_amount, flammable_code, hazard_code, wt_per_unit, x_axis_measure, y_axis_measure, z_axis_measure

We merge the attributes of the relation Sells we would normally have into the attributes of the Relation for Vendor. Relation resulting from many to one relationship Sells due to deviation:
Vendor (Vndr_ID, Item_ID, price, Vndr_name, Vndr_addr1, Vndr_addr2, Vndr_city, Vndr_country, Vndr_zip, Vndr_contact_name, Vndr_contact_phone, Vndr_contact_fax, Vndr_contact_email, Vndr_sec_phone)

Problems Introduced: We have included the candidate attributes of Product into the new relation for Vendor. So, Vendor has Item_ID as a field. Introduction of this attribute into the relation for Vendor would result in redundancies of data. This is essentially because for every product that a vendor might sell, the other attributes of the Vendor like his Vndr_contact_phone, Vndr_name are repeated.

Many to Many

We now consider the Many to Many relationship Purchase Order (Entry_date, Ship_data, TTL_units, TTL_amt)

Vendor (Vndr_ID, Vndr_name, Vndr_addr1, Vndr_addr2, Vndr_city, Vndr_country, Vndr_zip, Vndr_contact_name, Vndr_contact_phone, Vndr_contact_fax, Vndr_contact_email, Vndr_sec_phone)

- Primary Key: Vndr_ID
- Functional Dependencies: Vndr_ID -> Vndr_name, Vndr_addr1, Vndr_addr2, Vndr_city, Vndr_country, Vndr_zip, Vndr_contact_name, Vndr_contact_phone, Vndr_contact_fax, Vndr_contact_email, Vndr_sec_phone

Store_dept (Store_ID, Floor_ID, Dept_ID, Store_dept_mngr)

- Primary Key: Store_ID, Floor_ID, Dept_ID
- Functional Dependencies: Store_ID, Floor_ID, Dept_ID -> Store_dept_mngr

We merge the attributes of the relation Purchase Order we would normally have into the attributes of the Relation for Store_dept. Relation resulting from many to one relationship Purchase Order due to deviation:

Store_dept (Store-ID, Floor_ID, Dept_ID, Vndr_ID, Store_dept_mngr, Entry_date, Ship_data, TTL_units, TTL_amt)

Problems Introduced: We have included the candidate attributes of Vendor into the new relation for Store_dept. So, Store_dept has Vndr_ID as a field. Introduction of this attribute into the relation for Store_dept would result in redundancies of data. This is essentially because for every Vendor, the other attributes of Store_dept like its Store_dept_mngr, Ship_date are repeated.

References

Screen Designs

Interface of Login Screen

Figure 1: Employees of all grades will use the above interface to login to the system

Interface for Store Manager

Figure 2: Interface for the store manager - each icon leads to different sub-management systems
Figure 3: Displays all the products sold in the store along with vendors, price and stock in hand. Products can be added or price of products can be modified using above interface.

Figure 4: Displays insights about the sales in the store.
Interface for Order Management

Figure 5: Displays orders placed from the store to different vendors. The manager can click on the order reference numbers to approve orders waiting for approval.

Figure 6: Interface to add/approve an order.
Interface for Point of Sale

Figure 7: Operator interface at the billing counter

Interfaces for Vendor Management

Figure 8: Search for vendors based on name or ref
Figure 9: Add a new vendor

Figure 10: Contact Information of the vendor
Figure 11: *Price Catalogue of the vendor*

Figure 12: *Orders placed with the vendor*
Interface for Manufacturer Management

![Manufacturer Management Interface](image)

**Figure 13:** Search for manufacturers based on name or ref

![Manufacturer Management Interface](image)

**Figure 14:** Add a new manufacturer
Figure 15: Contact Information of the Manufacturer

Figure 16: Catalogue of the Manufacturer
Interface for Product Management

Figure 17: *Add a new product*

Figure 18: *Information about the product*
References and Acknowledgements

2. Balsamiq Mockups [http://balsamiq.com/](http://balsamiq.com/)
Normalization

Introduction

In this section, we normalize all the relations of our chosen schema first into 3NF, and if needed we further normalize them into BCNF. We do not normalize all relations into BCNF and we have retained some of them in their 3NF form. Reasons pertaining to why we chose to normalize a particular relation to its final normalized form has been explained whenever needed.

Model Specifics

Before we explain how we have normalized the relations, we enumerate few observations and features of our schema

- Since our relations were initially constructed and designed with thought and foresight, most relations that exist in our relational model do not have any non trivial functional and multivalued dependencies.
- Most of the dependencies are on keys, which are trivial. We therefore have not made any changes to these relations, this is because these relations are already in 4NF form and can directly be used.
- We have decomposed schemas where we have found dependencies and have also explained the data redundancies we have removed.
- Since our relational model has no non-trivial dependency in any schema, the final design of all those schema that are initially not in BCNF or 3NF is in 4NF. The decomposed forms formed due to normalization are also in 4NF.

Operational Hierarchy

In this section we consider the relations that are part of the Operational Hierarchy component of our schema

Region (Region_ID, Region_name)
Functional Dependencies:
- Region_ID -> Region_name

Since there is only one functional dependency involving the only two attributes in the relation, the above relation is in BCNF form and does not need any decomposition.

Department (Dept_ID, Dept_name)
Functional Dependencies:
- Dept_ID -> Dept_name

Since there is only one functional dependency involving the only two attributes in the relation, the above relation is in BCNF form and does not need any decomposition.

Store (Store_ID, Store_name, Store_mgr, Addr_1, Addr_2, City_name, State_code, Country_code, Zip_code, Open_date, Total_sqft)
Functional Dependencies:
Since there is only one functional dependency and that involves the key, this relation is already in BCNF form and won’t require any further decomposition

\[ \text{Store dept} \ (\text{Store ID}, \text{Floor ID}, \text{Dept ID}, \text{Store Dept Mgr}) \]

Functional Dependencies:

- \( \text{Store ID, Floor ID, Dept ID} \rightarrow \text{Store Dept Mgr} \)

The above relation is clearly in BCNF and won’t require any further decomposition due to the absence of redundancy of data

\[ \text{Floor} \ (\text{Store ID}, \text{Floor ID}, \text{Floor sqft}, \text{Floor desc}) \]

Functional Dependencies:

- \( \text{Store ID, Floor ID} \rightarrow \text{Floor sqft, Floor desc} \)

The above relation is clearly in BCNF and won’t require any further decomposition

### Merchandise Hierarchy

In this section we consider the relations that are part of the Merchandise Hierarchy component of our schema

\[ \text{Vendor} \ (\text{Vndr ID}, \text{Vndr name}, \text{Vndr addr1}, \text{Vndr addr2}, \text{Vndr city}, \text{Vndr country}, \text{Vndr zip}, \text{Vndr contact name}, \text{Vndr contact phone}, \text{Vndr contact fax}, \text{Vndr contact email}, \text{Vndr sec phone}) \]

Functional Dependencies:

- \( \text{Vndr ID} \rightarrow \text{Vndr name, Vndr addr1, Vndr addr2, Vndr city, Vndr country, Vndr zip, Vndr contact name, Vndr contact phone, Vndr contact fax, Vndr contact email, Vndr sec phone} \)

Since there is only one functional dependency in the relation involving the only key, the above relation is in BCNF form and does not need any decomposition

\[ \text{Merchandise} \ (\text{Mds group ID}, \text{Mds group name}, \text{Mds addr1}, \text{Mds addr2}, \text{Mds city}, \text{Mds country}, \text{Mds zip}, \text{Mds contact name}, \text{Mds contact phone}, \text{Mds contact fax}, \text{Mds contact email}, \text{Mds sec phone}) \]

Functional Dependencies:

- \( \text{Mds group ID} \rightarrow \text{Mds group name, Mds addr1, Mds addr2, Mds city, Mds country, Mds zip, Mds contact name, Mds contact phone, Mds contact fax, Mds contact email, Mds sec phone} \)

Since there is only one functional dependency in the relation involving the only key, the above relation is in BCNF form and does not need any decomposition

\[ \text{Product} \ (\text{Item ID}, \text{Item status}, \text{Item desc}, \text{Create date}, \text{Item color code}, \text{Item size code}, \text{min retail amt}, \text{max retail amt}, \text{flammable code}, \text{hazard code}, \text{wt per unit}, \text{x axis measure}, \text{y axis measure}, \text{x axis measure}) \]

Functional Dependencies:

- \( \text{Item ID} \rightarrow \text{Item status, Item desc, Create date, Item color code, Item size code, min retail amt, max retail amt, flammable code, hazard code, wt per unit, x axis measure, y axis measure, x axis measure} \)

The above relation is clearly in BCNF and won’t require any further decomposition

\[ \text{Sells} \ (\text{Item ID}, \text{Vndr ID}, \text{Price}) \]

Functional Dependencies:

- \( \text{Item ID, Vndr ID} \rightarrow \text{Price} \)

The above relation is clearly in BCNF and won’t require any further decomposition

\[ \text{Retail Price} \ (\text{Store ID}, \text{Item ID}, \text{Price}) \]

Functional Dependencies:
• Store_ID, Vndr_ID -> Price

The above relation is clearly in BCNF and won’t require any further decomposition due to the absence of redundancy of data

**Point of Sale**

In this section we consider the relations that are part of the Point of Sale component of our schema

**POS_Transaction** (Trans_seq_nbr, Register_nbr, Trans_timestamp, Trans_type, Ttl_item_cnt, Ttl_units_qty, Ttl_sales_amt, Discount_amt, Tender_type)

Functional Dependencies:
- Trans_seq_nbr, Register_nbr -> Trans_timestamp, Trans_type, Ttl_item_cnt, Ttl_units_qty, Ttl_sales_amt, Discount_amt, Tender_type

Since there is only one functional dependency in the relation involving the primary keys, the above relation is in BCNF form and does not need any decomposition

**PCS_Merchandise** (Trans_seq_nbr, Register_nbr, Store_ID, Item_ID, Sell_qty)

Functional Dependencies:
- Trans_seq_nbr, Register_nbr, Store_ID, Item_ID -> Sell_qty

The above relation is clearly in BCNF and won’t require any further decomposition due to the absence of redundancy of data

**Customer** (Customer_ID, Cust_name, Cust_type, Cust_addr1, Cust_addr2, Cust_city, Cust_country, Cust_zipcode, First_shop_timestamp, Last_shop_timestamp, Ttl_purchase_amt)

Functional Dependencies:
- Customer_ID -> Cust_name, Cust_type, Cust_addr1, Cust_addr2, Cust_city, Cust_country, Cust_zipcode, First_shop_timestamp, Last_shop_timestamp, Ttl_purchase_amt

Since there is only one functional dependency in the relation involving the primary keys, the above relation is in BCNF form and does not need any decomposition

**Inventory**

In this section we consider the relations that are part of the Inventory component of our schema

**Adjust_Inventory** (Store_ID, Product_ID, Adjustment_Code, Adjustment_qty, On_hand_qty)

Functional Dependencies:
- Store_ID, Product_ID, Adjustment_Code -> Adjustment_qty, On_hand_qty

The above relation is clearly in BCNF and won’t require any further decomposition

**Inventory_history** (Store_ID, Product_ID, EOP_date, EOP_on_order_qty, EOP_in_whs_qty, EOP_in_transit_qty, EOP_on_hand_qty, timestamp)

Functional Dependencies:
- Store_ID, Product_ID -> EOP_date, EOP_on_order_qty, EOP_in_whs_qty, EOP_in_transit_qty, EOP_on_hand_qty, timestamp

The above relation is clearly in BCNF and won’t require any further decomposition

**Current_Inventory** (Store_ID, Product_ID, Adjustment_Code, Adjustment_qty, On_hand_qty)

Functional Dependencies:
- Store_ID, Product_ID, Adjustment_Code -> Adjustment_qty, On_hand_qty

The above relation is in BCNF and won’t require any further decomposition
Orders

In this section we consider the relations that are part of the Merchandise Hierarchy component of our schema

**Purchase_Order** (Store_ID, Vendor_ID, Item_ID, Entry_date, Ship_date, Ttl_units, Ttl_amt)

Functional Dependencies:

- Store_ID, Vendor_ID, Item_ID -> Entry_date, Ship_date, Ttl_units, Ttl_amt

Since there is only one functional dependency in the relation involving the primary keys, the above relation is in BCNF form and does not need any decomposition

**PO_Order** (Store_ID, Vendor_ID, Item_ID, Order_qty, Sub_ttl_amt)

Functional Dependencies:

- Store_ID, Vendor_ID, Item_ID -> Order_qty, Sub_ttl_amt

The above relation is clearly in BCNF and won’t require any further decomposition

References

Schema and Data Population

Introduction

In this section, provide a brief overview of the SQL scripts written to create the Database schema we needed for our project. We also give an explanation of the constraints used wherever applicable. We also explain how we populated our database with test data. We had written scripts in python to web scrape the required data. We then transformed the data obtained into the form that is acceptable for use in PostgreSQL.

Schemas

We explain the various Schemas we created along with the Triggers in them

- Scripts were written separately for the different components of our database. The different components of our database are Inventories, Operations, Orders, Point of Sale and Products
- The inventory portion consists of the current inventory and the inventory history tables. The data in these tables are automatically filled using triggers
- Similarly the schema for all other relations were create according to our Entity Relationship Model
- The current inventory is updated using a trigger when we insert a record into the retail price relation
- Upon insertion into the po_line table, we update the total units and the total amounts of the corresponding product in the purchase order table
- Once the shipped flag in the order is set to true, meaning the order has been shipped to the store, we update the current inventory of that store with the items obtained using the order

Data Population

- We wrote python scripts to populate the data of our database
- We generated data for the tens of relations we have using about randomization and web scrapping
- We generated around 120 Megabytes of data for all our relations combined
- We took data regarding the names, addresses, cities from different websites
- We took data regarding the names of different products along with their categories from flipkart.com

References

4. www.fakenamegenerator.com
Testing

Introduction

In this section, we explain the testing process we had performed along with the conclusions of our project and the future extensions. We briefly cover the various aspects of the white-box testing we performed. All our queries are mostly sql insert and select statements. We tested the working of the triggers we had written, to test the consistency of our database.

ACID Properties

Our Database follows the ACID properties.

- **Atomicity**: Our database follows the atomicity property, all or nothing executions. This is essentially because we had implemented transactions

- **Consistency**: Our database is also consistent. We tested this by various queries we had designed to ensure all possible cases that might occur

- **Isolation**: The queries we handle in our model are inherently isolated in nature. Two queries are generally not dependent on each other

- **Durability**: Our database is also durable. However, this is not handled by us

Indices

By default, the only indices that the PostgreSQL server at the backend makes are those on the primary keys of each table. They are all BTree indices. Since most of the queries and SQL commands we use were largely on the primary key attributes, we didn’t have the need to generate separate indices to speed up our queries.

Error Handling

Basic error handling like login and filling up forms were also handled

- **Login**: When a user tries to login and he provides the wrong credentials, then he is notified as entering wrong inputs

- **Forms**: When a user submits incomplete or wrong information in forms, the data is not processed and the user is again prompted to fill in the form again

Performance Tests

The basic goal in this project was to implement a very succinct, self sustaining database. The speed of each query not necessarily important or vital. We focussed more on providing more dynamic and real life data and implement that. We had initially written our queries as different SQL statements and timed their execution. We then chose the SQL statements that was most efficient.

Cursors

As a future enhancement, we intend to implement cursors which facilitates row wise processing of the result set. We therefore read 10 rows at a time rather than read the whole result of the query. This improves performance drastically and saves memory when the number of rows in the result set is very large.
Conclusions

In this project, we have attempted to build a branch management system and in this endeavor we had one primary goal and that goal was to maximize the design and structure of our database. We would go so far to say that we have been fairly successful at this endeavor. We have supported good inventory management functionalities, along with sales insights and automatic order generation.

The better part of our project was spent on designing the database cause our proved to be very complex. Since the number of relations were enormous, it was an elaborate and time intensive process to ensure that all the cases would work as expected.

The next mountainous task was the testing of our final project. Given the huge amount of data we are trying to support, the number of possible input combinations to the program were huge. As far as other features are concerned, we tried to incorporate as much as possible. We have included a number of data integrity constraints, which ensure data correctness. We have included transactions in our input process.

References

3. http://www.gcreddy.net/2010/03/database-testing.html#.UpWy-2QW1I4