

Grocery Delivery Database

Database Project

CMPS 3420

Spring 2019

Group Number 5:

Mitchell Alvarado

Sean Fontes

Rodrigo Ortiz

Table of Contents

<u>Phase 1: Data Collection and Conceptual Database Design</u>	3
<u>1.1 Information Gathering Methods</u>	3
<u>1.1.1 Introduction to Enterprise</u>	4
<u>1.1.2 Description of Fact-Finding Techniques</u>	5
<u>1.1.3 Database Design Focus</u>	6
<u>1.1.4 Entity and Relation Set Definitions</u>	6
<u>1.1.5 User Groups, Data Views, and Operations</u>	12
<u>1.2 Conceptual Database Design</u>	12
<u>1.2.1 Entity Types</u>	13
<u>1.2.2 Relationship Types</u>	33
<u>1.2.3 Related Entity Types</u>	43
<u>1.2.4 E-R Diagram</u>	44
<u>Phase 2: Conceptual Database and Logical Database</u>	46
<u>2.1 E-R Model and Relational Model</u>	46
<u>2.1.1 Description of E-R Model and Relational Model</u>	47
<u>2.1.2 Comparison of Two Different Models</u>	47
<u>2.2 From Conceptual Database to Logical</u>	48
<u>2.2.1 Converting Entity Types to Relations</u>	49
<u>2.2.2 Converting Relationship Types to Relations</u>	50
<u>2.2.3 Database Constraints</u>	55
<u>2.3 Converting Conceptual to a Relational Database</u>	57
<u>2.3.1 Relation Schema for Local Database</u>	58
<u>2.3.2 Sample Data of Relation</u>	81
<u>2.4 Sample Queries to Database</u>	104
<u>2.4.1 Design of Queries</u>	105

2.4.2	<u>Relational Algebra Expressions for Queries</u>	106
2.4.3	<u>Relational Calculus for Queries</u>	108
2.4.4	<u>Tuple Relational Calculus Expressions for Queries</u>	108
2.4.5	<u>Domain Relational Calculus Expressions for Queries</u>	110
	<u>Phase 3: Logical and Conceptual Database with Postgres DBMS</u>	113
3.1	<u>Normalization of Relations</u>	113
3.1.1	<u>Normalization Forms and Anomalies</u>	113
3.1.2	<u>Relation Analysis and Updates</u>	118
3.2	<u>Purpose and Functionality of PSQL</u>	127
3.3	<u>Schema Objects in PostgreSQL</u>	129
3.3.1	<u>Schema Objects Definitions</u>	129
3.4	<u>Relation Schema and Contents</u>	136
3.4.1	<u>Relation Schema</u>	136
3.5	<u>Query Conversion to SQL</u>	144
3.6	<u>Data Loader Descriptions</u>	161
	<u>Phase 4: DBMS Procedural Language, Stored Procedures, and Triggers</u>	167
4.1	<u>Postgres PL/pgSQL</u>	167
4.1.1	<u>Postgres PL/pgSQL Language</u>	169
4.2	<u>Postgres PL/pgSQL Subprograms</u>	170
4.3	<u>PL/pgSQL-Like Languages in Microsoft SQL Server, MySQL and Oracle DBMS</u>	176
4.3.1	<u>Microsoft SQL Server: T-SQL</u>	176
4.3.2	<u>MySQL</u>	176
4.3.3	<u>Oracle DBMS: PL/SQL</u>	176
	<u>Phase 5: Graphical User Interface</u>	187
5.1	<u>Functionalities and User Group of GUI Application</u>	188

<u>5.1.1 User Group Descriptions</u>	189
<u>5.1.2 Tables and View Descriptions</u>	192
<u>5.2 Programming</u>	198
<u>5.2.1 Server-Side Programming</u>	201
<u>5.2.2 Middle-Tier Programming</u>	207
<u>5.3 Learning and Implementing</u>	210
<u>5.4 Survey</u>	211

Phase 1: Data Collection and Conceptual Database Design

Over the course of this phase, a foundation will be established concerning the roots of the design for the database. Specifically, we will be taking a look through the business itself, as well as some of the varying methods by which we obtained the relevant information concerning the business. Following that, we will establish the focus of the database in the context of the information in relation to the business. After creating the focus, a basis for the entity design will be explored, as well as analyzing some of the main user groups for the finalized database.

With the basis detailed, we will then expand into creating a more concrete structure—outlining both entities, relations, and providing more information as to the data types and cardinality intended for each attribute and relation. Lastly, the section will culminate in the visual depiction of the data in the form of an E-R Diagram.

1.1 Information Gathering Methods

Before diving directly into the meat of the database design, first we need to understand the needs of the business, and what the aims of the database should be designed to address. As such, this section will first analyze the business itself, particularly the structure of the business, and how it will impact the design of the database. Of course, in order to analyze the business, information concerning the business must first be obtained, and the methodology by which we obtained said information will also be covered in the following sections.

After the nature of the business has been explored, we will create a preliminary outline of the entities and relation sets that will be used to form the base idea on which the database will be structured. In the last subsection, the user groups for the database will also be analyzed to refocus the aims for the design of the end result.

1.1.1 Introduction to the Business

With the onset of the internet, customers' needs are rapidly evolving, and businesses are forced to keep pace in order to maintain their audience. As such, one of the many new trends is home delivery, as more and more stores take to online shopping for their prime source of revenue. More specifically to the matter at hand, home delivery of frozen goods. Schwan's uses a model similar to that of a larger chain such as Amazon. In contrast, however, Schwans has a much more personal relationship with their consumers, as the deliveries are handled within the scope of the business itself, rather than using a third party to ship it.

Taking this into account, the needs of the business must then also account for the delivery and manufacturing of the products as it's all done in house. One last key

thing to note: although Schwan's was the inspiration, our database will be a more generalized model for the business structure. With the basics of the scope of the business established, let's now take a look at the methods that were used to determine some of the specifics in regards to the business.

1.1.2 Information Gathering Techniques

There were two key sources of information at play when the business was analyzed. The first of which being the website, and prime selling point of the business itself. From there, a baseline was established clarifying some of the key aspects of the business. Of which, it could be concluded the business had its own manufacturing department, delivery department, and assumedly a corporate aspect to it as well. From there, it was decided to focus on the concrete information provided in regards to the manufacturing and delivery department.

Some basic questions arose - such as does the business still rely on suppliers or does the manufacturing department handle the entirety of the production for the business. Luckily, there was an inside source to refer to: Rodrigo's father, who works directly for Schwan's, providing a more intimate look at the business itself. He provided the confirmation that the business did indeed produce all of their products themselves, meaning suppliers no longer need be factored into the equation for the database, but rather instead only look at the cost of the materials needed to manufacture the products. Between the sources available online from the company itself and the insider perspective, it was possible to establish a baseline on which to look at the data of the company.

Personal information of the employees, department IDs, supplier IDs, client information, order history, and product information would be stored on the database. The stock of product would be updated when suppliers deliver their product for the business to sell and when clients order products. Managers would enter that information as they would be the ones to control the department. The database would need to continually be updated when suppliers bring sell their supplies to the departments.

1.1.3 Database Design Focus

With the nature of the information we were able to discover, it was decided to go for a more generalized approach to the design of the database. Had we stuck strictly to Schwan's rather than the less specific design we are now aiming for, we felt it would have been necessary to take into account more of the corporate aspect of the business. Instead, with our approach, we can focus on the function of the business itself, in place of incorporating more of an approach to business nature.

Taking a more loose approach to the hierarchy of the business allowed for a more rigorous analysis regarding the functionality of the business: specifically in the case of our database, the highest degree of focus was placed on the physical products themselves that were being produced, or purchased in the more generalized format, and then delivered to the customers. As such, our database solely encompasses client, order, and employee information.

1.1.4 Entity and Relation Set Definitions

Below is a full listing of the entities and relations used in our database, as well as

a brief description regarding their purposes. Section 1.2.1 will feature a more nuanced depiction of each entity and attribute, with the below listing serving as a more itemized format.

Entity Type Definitions:

Department: Department_ID, Name, Description, Location

Department_ID - Department identifier

Name - Department name

Location - Department location

Description - Brief summary of the department

Example - The sales department oversees the delivery of the products

Employee: Employee_ID, Name, SSN, DOB, Address, Salary, Phone

Employee_ID - Employee identifier

Name - Name of the employee

SSN - Employee's social security number for payroll purposes

DOB - Employee's date of birth

Address - Employee's address

Salary - How much the employee earns

Phone - Employee's contact number

Example - One of the employees delivers products to the clients

Routes: PathID, Path_Name, Start_Time, Vehicle_Num, First_Address,

Next_Address Scheduled_Stops

PathID - Path Identifier

Path_Name - Brief overview of the route

Start_Time - Time when employee started on the route

Vehicle_Num - Identifier for vehicle in use on the route

First_Address - Identifier pointing to an address in locations entity

Next_Address - Identifier pointing to the next address in locations

Scheduled_Stops - Number of clients on the route

Example - One route tracks a number of clients and the vehicle assigned to it

Locations: Loc_ID, State, City_Name, Zip_Code, Street_Address

Loc_ID - Location identifier

State - State where the address is located

City_Name - City where the address is located

Zip_Code - Zip code for the address

Street_Address - Street and number for the address

Example - A location is anywhere the client has designated within their contract for their order to be delivered

Clients: Client_ID, Name, Address, Phone, DOB, Credit_Card, E-mail

Client_ID - Client identifier

Name - Client's name

Address - Client's Address

Phone - Client's phone number

E-mail - Client's email address

DOB - Client's date of birth

Credit_Card - Client's payment information

Example - One of the clients has had past orders from the company

Contracts: Contract_ID, Frequency, Start_Date, End_Date

Contract_ID - Contract identifier

Frequency - How often the order is to be delivered

Start_Date - Date the contract started

End_Date - Date the contract ends

Example - A customer created a contract with the company

Product: Product_ID, Product_Name, Sale_Price, Purchase_Price

Product_ID - Product identifier

Product_Name - Name of the product

Sale_Price - Selling price of the product

Purchase_Price - Cost to obtain the product

Example - All of the items we sell are considered a product

Warehouse: Ware_ID, Location

Ware_ID - Identifier for the warehouse

Location - Address data for the warehouse

Example - The warehouses store the products

Supplier: S_Name, S_ID

S_Name - Name of the supplier

S_ID - Supplier identifier

Example - Our company purchases products from the supplier

Purchase_Order: Order_ID, Date_Submitted, Date_Fulfilled

Order_ID - Identifier for the order

Date_Submitted - Date the order was placed

Date_Fulfilled - Date the order was delivered to the warehouse

Example - The manager placed a purchase order with the supplier

Relationship Type Definitions:

Works_for: Department (M) to Employee (N)

Start_Date - Date the employee began working for the department

End_Date - Date the employee stopped working for the department

An unspecified number of employees can work for any one department.

Manages: Department (M) to Employee (N)

Start_Date - Date the employee began managing a department

End_Date - Date the employee stopped managing a department

One employee can manage one department.

Supervises: Employee (M) to Employee (N)

Start_Date - Date the employee began supervising other employees

End_Date - Date the employee stopped supervising other employees

An unspecified number of employees can supervise other employees.

Drives: Employee (1) to Route (N)

One employee can work on any number of routes.

Leads_to: Routes (M) to Locations (N)

Any number of clients can be on any number of routes. This allows for the handling of multiple orders and so forth.

Provisioned: Contracts (M) to Locations (N)

Any number of contracts can refer to any number of locations. This allows

for clients to place orders for different locations.

Contains: Contracts (M) to Products (N)

Quantity - Number of each type of product ordered

Any number of contracts can contain any number of products.

Sign_For: Clients (1) to Contracts (M)

A client creates a contract by signing for it.

Stores: Products (M) to Warehouse (N)

Quantity - the number of each product stored

Any number of warehouses store any number of products.

Supplies: Supplier (N) to Warehouse (M) and to Products (M)

Quantity - the number of each product supplied

Any number of suppliers can supply any number of products to any number of the warehouses.

Distributed: Products (N) to Department (M)

Any number of departments can handle the distribution of any number of products to the delivery personnel.

Creates: Employee (N) to Purchase_Order (M)

Any number of employees can place any number of purchase orders.

Fills: Purchase_Order (N) to Supplier (1)

One supplier fills any amount of purchase orders sent to them.

1.1.5 User Groups, Data Views, and Operations

The database will include four user groups. For more of the front-end side, we

would have the customers; they would be able to view their transaction history, add and remove orders, view favorited orders, and receive exclusive membership deals. The delivery personnel would be able to view the completed orders of the customers that are assigned to their route; the operations of the delivery personnel would include checking off attended customers, inputting the amount of products received, reviewing the condition of the operating vehicle and updating their route manifest. The managerial group have the authority to view the routes of all the delivery personnel as well as the incoming products. Managers are able to manipulate routes, approve of the supplied products, review the condition of all operable delivery vehicles and send orders to the supplier. The supplier have access to the orders that are being requested; suppliers would deliver the products to the warehouse.

1.2 Conceptual Database Design

With the generic definitions established, we will now seek to create more of a definitive split between entities, specifically in more solidly defined attributes and more explicit relationships. Also covered in this section will be a limited portion discussing the specializations and generalizations that will be encompassed within our database design. Lastly, a visual depiction of the structure of the database is included in the form of an E-R Model.

1.2.1 Entity Types

Department: Department_ID, Description, Name, Location

The Department entity will oversee the distribution and

management of products. Each department will have their own ID in order to differentiate between them and to better coordinate information and products. Each department will be managed by one employee but will be able to hire multiple employees. The employees will work for a single department only. The departments will manage the products once they are distributed. The Department entity would need to be updated when a manager is hired or fired but will not be updated regularly. It would also need to be updated if the department decides to change their role.

Candidate Keys: Department_ID, Name

Primary Key: Department_ID

Entity Type: Strong

Fields to be Indexed: Department_ID, Name

Department Attributes:

Attribute name:	Name	Department_ID	Description	Location
Description	Name of departments	Each department will have a different identifier for the system to	A description of each department purely for	Street Address, City, State, Zip code

		differentiate.	basic knowledge of the workplace.	
Domain/Type	String	Integer	String	String, String, String, Integer
Value - range	Any	000001-999999 (Any 6 digits)	Any	Any, Any, Any, 00000-99999
Default Value	None	000000	None	Null
NULL Value Allowed	No	No	Yes	Yes
Unique	No	Yes	No	No
Single or Multi-value	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Composite

Employee: Name, Employee_ID, SSN, DOB, Address, Salary, Phone

The entity type employee would be the workforce of the company; each employee would work for a single department. In our focus of the company most employees would be delivery men and would drive the routes they have been assigned; an employee may also be a manager to a department as well and would manage the department, however, these

two employee positions are mutually exclusive (disjoint specialization constraint). If an employee would be absent and if they are a delivery driver, then the routes would be updated to accompany this change.

Candidate Keys: Name, Employee_ID, SSN

Primary Key: Employee_ID

Entity Type: Strong

Fields to be Indexed: Name, Employee_ID, SSN

Employee Attributes:

Attribute Name	Name	Employee_ID	SSN	Phone
Description	Employee name consists of: First name, Middle initial, Last name	Every employee is assigned an identification number	Every employee possesses a Social Security Number	Clients can have a single phone number registered (US numbers only)
Domain / Type	String	Integer	Integer	Integer

Value-Range	Any	000001-999999 (Any 6 digits)	000000001-999999999 (Any 9 digits)	0000000001-9999999999 (Any 10 digits)
Default Value	John Q Public	000000	000000	0000000000
NULL Value Allowed	No	No	No	Yes
Unique	No	Yes	Yes	No
Single or Multi-value	Single	Single	Single	Single
Simple or Composite	Composite	Simple	Simple	Simple

Employee Attributes (continued):

Attribute Name	DOB	Address	Salary
Description	Year employee was born: MM/DD/YYYY	Street Address, City, State, Zip code	Annual income of employee in \$
Domain / Type	Date	String, String, String, Integer	Integer
Value-Range	Date	Any, Any, Any, 00000-99999	000001-999999 (Any 6 digits)
Default Value	Null	Null	Null
NULL Value Allowed	Yes	Yes	Yes
Unique	No	No	No

Single or Multi-value	Single	Single	Single
Simple or Composite	Simple	Composite	Simple

Products: Product_ID, Product_Name, Sale_Price, Purchase_Price

The product is an entity that has been ordered by a customer where it originates from the supplier and is distributed to a department. A product would have an ID and distinguishable name, it would also contain nutrition information, the amount to purchase off from the supplier and the price at which the product is being sold. New products may become available at any given moment, and sale prices would be frequently updated for seasonal products.

Candidate Keys: Product_ID, Product_Name

Primary Key: Product_ID

Entity Type: Weak

Fields to be Indexed: Product_ID, Product_Name, Sale_Price

Product Attributes:

Attribute Name	Product_ID	Product_Name	Sale_Price	Purchase_Price
Description	Every product is assigned an identification number	Every product possesses a name	Price which product is sold to customers in \$	Price which product is bought from supplier in \$
Domain / Type	Integer	String	Numeric	Numeric
Value-Range	0000001-999999 (Any 6 digits)	Any	001.00-999.99 (Any 3 digits)	001.00-999.99 (Any 3 digits)
Default Value	000000	None	000	000
NULL Value Allowed	No	No	No	No
Unique	Yes	No	No	No
Single or	Single	Single	Single	Single

Multi-value				
Simple or Composite	Simple	Simple	Simple	Simple

Supplier: S_Name, S_ID

The suppliers are a listing of all of the partners contracted by the business for the purpose of obtaining the various products. Additionally, the suppliers allow for redundancy, in that it is possible to obtain the same or similar products from more than one supplier.

Candidate Keys: S_Name, S_ID

Primary Key: S_ID

Entity Type: Strong

Fields to be Indexed: S_Name, S_ID

Supplier Attributes:

Attribute Name	S_Name	S_ID
Description	Name of the supplier	Suppliers identification number
Domain / Type	String	Integer
Value-Range	Any	000001-999999

		(Any 6 digits)
Default Value	Null	000000
NULL Value Allowed	Yes	No
Unique	Yes	Yes
Single or Multi-value	Single	Single
Simple or Composite	Simple	Simple

Warehouse: Ware_ID, Location

The warehouse stores all products that have been listed in the purchase order which has been placed by the department manager. The supplier supplies the products to the warehouse. Warehouses are rarely added, and they will almost never be updated or removed.

Candidate Keys: Ware_ID, Address

Primary Key: Ware_ID

Entity Type: Strong

Fields to be Indexed: Ware_ID, Location

Supplier Attributes:

Attribute Name	Ware_ID	Location
Description	Warehouse identification number	Street Address, Longitude, Latitude
Domain / Type	Integer	String, String, String, Integer
Value-Range	000001-999999 (Any 6 digits)	Any, Any, Any, 00000-99999

Default Value	000000	Null
NULL Value Allowed	No	No
Unique	Yes	Yes
Single or Multi-value	Single	Single
Simple or Composite	Simple	Composite

Routes: Path_ID, Path_Name, Start_Time, Vehicle_Num, First_Address,
Next_Address, Scheduled_Stops

The routes entity exist as a sequence of locations where the employee must drive to reach the customers residency. Each route has its own identification number along with provided vehicle with its own identification number. The first address will be the starting address of the route, if that address doesn't exist or once the address has been delivered to, the next address will be used for the next delivery and so on. Routes are regularly updated if an order is placed or canceled; it is rare that a route is removed since the route does not depend on a single destination but on cluster of destinations.

Candidate Keys: Path_ID, Path_Name

Primary Key: Path_ID

Entity Type: Strong

Fields to be Indexed: Path_ID, Path_Name, Vehicle_Num

Routes Attributes:

Attribute Name	Path_ID	Path_Name	Start_Time	Vehicle_Num
Description	Every route has an identification number	Every route has a name	Time employee begins route in 24-hour format	Every vehicle has an identification number
Domain / Type	Integer	String	Integer	Integer
Value-Range	000001-999999 (Any 6 digits)	Any	0000-2359 (24hr Time)	000001-999999 (Any 6 digits)
Default Value	000000	Null	Null	000000
NULL Value Allowed	No	Yes	No	No
Unique	Yes	Yes	No	No
Single or Multi-value	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple

Routes Attributes (continued):

Attribute Name	First_Address	Next_Address	Scheduled_Stops
Description	The identification number for a home (foreign key for Loc_ID)	The identification number for the next home (foreign key for Loc_ID)	Number of destinations a route has
Domain / Type	Integer	Integer	Integer
Value-Range	000001-999999 (Any 6 digits)	000001-999999 (Any 6 digits)	01-99 (Any 2 digits)
Default Value	000000	000000	00
NULL Value Allowed	No	No	No
Unique	No	No	No
Single or Multi-value	Single	Single	Multi-value
Simple or Composite	Simple	Simple	Simple

Locations: Loc_ID, State, City_Name, Zip_Code, Street_Address

Locations are areas where the client has registered to their account to be a valid place for delivery. Locations are not necessarily the residency

of the client, as multiple clients can have the same location. It is uncommon for locations to be updated or removed.

Candidate Keys: Loc_ID, Street_Address

Primary Key: Loc_ID

Entity Type: Strong

Fields to be Indexed: Loc_ID, State, City_Name, Street_Address

Locations Attributes:

Attribute Name	Loc_ID	State	City_Name	Zip_Code
Description	The identification number for a home	State of where the location resides in	City of where the location resides in	Zip code of where the location resides in
Domain / Type	Integer	String	String	Integer
Value-Range	000001-999999 (Any 6 digits)	Any (Two letters)	Any	00000-99999 (Any 5 digits)
Default Value	000000	Null	Null	Null
NULL Value Allowed	No	Yes	Yes	No
Unique	No	No	No	No
Single or Multi-value	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple

Locations Attributes (continued):

Attribute Name	Street_Address
Description	Street Address Longitude, Latitude
Domain / Type	String, numeric, numeric
Value-Range	Any, -180.0000000 - 180.0000000 (-180 through 180), -90.0000000 - 90.0000000 (-90 through 90)
Default Value	Null
NULL Value Allowed	No
Unique	No
Single or Multi-value	Single
Simple or Composite	Composite

Clients: Client_ID, Name, Address, Phone, DOB, Credit_Card, E-mail

The client is someone who purchases at least one product; the product is then placed into an order. A clients information will be stored indefinitely unless they themselves have deleted their account, otherwise, it is uncommon for customers to update their information.

Candidate Keys: Client_ID, Name, Phone, E-mail

Primary Key: Client_ID

Entity Type: Strong

Fields to be Indexed: Order_ID, Name, Phone, Address, E-mail

Clients Attributes:

Attribute Name	Name	Client_ID	Phone	E-mail
Description	Client name consists of: First name, Middle initial, Last name	Every client is assigned an identification number	Clients can have a single phone number registered (US numbers only)	The primary way of notifying the customer of their order
Domain / Type	String	Integer	Integer	String

Value-Range	Any	0000000001-9999999999 (Any 10 digits)	0000000001-9999999999 (Any 10 digits)	Any
Default Value	John Q Public	0000000000	0000000000	johndoe@email.com
NULL Value Allowed	No	No	Yes	No
Unique	No	Yes	No	Yes
Single or Multi-value	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple

Clients Attributes (continued):

Attribute Name	DOB	Address	Credit_Card
Description	Year client was born: MM/DD/YYYY	Street Address, City, State, Zip code	16 digits of the credit card
Domain / Type	Date	String, String, String, Integer	Integer
Value-Range	Date	Any, Any, Any, 00000-99999 (Any 5 digits)	0000000000000000 00-9999999999999999 99 (Any 16 digits)
Default Value	Null	Null	0000000000000000 00

NULL Value Allowed	Yes	No	No
Unique	No	No	No
Single or Multi-value	Single	Single	Single
Simple or Composite	Simple	Composite	Simple

Contracts: Contract_ID, Frequency, Start_Date, End_Date

Contracts are created by clients to have an automatic specified order be delivered at a location periodically. Contracts are also utilized as a regular order, as in an order would be placed only a single time.

Contracts are frequently added though it is less common for contracts to be updated and even less so for a contract to be removed.

Candidate Keys: Contract_ID

Primary Key: Contract_ID

Entity Type: Weak

Fields to be Indexed: Contract_ID, Frequency, Address

Contract Attributes:

Attribute Name	Contract_ID	Frequency	Start_Date	End_Date
Description	Every contract has an identification number	How frequent an order should be delivered	Date when contract was created: MM/DD/YYYY	Date when contract will end: MM/DD/YYYY
Domain / Type	Integer	Integer	Date	Date
Value-Range	0000000001-9999999999 (Any 10 digits)	0 - 365 (days)	Date	Date
Default Value	0000000000	0	Null	Null
NULL Value Allowed	No	No	No	No

Unique	Yes	No	No	No
Single or Multi-value	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple

Purchase_Order: Order_ID, Date_Subitted, Date_Fulfilled

A manager would place an order for products to be supplied to the warehouse. Orders are frequently added, they are never updated and it's uncommon for orders to be removed.

Candidate Keys: Order_ID

Primary Key: Order_ID

Entity Type: Weak

Fields to be Indexed: Order_ID

Purchase_Order Attributes:

Attribute Name	Order_ID	Date_Submitted	Date_Fulfilled
Description	Every order has an identification number	Date when order was submitted: MM/DD/YYYY	Date when order has been delivered:

			MM/DD/YYYY
Domain / Type	Integer	Date	Date
Value-Range	0000000001-9999999999 (Any 10 digits)	Date	Date
Default Value	0000000000	Null	Null
NULL Value Allowed	No	No	No
Unique	Yes	No	No
Single or Multi-value	Single	Single	Single
Simple or Composite	Simple	Simple	Simple

1.2.2 Relationship Types

The relations detailed forthwith describe the associations between one or more entities. Another key aspect of the relations to be addressed involves any attributes tied to the relation between the two entities, as will be seen in the relation between the Orders and Products entity. Accompanying the enumeration of the relations will be a brief description of the purpose of said relation, the explicit entities involved, the cardinality of the relation, and the Participation constraints of the relation.

Relation: Works_For

Description:

Works for is used to establish the relation between the Department and Employee. In this instance, the employee works for the department, in a scenario where many employees can work for the department, but there

can only be one department per employee.

Entity Sets Involved:

Employee, Department

Cardinality:

M...N

Descriptive Fields:

Start_Date, End_Date

Participation Constraints:

Total participation is required for employees, as every employee works for a specific department. However, there is only partial participation for a department, as in the case of restructuring, a department need not always have employees in the interim.

Relation: Manages

Description:

Manages is used to detail a separate relation between an employee and a department. Specifically, the scenario where one employee manages one department.

Entity Sets Involved:

Employee, Department

Cardinality:

M...N

Descriptive Fields:

Start_Date, End_Date

Participation Constraints:

There is a total participation constraint on the department for this relation, as every department needs a manager, but only a partial participation constraint on the employee, as not every employee will be a manager of a department.

Relation: Supervises**Description:**

Supervises describes a link between the employee entity and itself to establish a workflow to oversee employees on more of a day-to-day occurrence, as would be necessary for businesses.

Entity Sets Involved:

Employee

Cardinality:

1...N

Descriptive Fields:

Start_Date, End_Date

Participation Constraints:

There would be a total participation constraint on the employees as every employee would either be supervising a subset of employees, or would be getting supervised by another employee.

Relation: Drives**Description:**

Drives details the relation between the Employee and a Route. More aptly, it entails which routes an employee may be assigned to for delivery purposes. Similar to the shipping business, one employee may be assigned to many routes, depending on the volume of any one route.

Entity Sets Involved:

Employee, Routes

Cardinality:

1...N

Descriptive Fields:

None

Participation Constraints:

There is a total participation constraint on the route in this relation, as every route must have a driver assigned to it for deliveries. Conversely, not every employee is necessarily a delivery driver, placing only a partial participation constraint on the employee entity in the relation.

Relation: Leads_To

Description:

Leads To entails the relation between the routes and the clients themselves, and more specifically how the routes will obtain the relevant addresses to generate the listings. As such, there will be multiple clients on multiple routes, as each client is not constrained to only one order per day and may be in several routes.

Entity Sets Involved:

Routes, Clients

Cardinality:

M...N

Descriptive Fields:

None

Participation Constraints:

Total Participation is required for the routes, as every route must lead to some client for the deliveries. Only partial participation is required for the client side of the relation, as not every client will always have an active order to be delivered.

Relation: Provisioned

Description:

Provisioned details which locations are associated with each contract, creating a mapping of locations to the contracts, and allowing for customers to place contracts for locations other than their personal address.

Entity Sets Involved:

Locations, Contracts

Cardinality:

M...N

Descriptive Fields:

None

Participation Constraints:

Total participation is required for the contracts, as every contract must have a location associated with it. Conversely, there is only a partial participation constraint for locations, as not all locations may have an active contract at any given time.

Relation: Sign_For

Description:

Sign_For represents the link between the Clients and the Contracts they have placed. For any one client, there could be multiple contracts.

Entity Sets Involved:

Clients, Contracts

Cardinality:

1...N

Descriptive Fields:

None

Participation Constraints:

Only partial participation is required for clients in this relation, as brand new clients will not have any past or ongoing contracts. Conversely, total participation is required for the contracts, as every contract must be associated to a client.

Relation: Contains

Description:

Contains establishes the link between the contract entity, used akin to an order history, and the products constituting the order, including the quantity of each item. All of the contracts will have a varying amount of products associated to them via the relation.

Entity Sets Involved:

Contracts, Products

Cardinality:

M...N

Descriptive Fields:

Quantity

Participation Constraints:

Every order is required to have a link to a product, for a total participation constraint. Not every product will be in every order, establishing only a partial participation constraint.

Relation: Stores

Description:

Stores entails the relation between products and the warehouse in which they are stored at. More specifically, it tracks the quantity of each product stored at each warehouse.

Entity Sets Involved:

Warehouse, Products

Cardinality:

M...N

Descriptive Fields:

Quantity

Participation Constraints:

Total participation is required for warehouse, as every warehouse must be storing at least one product. Products also has a total participation constraint, as every product must be stored at some warehouse.

Relation: Supplies

Description:

Supplies details the association which warehouse the suppliers have delivered to. A more accurate summation would be it creates the link with the products stored at the warehouses and the suppliers they were purchased from.

Entity Sets Involved:

Supplier, Warehouse, Products

Cardinality:

N...M...M

Descriptive Fields:

Quantity

Participation Constraints:

There is a total participation constraint on the suppliers, the warehouse, and the products in this relationship, as every supplier has to be linked to a warehouse it is supplying, every warehouse needs at least one supplier to be supplying it, and every product needs to have come from at one warehouse.

Relation: Distributes**Description:**

Distributes entails the process by which the products are separated to the routes they will be delivered on, handled by the department. As there will be several departments handling the ordering and inventory of the products, as well as the distribution of the products to the customers

Entity Sets Involved:

Products, Departments

Cardinality:

N...M

Descriptive Fields:

None

Participation Constraints:

Total participation is enforced for the products, as they all must be purchased and distributed by the departments. Only partial participation is required for the departments, as not every department will deal directly with the products, see HR departments.

Relation: Creates

Description:

Creates details the relationship of an employee creating a purchase order that is then to be sent to a supplier to filled.

Entity Sets Involved:

Employee, Purchase_Order

Cardinality:

N...M

Descriptive Fields:

None

Participation Constraints:

Total participation would be enforced for the Purchase_Order entity, as

there can be no purchase order that exists without having been created first. There is only a partial participation constraint enforced on employee, as not every employee will have the privileges to create a purchase order.

Relation: Fills

Description:

Fills describes the process of the supplier handling any purchase orders that have been created by the employees and sent to them.

Entity Sets Involved:

Purchase_Order, Department

Cardinality:

N...1

Descriptive Fields:

None

Participation Constraints:

There is a total participation constraint enforced on both the Purchase_Order and the Supplier. The reasoning being that every purchase order must be made for an explicit supplier, and if a supplier has never had a purchase order, they would not be considered a supplier, and be a waste to store in the database.

1.2.3 Related Entity Types

In enhanced entity relationship models, there exists specializations and generalizations. Specialization and generalization would involve defining a set of

subclasses (subtypes) which are members of an entity type that are identified from other entity types or subgroups. Specialization involves subclasses being derived from a single entity type; the entity type would be known as the superclass.

Generalization would be the reverse of specialization; common characteristics must be identified and entity types (subclasses) would be generalized into a single superclass. In our case, managers and delivery men work under the same department and both job positions share common attributes, so the EMPLOYEE entity type is the superclass that is created for these positions.

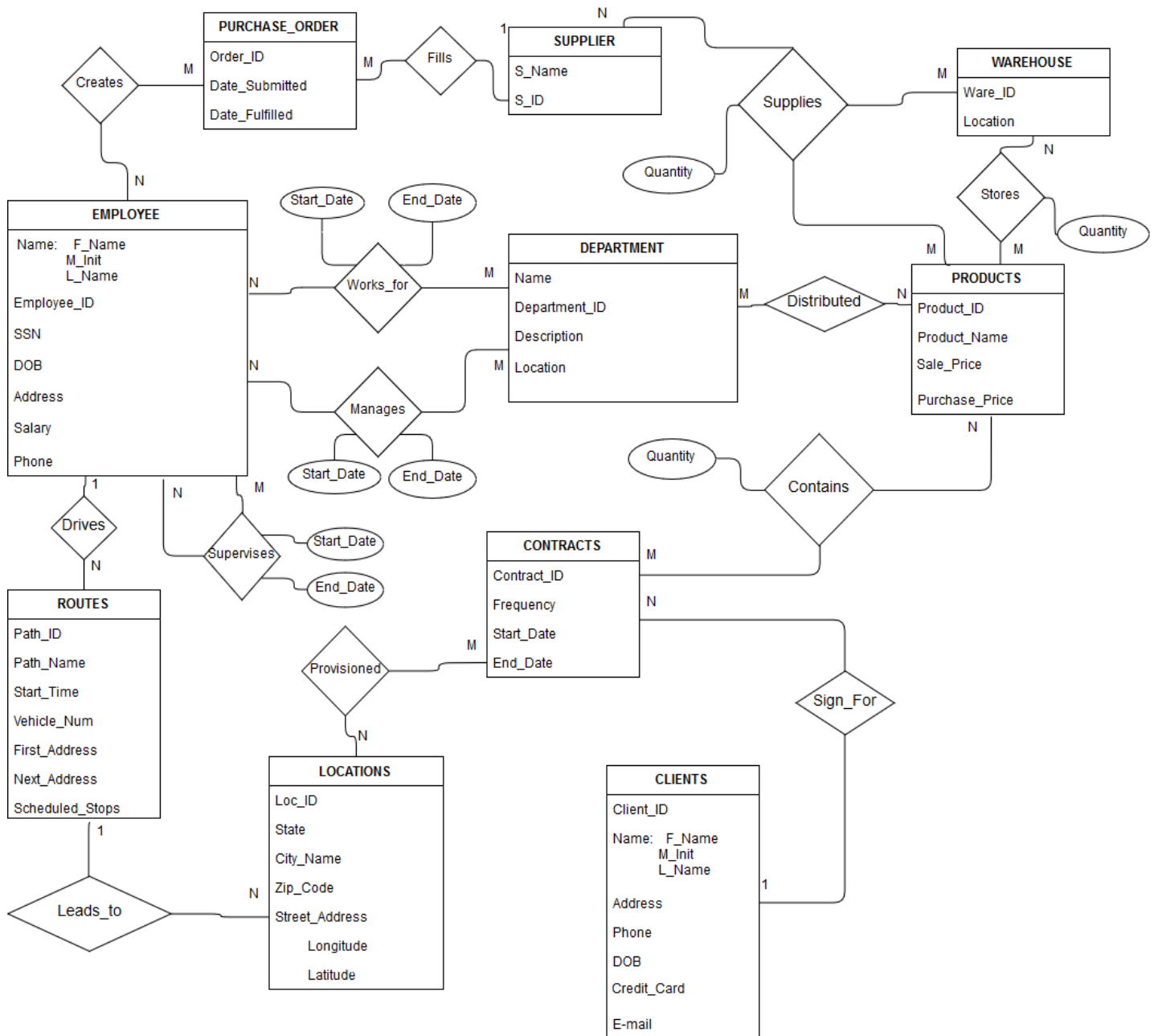
Two types of constraints exist for specializations and generalizations, they are named disjointness and totalness constraints. The disjointness constraint exists for subclasses that must be disjoint, where an entity can be a member for only one of the subclasses. One sample of disjoint within our design would be the distinction in an employee being classified as either a part of the delivery department, or the supply department, but not coexisting within both. The totalness constraint defines that every entity in the superclass must participate in at least one subclass of the specialization. Within the scope of our database, this totalness constraint can be seen in the employee entity as well, where an employee is capable of both supervising other employees as well as being in charge of a delivery route.

Lastly is a type of relationship, the concept of aggregation, or rather, the relationship between a whole object and its component parts. In the context of our database, aggregation shares a bit of overlap with the entities being classified as weak, or requiring another entity to fully define itself. Specifically, our orders entity outlined in the prior sections, would be a prime example of the aggregation of our database, as the

while the orders entity is meant to encompass the customers' orders, it alone does not store product information or the quantity of the order itself. However, all of this serves to merely outline some of the intricacies interplaying between our entities and their relations. Let's now turn to a more visual approach to our database.

1.2.4 E-R Diagram

While detailed descriptions of relations and entities help create a baseline on which to enumerate the dichotomy of the structure for the database, a more visual approach can help to picture the workflow itself. Specifically: shown below is an E-R Diagram modelling the entities and relations outlined in the prior sections.



Phase 2: Conceptual and Logical Database

Phase 1 was spent developing the nuance of the conceptual model of a

database in the form of the Entity-Relationship model. While the conceptual model serves its purpose in establishing a baseline for design, particularly in conferring with the individuals who might not be as familiar with some of the subtleties of the inner workings of a database, the conceptual falls short when it comes to implementation of design.

As such, the entirety of this phase will be spent analyzing and converting the conceptual model formed in Phase 1 into a Relational Model using the Relational model. After the conversion, sample data will be constructed to demonstrate all of the relations in the new model. Lastly, the largest benefit of the logical model will come in the last portion of the phase, wherein we will create sample queries to the database and translate them into three types of expressions: Relational Algebra, Tuple Relational Calculus, and Domain Relational Calculus.

2.1 E-R Model and Relational Model

While the conversion and associated benefits are the overarching goal of the phase, it's important to understand why the Relational Model is more beneficial from an implementation standpoint than the Entity-Relationship model. As such, this section will be discussing some of the history and purposes behind the two opposing models. Once that is established, the models will then be directly compared and contrasted to create a basic idea of pros and cons of each model.

2.1.1 Description of E-R Model and Relational Model

The Entity-Relationship Model and Diagram was designed by Peter Chen in 1976 for the conceptual design of database systems. This model describes data as entities, attributes, and relationships. The purpose of the E-R model is to provide a visual

representation of a system's data and to create the conceptual design of a database.

The Relational Model was first proposed in 1969 by Edgar F. Codd and is based on the mathematical concept of a relation. All data is represented in terms of tuples and grouped into relations. Unlike the E-R Model, relationship types are not represented in the Relational Model and steps are taken to convert them. The purpose of the Relational Model is to provide a method for specifying data and queries where the users would directly state the information the database has and what they want from it.

2.1.2 Comparison of Two Different Models

In comparing the Entity-Relationship Model with the Relational, the first thing to consider is the level at which the data is being analyzed. For the former, the data is looked at conceptually from a high level. In that sense, the Entity-Relationship model is a good starting point, particularly in its low barrier of entry for understanding. From that vantage point, it could be used to confer with clients about the design, given the more visual approach to representing the data. However, as a drawback due to its simplicity, the more conceptual approach of the Entity-Relationship does not translate well directly into a database implementation, hence the need of the Relational model..

In light of the restrictions of the Entity-Relationship model, the need for the Relational model is more apparent. Conversely to the Entity-Relational model, the Relational model is a more logical approach to representing the data. While the E-R model is able to represent composite attributes, which translates better into the real life scenario it's intended to represent, it does not translate into a database. Therefore the composite attributes are deconstructed during the conversion into the Relational model.

However, we have not yet addressed the main disparity between the two

different approaches—language. The Relational model's prime strength is the languages associated with it. Namely: Relational Algebra, Tuple Relational Calculus, and Domain Relational Calculus. These three languages allow for a much smoother transition into using a query language for the actual database, such as SQL.

2.2 Designing a Logical Database from a Conceptual

Because a Relational model is more suited towards a database we would need to understand the procedures in converting an Entity-Relationship model to a Relational model. The foundation of the relation schema must be created, which would be converting entity types into relations. As the Relational model only allows for single simple attributes, we would then detail how other types of attributes may be converted to conform to the schema.

2.2.1 Converting Entity Types to Relations

All entity types from the former E-R model would be designated as entity relations and follow the relation schema. The relation schema would consist of a relation name and the listing of attributes for the relation; the relation schema also allows the existence of simple single-value attributes. Entity types can either be strong or weak and would include single, multi-value, simple and composite attributes; all of these characteristics will be taken into consideration when converting an entity type into a relation.

Conversion of Strong and Weak Entity Types

Every strong entity type (E) will be converted to a relation (R); all of the

attributes of the former strong entity type will be retained in R. Any existing composite attributes of E would instead extract simple attributes of the composite attribute. Similar to the ER-Model, only a single key attribute is allowed to become the primary key, however, if the primary key is a composite attribute, then only the simple attributes of the composite attribute will be utilized. If E has more than one key, then the keys will become candidate keys and be stored for potential indexing.

Much of the process is similar for weak entity types (W). Every W will be converted into a relation (R); R will retain all simple attributes of the former entity type. The foreign key of W represents a relationship between W and the weak entity's owner (E); if W has an owner who's also a weak entity type then the weak entity type owner will be mapped first in order to obtain a primary key.

Simple and Composite Attributes

Simple attributes of an entity type will be converted to the relations schema and remain as a simple attribute. Composite attributes will be broken down into their simple attributes. If a primary key is a composite attribute then the broken down set of simple attributes would be designated as the primary key of that relation.

Single and Multi-Valued Attributes

Single attributes of an entity type will be converted to the relations schema and remain as a single attribute. For every multivalued attribute (A), one new relation (R) will be created for the purpose to house the attributes: A and the primary key attribute (K) where K is a foreign key that stands for the entity or relationship type that includes A. To elaborate on the newly created relations' primary key, as stated it is a foreign key first, however, the combination of the key and the multivalued attribute

($K + A$) results in the primary key for R.

2.2.2 Converting Relationship Types to Relations

Given the disparity between constraints, such as those of cardinality and participation of the Entity-Relationship model, there is a necessity for certain procedures on how to approach the conversion of an E-R model to a Relational model to still account for those constraints within the relation schemas. This section will detail some of the approaches necessary for representing the aspects of the E-R model within the scope of a Relational model.

❖ Relationship Types with a 1:1 Cardinality Constraint

➤ Foreign Key Method

- The foreign key approach entails taking the primary key from the first relation, and placing it into the second relation as a foreign key.
- However, as would be expected, each of the approaches have their own drawbacks. In the case of the foreign key method, there is the potential for wasting resources, except for the situation where one object has a total participation constraint. The reasoning is for any relation that does not participate, there would still need to be a stored value, null in that case.

➤ Merged Relation Method

- The merged relation method involves taking the attributes of both of the relations and creating a new relation that encompasses both of the original two.
- While the merged relation method can certainly be useful, there

generally shouldn't be a situation where it can be used, as it indicates a logical error in the conceptual phase, meaning the two entities should not have been separate entities to begin with.

➤ Cross Reference Method

- The cross reference method entails creating a new relation that contains the primary keys from both of the two relations as foreign keys. Including the foreign keys is the minimum requirement, as in the situation of the relationship having simple attributes, those attributes are also included in the new relation.
- The main usage for the cross reference method comes when neither of the entities being converted have a total participation constraint.

❖ **Relationship Types with a 1:N Cardinality Constraint**

➤ Foreign Key Method

- This shares several similarities with the same method for the 1:1 Cardinality Constraint. The differentiation comes in the primary key being included into the N side of the relation as a foreign key, as well as any simple attributes present in the relationship.

➤ Cross Reference Method

- The cross reference method detailed from the 1:1 cardinality constraints may be used again, assuming there is a low participation constraint on the N half of the relationship.

❖ **Relationship Types with an M:N Cardinality Constraint**

➤ Cross Reference Method

- The cross reference method is the only possible means to address a relationship type with a cardinality constraint of M:N. As addressed in the two previous sections, the primary key in the new relation would be a combination of the primary keys of the entities it's comprised of, as well as any simple attributes present in the original relationship.

◆ **Is-A and Has-A Relationships**

- This is also referred to as Super and Subclass Relationships
- Superclass and Subclass Relations
 - This approach converts the superclass and subclass entities into their own respective relations. However, all of the subclass entities all have the primary key of the superclass included as a foreign key to be used as the primary key for the subclass.
 - While effective, the approach has the drawbacks of dividing up the data across several more relations, which can make constructing queries more difficult.
- Only Subclass Relations
 - Creating only subclass relations involves including all of the superclass's attributes within the new subclass relation.
 - One limiting factor is the subclass must have a total participation constraint with the superclass, or the method cannot be implemented without losing data.

➤ Single Relation - Single Attribute Type

- The idea for this method involves merging all of the superclasses with the subclasses, and including the attributes of the subclasses.
- The prime drawback from this approach is that by necessity, there will be null entries for any of the attributes for subclasses that are disjoint. Stemming from this: the prime use of this method is best suited for situations where the subclasses feature a fair amount of overlap.

➤ Single Relation - Multiple Attribute Type

- This approach has a fair amount of overlap with the single attribute method in that it involves creating one relation that contains the union of the attributes from the super and subclasses, additionally pulling all of the attributes from the subclasses.
- Similarly to the prior method, this also will waste space with multiple NULL values, and should be used in scenarios with similar overlap in subclasses.

◆ **Recursive Relationships**

- When an entity has a relationship with itself, it can be represented either through the Foreign Key method, of adding an attribute of it's primary key within the same relation, or through the Cross Reference method, of creating a new relation with two foreign keys referencing the primary key of the entity with the recursive relationship.
- Similar to when these methods were discussed in other contexts, the

foreign key method makes for easier queries, but more inefficient use of space. Meanwhile, the cross-reference saves on space, but makes for more complex queries.

❖ **Relationships Between More Than Two Entities**

- If there is a situation when more than two entities participate in a relationship, during the conversion to a relational model, it is necessary to create a new relation, containing all of the primary keys from the participating entities, as well as any attributes present within the relationship itself.

❖ **Union Type Relationships**

- Union type relationships depict when a subclass belongs to multiple superclass entities. During the conversion, a new relation will be created containing the attributes of the subclass, as well as a new “surrogate key”, which is then retroactively inserted into the superclasses. The surrogate key then serves as the primary key within the relation created from the subclass.

2.2.3 Database Constraints

While the previous section detailed some of variety of ways to address the different types of relationships, this section will explore some of the constraints that a database needs to conform to. The constraints can result from several different reasons, whether they exist to ensure the integrity of the data, enforce rules for that specific business, or how to handle the data when changes are made. The following constraints will be detailed:

- Domain Constraints
- Entity Constraints
- Primary and Unique Key Constraints
- Referential Constraints
- Check Constraints and Business Rules

❖ **Domain Constraints**

- Domain constraints exist to verify if the values for a tuple fall within the domain specified for each attribute. More specifically, if the values match the data type for that attribute. It is also possible to further constrain the data type itself to a subset of that data type. Once the constraint is in place, the database management system will refuse to allow any updates or new values that do not exist within the domain of the attribute.

❖ **Entity Constraint**

- The entity constraint primarily relates to the primary key within all of the tuples. Specifically, it restricts the primary key from being set to 'null'. Primary keys are used as identifiers for the tuples, and as such the database management system will not allow updates or insertions that contain a null value for a primary key.

❖ **Primary and Unique Key Constraints**

- Similar to the former entity constraint, the constraints on primary and unique keys exist to protect the uniqueness of said keys. Primary keys by necessity must be unique, and as such, the database management system will reject any updates or insertions that could cause a conflict with a preexisting primary or unique key.

❖ **Referential Constraints**

- Referential constraints mainly involve foreign key attributes. If a tuple in one relation references a second tuple in another relation using a foreign key, this constraint requires that the second tuple must exist within that relation. The database management system again is the primary piece in enforcing this constraint. The most common form is the cascade operation, in that when a tuple is deleted, all references to the deleted tuple are also deleted. Similarly, if a new tuple is inserted with an invalid reference, the insert operation will be rejected by the DBMS.

❖ **Check Constraints and Business Rules**

- The aforementioned constraints were all focused on the integrity of the database from a data and logical perspective. Conversely, check constraints and business rules are designed to ensure the data functions in accordance with the business's expectations. As such, there are no explicit constraints to mention, as they are liable to change on a per database case.

2.3 Converting E-R Database into Relational

With the baseline dichotomy established between the Entity-Relationship Model and the Relational Model, as well as the methods and constraints to keep in mind when converting between the two opposing models, this section will feature a less abstract, and more direct approach to converting the model we described within Phase 1 into the Logical form of a Relational Model.

After detailing the particulars for the new relations, as well as some of the

constraints placed upon them, the remainder of this section will be spent depicting a variety of sample data, and how the relations depicted below would translate into storing the actual data, before the next section dives into how to actually parse through the relations and associated data with queries.

2.3.1 Relation Schema for Database

Now that we've established the methodology for maintaining the constraints in the transition between the logical form of the Entity-Relationship model and the Relational model, we will detail the specific relations included in our model, as well as the constraints placed on the attributes that are part of each of the following relations.

Employee (Employee_ID, F_Name, M_Init, L_Name, SSN, DOB, Address_ID, Salary, Phone)

→ Employee_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

→ F_Name

- ◆ Domain: String
- ◆ Range: varchar2(50)

→ M_Init

- ◆ Domain: String
- ◆ Range: varchar2(1)

→ L_Name

- ◆ Domain: String

- ◆ Range: varchar2(50)

➔ SSN

- ◆ Domain: Integer
- ◆ Range: 000000000-999999999
- ◆ Unique

➔ DOB

- ◆ Domain: Date

➔ Address_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key

➔ Candidate Keys

- ◆ Employee_ID
- ◆ SSN

➔ Primary Key

- ◆ {Employee_ID}

➔ Primary Key Constraints

- ◆ Employee_ID must be unique
- ◆ Employee_ID cannot be NULL

➔ Uniqueness Constraints

- ◆ SSN must be unique

➔ Business Constraints

- ◆ DOB must be longer than 18 years ago

Department (Department_ID, Name, Description, Address_ID)

→ Department_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

→ Name

- ◆ Domain: String
- ◆ Range: varchar2(50)

→ Description

- ◆ Domain: String
- ◆ Range: varchar2(255)

→ Address_ID

- ◆ Domain: Integer
- ◆ Range: 0 — MaxID
- ◆ Foreign Key

→ Candidate Keys

- ◆ Department_ID

→ Primary Key

- ◆ {Department_ID}

→ Primary Key Constraints

- ◆ Department_ID must be unique
- ◆ Department_ID cannot be NULL

→ Uniqueness Constraints

- ◆ None

→ Business Constraints

- ◆ None

Routes (Path_ID, Path_Name, Start_Time, Vehicle_Num, First_Address, Next_Address, Scheduled_Stops, Employee_ID)

➔ Path_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

➔ Path_Name

- ◆ Domain: String
- ◆ Range: varchar2(50)

➔ Start_Time

- ◆ Domain: Integer
- ◆ Range: 0000 — 2359 (24 hour time)

➔ Vehicle_Num

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

➔ First_Address

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key

➔ Next_Address

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key

➔ Scheduled_Stops

- ◆ Domain: Integer

- ◆ Range: 0 — 99 (Any 2 digits)

➔ Employee_ID

- ◆ Domain: Integer
- ◆ Range: 0 — MaxID
- ◆ Foreign Key

➔ Candidate Key

- ◆ Path_Name

➔ Primary Key

- ◆ {Path_ID}

➔ Primary Key Constraints

- ◆ Path_ID must be unique
- ◆ Path_ID cannot be NULL

➔ Uniqueness Constraints

- ◆ First_Address must be unique
- ◆ Next_Address must be unique

➔ Referential Integrity Constraints

- ◆ First_Address
- ◆ Next_Address

➔ Business Constraints

- ◆ Start_Time cannot be later than 1200

Locations (Loc_ID, State, City_Name, Zip_Code, Longitude, Latitude, Path_ID)

➔ Loc_ID

- ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
-

→ State

- ◆ Domain: String
- ◆ Range: varchar2(50)

→ City_Name

- ◆ Domain: String
- ◆ Range: varchar2(75)

→ Zip_Code

- ◆ Domain: Integer
- ◆ Range: 00000 — 99999

→ Longitude

- ◆ Domain: Numeric
- ◆ Range: -180.0000000 — 180.0000000

→ Latitude

- ◆ Domain: Numeric
- ◆ Range: -90.0000000 — 90.0000000

→ Path_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key

→ Candidate Keys

- ◆ Loc_ID
- ◆ {Longitude, Latitude}

→ Primary Key

- ◆ {Loc_ID}

→ Primary Key Constraints

- ◆ Loc_ID must be unique
 - ◆ Loc_ID cannot be NULL
 - ➔ Uniqueness Constraints
 - ◆ {Longitude, Latitude} must be unique
 - ➔ Business Constraints
 - ◆ Longitude, Latitude must fall within the U.S.
-

Contracts (Contract_ID, Frequency, Start_Date, End_Date, Client_ID)

- ➔ Contract_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ➔ Frequency
 - ◆ Domain: Integer
 - ◆ Range: 0 — 365
 - ➔ Start_Date
 - ◆ Domain: Date
 - ➔ End_Date
 - ◆ Domain: Date
 - ➔ Client_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Candidate Keys
 - ◆ Contract_ID
 - ➔ Primary Key
-

- ◆ {Contract_ID}

➔ Primary Key Constraints

- ◆ Contract_ID must be unique
- ◆ Contract_ID cannot be NULL

➔ Uniqueness Constraints

- ◆ None

➔ Business Constraints

- ◆ Frequency must be greater than 0

Clients (Client_ID, F_Name, M_Init, L_Name, Address_ID, Phone, DOB, Credit_Card, E-Mail)

➔ Client_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

➔ F_Name

- ◆ Domain: String
- ◆ Range: varchar2(50)

➔ M_Init

- ◆ Domain: String
- ◆ Range: varchar2(1)

➔ L_Name

- ◆ Domain: String
- ◆ Range: varchar2(50)

➔ Address_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

- ◆ Foreign Key
- ➔ Phone
 - ◆ Domain: Integer
 - ◆ Range: 000000000000 - 99999999999
- ➔ DOB
 - ◆ Domain: Date
- ➔ Credit_Card
 - ◆ Domain: Integer
 - ◆ Range: 0000000000000000-9999999999999999
- ➔ E-Mail
 - ◆ Domain: String
 - ◆ Range: varchar2(75)
- ➔ Candidate Keys
 - ◆ Client_ID
- ➔ Primary Key
 - ◆ {Client_ID}
- ➔ Primary Key Constraints
 - ◆ Client_ID must be unique
 - ◆ Client_ID cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ Address_ID
- ➔ Uniqueness Constraints
 - ◆ Address_ID must be unique
- ➔ Business Constraints
 - ◆ DOB must be longer than 18 years ago

- ◆ E-Mail must contain @*.*
- ◆ Phone_Num must be 11 digits long

Products (Product_ID, Product_Name, Sale_Price, Purchase_Price)

→ Product_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

→ Product_Name

- ◆ Domain: String
- ◆ Range: varchar2(50)

→ Sale_Price

- ◆ Domain: Numeric
- ◆ Range: 00000000.00 — 99999999.99

→ Purchase_Price

- ◆ Domain: Numeric
- ◆ Range: 00000000.00 — 99999999.99

→ Candidate Keys

- ◆ Product_ID

→ Primary Key

- ◆ {Product_ID}

→ Primary Key Constraints

- ◆ Product_ID must be unique
- ◆ Product_ID cannot be NULL

→ Uniqueness Constraints

- ◆ None

→ Business Constraints

- ◆ Sale_Price must be greater than 0
- ◆ Purchase_Price must be greater than 0

Warehouse (Ware_ID, Address_ID)

→ Ware_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)

→ Address_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key

→ Primary Key

- ◆ {Ware_ID}

→ Primary Key Constraints

- ◆ Ware_ID must be unique
- ◆ Ware_ID cannot be NULL

→ Referential Integrity Constraints

- ◆ Address_ID

→ Uniqueness Constraints

- ◆ Address_ID must be unique

→ Business Constraints

- ◆ None

Supplier (S_ID, S_Name)

→ S_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Primary Key

→ S_Name

- ◆ Domain: String
- ◆ Range: varchar2(75)

→ Primary Key

- ◆ {S_ID}

→ Primary Key Constraints

- ◆ S_ID must be unique
- ◆ S_ID cannot be NULL

→ Uniqueness Constraints

- ◆ None

→ Business Constraints

- ◆ None

Purchase_Order (Order_ID, Date_Submitted, Date_Fulfilled, S_ID)

→ Order_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Primary Key

→ Date_Submitted

- ◆ Domain: Date

→ Date_Fulfilled

- ◆ Domain: Date
 - ➔ S_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Primary Key
 - ◆ {Order_ID}
 - ➔ Primary Key Constraints
 - ◆ Order_ID must be unique
 - ◆ Order_ID cannot be NULL
 - ➔ Referential Integrity Constraints
 - ◆ S_ID
 - ➔ Uniqueness Constraints
 - ◆ S_ID must be unique
 - ➔ Business Constraints
 - ◆ Date_Fulfilled must be after Date_Submitted
-

Works_for (Department_ID, Employee_ID, Start_Date, End_Date)

- ➔ Department_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Employee_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
-

- ◆ Foreign Key
- ➔ Start_Date
 - ◆ Domain: Date
- ➔ End_Date
 - ◆ Domain: Date
- ➔ Primary Key
 - ◆ {Department_ID, Employee_ID}
- ➔ Primary Key Constraints
 - ◆ {Department_ID, Employee_ID} must be unique
 - ◆ {Department_ID, Employee_ID} cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ Department_ID
 - ◆ Employee_ID
- ➔ Uniqueness Constraints
 - ◆ Employee_ID must be unique
- ➔ Business Constraints
 - ◆ End_Date must be after Start_Date or NULL

Manages (Department_ID, Employee_ID, Start_Date, End_Date)

- ➔ Department_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Employee_ID
 - ◆ Domain: Integer
-

- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key
- ➔ Start_Date
 - ◆ Domain: Date
- ➔ End_Date
 - ◆ Domain: Date
- ➔ Primary Key
 - ◆ {Employee_ID, Department_ID}
- ➔ Primary Key Constraints
 - ◆ {Employee_ID, Department_ID} must be unique
 - ◆ {Employee_ID, Department_ID} cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ Employee_ID
 - ◆ Department_ID
- ➔ Uniqueness Constraints
 - ◆ None
- ➔ Business Constraints
 - ◆ End_Date must fall after Start_Date or be NULL

Supervises (Super_Emp_ID, Employee_ID, Start_Date, End_Date)

- ➔ Super_Emp_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Employee_ID
-

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key
- ➔ Start_Date
 - ◆ Domain: Date
- ➔ End_Date
 - ◆ Domain: Date
- ➔ Primary Key
 - ◆ {Super_Emp_ID, Employee_ID}
- ➔ Primary Key Constraints
 - ◆ {Super_Emp_ID, Employee_ID} must be unique
 - ◆ {Super_Emp_ID, Employee_ID} cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ Super_Emp_ID
 - ◆ Employee_ID
- ➔ Uniqueness Constraints
 - ◆ Super_Emp_ID
- ➔ Business Constraints
 - ◆ Employee_ID cannot be the same as Super_Emp_ID
 - ◆ End_Date must fall after Start_Date or be NULL

Provisioned (Contract_ID, Loc_ID)

- ➔ Contract_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
-

- ◆ Foreign Key
- ➔ Loc_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
- ➔ Primary Key
 - ◆ {Contract_ID, Loc_ID}
- ➔ Primary Key Constraints
 - ◆ {Contract_ID, Loc_ID} must be unique
 - ◆ {Contract_ID, Loc_ID} cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ Contract_ID
 - ◆ Loc_ID
- ➔ Uniqueness Constraints
 - ◆ None
- ➔ Business Constraints
 - ◆ None

Contains (Contract_ID, Product_ID, Quantity)

- ➔ Contract_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Product_ID
 - ◆ Domain: Integer
-

- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key
- ➔ Quantity
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999
- ➔ Primary Key
 - ◆ {Contract_ID, Product_ID}
- ➔ Primary Key Constraints
 - ◆ {Contract_ID, Product_ID} must be unique
 - ◆ {Contract_ID, Product_ID} cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ Contract_ID
 - ◆ Product_ID
- ➔ Uniqueness Constraints
 - ◆ None
- ➔ Business Constraints
 - ◆ Quantity must be greater than 0

Distributed (Department_ID, Product_ID)

- ➔ Department_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Product_ID
 - ◆ Domain: Integer
-

- ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Primary Key
 - ◆ {Department_ID, Product_ID}
 - ➔ Primary Key Constraints
 - ◆ {Department_ID, Product_ID} must be unique
 - ◆ {Department_ID, Product_ID} cannot be NULL
 - ➔ Referential Integrity Constraints
 - ◆ Department_ID
 - ◆ Product_ID
 - ➔ Uniqueness Constraints
 - ◆ None
 - ➔ Business Constraints
 - ◆ None
-

Stores (Ware_ID, Product_ID, Quantity)

- ➔ Ware_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Product_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
 - ➔ Quantity
-

- ◆ Domain: Integer
- ◆ Range: 0 — 999999
- ➔ Primary Key
 - ◆ {Ware_ID, Product_ID}
- ➔ Primary Key Constraints
 - ◆ {Ware_ID, Product_ID} must be unique
 - ◆ {Ware_ID, Product_ID} cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ Ware_ID
 - ◆ Product_ID
- ➔ Uniqueness Constraints
 - ◆ None
- ➔ Business Constraints
 - ◆ None

Supplies (S_ID, Ware_ID, Product_ID, Quantity)

- ➔ S_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
- ➔ Ware_ID
 - ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
- ➔ Product_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Foreign Key

→ Quantity

- ◆ Domain: Integer
- ◆ Range: 0 — 999999

→ Primary Key

- ◆ {S_ID, Ware_ID, Product_ID}

→ Primary Key Constraints

- ◆ {S_ID, Ware_ID, Product_ID} must be unique
- ◆ {S_ID, Ware_ID, Product_ID} cannot be NULL

→ Referential Integrity Constraints

- ◆ S_ID
- ◆ Ware_ID
- ◆ Product_ID

→ Uniqueness Constraints

- ◆ None

→ Business Constraints

- ◆ Quantity must be greater than 0

Creates (Employee_ID, Order_ID)

→ Employee_ID

- ◆ Domain: Integer
 - ◆ Range: 0 — 999999 (Any 6 digits)
 - ◆ Foreign Key
-

➔ Order_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 9999999999 (Any 10 digits)
- ◆ Foreign Key

➔ Primary Key

- ◆ {Employee_ID, Order_ID}

➔ Primary Key Constraints

- ◆ {Employee_ID, Order_ID} must be unique
- ◆ {Employee_ID, Order_ID} cannot be NULL

➔ Referential Integrity Constraints

- ◆ Employee_ID
- ◆ Order_ID

➔ Uniqueness Constraints

- ◆ None

➔ Business Constraints

- ◆ None

Address (Address_ID, Street, City, State, Zip)

➔ Address_ID

- ◆ Domain: Integer
- ◆ Range: 0 — 999999 (Any 6 digits)
- ◆ Primary Key

➔ Street

- ◆ Domain: String
- ◆ Range: varchar2(50)

- ➔ State
 - ◆ Domain: String
 - ◆ Range: varchar2(50)
- ➔ City_Name
 - ◆ Domain: String
 - ◆ Range: varchar2(75)
- ➔ Zip_Code
 - ◆ Domain: Integer
 - ◆ Range: 00000 — 99999
- ➔ Primary Key
 - ◆ {Address_ID}
- ➔ Primary Key Constraints
 - ◆ {Address_ID} must be unique
 - ◆ {Address_ID} cannot be NULL
- ➔ Referential Integrity Constraints
 - ◆ None
- ➔ Uniqueness Constraints
 - ◆ None
- ➔ Business Constraints
 - ◆ None

2.3.2 Sample Data of Relation

Now that the conversion from the conceptual to the logical model has been fully detailed, this section will serve to enumerate more specific examples of how each relation might function, using hard data. To be safe, there will be a fair amount of tuples per relation and relation set included in the examples below.

Clients								
Client_ID	F_Name	M_Init	L_Name	Address_ID	Phone	DOB	Credit_Card	E-Mail
0000000001	Gustav	B	Ganelea	0000000036	724-188-2710	7/24/1959	4981937169983648	Stapie@Lenfle.edu
0000000002	Bonnie	G	Nathaniel	0000000037	109-525-8344	2/8/1993	4556716572149046	Manken@Nathaniel.edu
0000000003	Jasmin	Y	Donese	0000000038	838-664-8846	7/1/1969	4716059369083798	Daflen@Huneneo.net
0000000004	Nensen	T	Funuse	0000000039	278-345-5415	11/12/1969	4556786613061187	Tenonee@Honsie.com
0000000005	Conton	Q	Antonia	0000000040	216-249-3276	1/19/1970	4024007133247139	Herbert@Darsie.net
0000000006	Seymour	D	Monehe	0000000041	968-309-5421	5/8/1971	4532592267648140	Frodie@Thinden.com
0000000007	Benonee	F	Diblin	0000000042	680-191-7898	2/20/1992	4189886715543515	Hinewei@Nicholas.com
0000000008	Henilon	Y	Stedie	0000000043	469-855-2894	2/10/1996	4716235268530234	Oscar@Jeffrey.edu
0000000009	Lunahei	J	Mancen	0000000044	481-398-2138	10/31/1992	4929376077162190	Spendon@Karfle.net
0000000010	Shendin	A	Minhen	0000000045	312-151-6206	3/5/2001	4539355619459290	Dinome@Briesie.net
0000000011	Beverley	I	Cecilia	0000000046	188-189-5455	12/11/1980	4024007109050335	Harry@Minese.com
0000000012	Arin	G	Aron	0000000047	417-545-8663	7/17/1974	4024007181213470	Tispan@Nenamee.edu
0000000013	Cinosea	U	Oran	0000000048	429-723-1985	5/30/1993	4485208570821020	Danamei@Spashon.com
0000000014	Naneheo	O	Spendin	0000000049	993-759-2607	6/9/2001	4929486784662570	Tenoteo@Lonenee.edu
0000000015	John	P	Doe	0000000050	273-977-6529	10/6/1962	4539819351632109	John@Doe.net

Employee								
Employee_ID	F_Name	M_Init	L_Name	Address_ID	Phone	DOB	Salary	SSN
0000000001	John	M	Doe	0000000001	499-682-	1/29/2001	223283	310-35-7362

					9581			
0000000002	Detren	J	Nonodeo	0000000002	464-653-5216	2/15/1967	057176	324-28-6325
0000000003	Stolie	S	Curtis	0000000003	992-747-7279	7/14/1992	850267	219-19-0535
0000000004	Spondan	O	Oswald	0000000004	241-228-1786	5/1/1973	027181	041-78-2212
0000000005	Samuel	F	Nenelon	0000000005	793-203-2122	4/4/1991	100146	206-22-1009
0000000006	Spondin	U	Eron	0000000006	722-142-2228	5/18/1984	079888	517-44-7960
0000000007	Frewie	F	Aron	0000000007	306-516-3876	5/9/1987	126567	217-56-3375
0000000008	Cenoseo	E	Sineme	0000000008	829-640-3566	9/23/1983	073624	481-86-6431
0000000009	Binane	P	Oron	0000000009	180-630-4101	7/22/1988	092057	489-74-8719
0000000010	Tadren	T	Thonie	0000000010	749-625-4607	3/19/1991	028806	182-78-4361
0000000011	Tanalan	C	Maifan	0000000011	398-321-1521	9/11/1983	035380	619-32-0231
0000000012	Amanda	N	Marion	0000000012	681-518-1326	12/11/1991	115015	023-86-1995
0000000013	Oron	Q	Nichole	0000000013	603-176-3277	8/26/1968	127564	252-12-7388
0000000014	Roberta	K	Fralie	0000000014	952-525-7019	1/24/2000	031679	264-15-4502
0000000015	Spandan	H	Sonalan	0000000015	582-636-4768	6/11/1958	085571	411-17-6684
0000000016	Edmund	A	Nanilon	0000000016	361-391-9197	4/23/1997	090423	051-72-7948
0000000017	Nonolen	K	Freddie	0000000017	694-412-9268	6/3/1958	185908	227-75-1598
0000000018	Tenaye	D	Henile	0000000018	447-180-4921	8/12/1966	140749	506-13-7306
0000000019	Sincle	J	Cinekeo	0000000019	788-254-5823	1/2/1983	135117	253-75-4298

0000000020	Thetie	F	Lanlin	0000000020	443-306-8345	4/30/1967	045703	382-33-1747
0000000021	Nanelen	H	Orin	0000000021	280-351-1226	1/2/1983	090209	088-56-8748
0000000022	Shindan	Q	Stemie	0000000022	651-823-2731	7/7/1987	032636	258-36-6861
0000000023	Nanahei	M	Teflon	0000000023	765-745-4751	1/20/1982	042664	374-24-3586
0000000024	Spendan	E	Tonilen	0000000024	240-292-9588	8/10/1970	121732	574-09-3088
0000000025	Trendan	B	Sharon	0000000025	834-132-9383	12/3/1991	161018	228-88-3661
0000000026	Hunewee	T	Flindan	0000000026	216-129-3222	12/22/1992	125337	491-18-9722
0000000027	Brivie	V	Spendon	0000000027	274-353-8808	11/10/1967	067829	098-70-7953
0000000028	Isaac	U	Rodney	0000000028	877-180-9848	2/27/1986	113587	245-36-5284
0000000029	Baniwea	C	Calvin	0000000029	233-318-6403	6/14/1991	028029	473-05-7669
0000000030	Monele	I	Tardie	0000000030	187-214-8264	5/25/1976	054090	308-44-3885
0000000031	Sanelen	X	Daihin	0000000031	637-121-3707	9/8/1965	109669	516-43-5332
0000000032	Nanohe	O	Frocie	0000000032	976-670-1150	11/7/1993	044290	562-65-2471
0000000033	Sorgle	Y	Flinden	0000000033	743-581-8391	7/28/1962	085141	549-68-3320
0000000034	Hanenee	R	Tonsin	0000000034	280-137-2695	6/17/1995	113871	117-86-2311

0000000035	Nonelon	B	Stayie	0000000035	995-176-1553	3/31/1958	074305	411-12-7030
------------	---------	---	--------	------------	--------------	-----------	--------	-------------

Address				
Address_ID	Street	City	State	Zip
0000000001	670 Nonulen Crescent	Batavia	New Jersey	38186
0000000002	240 Orin Gardens	Breckenridge	Washington	56681
0000000003	326 Dorothy Close	Canton	Ohio	43152
0000000004	326 Chesie Way	Seaside	Delaware	38144
0000000005	314 Plendin Mead	Jacksonville	North Carolina	78676
0000000006	409 Franie Way	Temple	Indiana	55406
0000000007	554 Ginasea Rise	Pompano Beach	Kansas	19527
0000000008	240 Sinibe Rise	Orlando	Kansas	73886
0000000009	170 Arin Gardens	Aberdeen	Mississippi	56326
0000000010	976 Susan Place	Priest River	Utah	48956
0000000011	956 Stetie Hill	Lexington	Texas	90325
0000000012	559 Sinolea Avenue	Durant	Utah	75695
0000000013	862 Spondan Mead	Tallahassee	Montana	28364
0000000014	331 Elizabeth Gardens	Alcoa	Utah	82469
0000000015	284 Tetron Street	Michigan	Texas	96850
0000000016	715 Ceafle Rise	Oakland	Ohio	92005
0000000017	848 Arin Hill	Green River	Kansas	91612
0000000018	946 Conanee Vale	Petaluma	Montana	77391
0000000019	939 Stadie Place	Royal Oak	Illinois	50622
0000000020	180 Stogie Street	Altoona	Nebraska	59277
0000000021	128 Jeremiah Road	Lower Southampton	Oregon	74837
0000000022	540 Oren Place	Summit	California	22359
0000000023	508 Ninahea Grove	New Rochelle	Utah	44413
0000000024	297 Banelan Rise	Mansfield	Wisconsin	21688
0000000025	156 Feneye Way	Shreveport	Illinois	49117
0000000026	762 Taspan Grove	Mississippi	Mississippi	68163
0000000027	374 Wanameo Rise	Coupeville	Indiana	15839
0000000028	969 Stelie Vale	Michigan City	Oregon	26997

0000000029	650 Torsen Road	Sandwich	Louisiana	83406
0000000030	295 Arin Mews	Millville	California	82608
0000000031	892 Tanenei Mews	Winslow	North Carolina	69066
0000000032	508 Tanetea Rise	Muskogee	Virginia	13791
0000000033	381 Tochen Close	Portsmouth	Washington	43662
0000000034	670 Taspin Road	French Lick	Louisiana	43081
0000000035	645 Ellen Place	Hingham	Florida	85996
0000000036	441 Eugene Square	Norwood	Nebraska	15080
0000000037	755 Aron Street	Paterson	Nebraska	84155
0000000038	655 Tieben Gardens	Mount Vernon	West Virginia	92460
0000000039	325 Arin Way	Spring Green	Virginia	18399
0000000040	285 Dineme Lane	Guntersville	Louisiana	70063
0000000041	220 Doflon Hill	West Point	Wisconsin	93359
0000000042	698 Britie Square	Bismarck	Louisiana	44569
0000000043	953 Ceugon Road	Oxnard	Montana	93181
0000000044	827 Eran Drive	Middletown	Mississippi	49409
0000000045	379 Tispon Drive	Little Falls	South Carolina	81740
0000000046	842 Arin Place	Barnstable	Illinois	53093
0000000047	770 Teninee Mead	Lake Charles	Oregon	47430
0000000048	792 Chagie Place	Mesa	Virgin Island	92038
0000000049	885 Lanefe Vale	West Bridgewater	Virginia	48917
0000000050	637 Denelan Mead	Rhode Island	Arizona	41982
0000000051	5 Debs Park	San Diego	California	92160
0000000052	42 Johnson Plaza	San Antonio	Texas	78260
0000000053	435 Lighthouse Bay Pass	Detroit	Michigan	48275
0000000054	08387 Autumn Leaf Circle	Suffolk	Virginia	23436
0000000055	26 Holmberg Street	Fairbanks	Alaska	99790
0000000056	7434 Fuller Lane	Charlotte	North Carolina	28263
0000000057	4846 Iowa Center	San Francisco	California	94177
0000000058	45990 Fremont Plaza	Muskegon	Michigan	49444
0000000059	10 Maple Point	Lexington	Kentucky	40524
0000000060	03 6th Pass	Charleston	West Virginia	25336
0000000061	448 Pawling Lane	Salt Lake City	Utah	84145
0000000062	019 Weeping Birch Pass	Anchorage	Alaska	99599
0000000063	29 Judy Terrace	Peoria	Illinois	61656
0000000064	4996 East Point	Bradenton	Florida	34210
0000000065	8 Elgar Hill	Washington	District of Columbia	20010

Locations						
Loc_ID	City	State	Zip	Longitude	Latitude	Path_ID
000001	Batavia	New Jersey	38186	- 115.2669373787 5792	34.5925607908 1971	000001
000002	Breckenridge	Washington	56681	- 100.0618592537 5792	47.8094990468 44714	000002
000003	Canton	Ohio	43152	- 120.5403748787 5792	40.8761792373 71524	000003
000004	Seaside	Delaware	38144	- 108.1477967537 5792	36.2044310683 1824	000004
000005	Jacksonville	North Carolina	78676	- 97.90853894125 792	28.4156038505 01267	000005
000006	Temple	Indiana	55406	- 82.39584362875 792	27.0542783607 60208	000006
000007	Pompano Beach	Kansas	19527	- 98.74349987875 792	35.1693589299 3562	000007
000008	Orlando	Kansas	73886	- 76.46322644125 792	39.8718420456 1187	000007
000009	Aberdeen	Mississippi	56326	- 69.82748425375 792	44.1783004066 1964	000009
000010	Priest River	Utah	48956	- 84.54916394125 792	44.7428672554 5292	000010
000011	Lexington	Texas	90325	- 97.46908581625 792	47.5431974235 9321	000011
000012	Durant	Utah	75695	-	41.7016646941	000010

				100.4573670662 5792	61906	
000013	Tallahassee	Montana	28364	- 115.3108826912 5792	43.2892383742 2243	000012
000014	Alcoa	Utah	82469	- 116.9808045662 5792	33.0225238642 5122	000010
000015	Michigan	Texas	96850	- 123.6165467537 5792	47.0963393105 4157	000011
000016	Nashville	Tennessee	62701	- 86.78156921872 693	36.1484502177 5153	000014
000017	Atlanta	Georgia	14425	- 84.42126551875 668	33.7249977071 93125	000015
000018	Jackson	Mississippi	23421	- 90.15664996312 87	32.3244202854 39615	000001
000019	Amarillo	Oklahoma	13634	- 101.8231070704 0921	35.2288889179 2396	000007
000020	Santa Fe	New Mexico	16635	- 105.9234783966 8582	35.6633061390 6651	000009
000021	Los Angeles	California	90263	- 118.2189364154 6558	34.0303854467 52355	000011
000022	Las Vegas	Nevada	93725	- 115.1335350592 4533	36.1771994706 2528	000008
000023	Fort Worth	Texas	78402	- 97.36563825462 917	32.7576876274 2407	000003
000024	Houston	Texas	11418	- 95.44478767965 893	29.7803707475 52833	000002
000025	Alcoa	Utah	82469	- 113.5904944782 2667	37.0813993391 43324	000010
000026	Lincoln	Nebraska	90806	- 96.65168338866 75	40.7743842653 45326	000013

000027	Denver	Colorado	65211	- 104.9598083436 714	39.7386679645 85645	000006
000028	Sioux Falls	South Dakota	03054	- 96.67207732240 877	43.5540774371 8912	000004
000029	Des Moines	Iowa	96766	- 93.56470330993 852	41.6037322152 76025	000007
000030	Springfield	Illinois	27888	- 89.63511153499 803	39.7944779563 0532	000005
000031	Madison	Wisconsin	80024	- 89.39618297435 527	43.0736827995 0761	000011
000032	Columbia	Missouri	54902	- 92.37285545583 95	38.9242573411 9287	000011
000033	Detroit	Michigan	61354	- 83.08248022673 194	42.3581940479 3734	000011
000034	Cleveland	Ohio	17777	- 81.66030145534 046	41.4848501480 6882	000011
000035	Louisville	Kentucky	16433	- 85.71330345029 355	38.2057366939 1505	000011

Routes							
Path_ID	Path_Name	Start_Time	Vehicle_Num	First_Address	Next_Address	Scheduled_Stops	Employee_ID
000001	Pfeifer	0930	310440	000020	000032	17	0000000031
000002	Lackey	0805	479697	000011	000012	13	0000000023
000003	Rea	0910	196655	000008	000018	9	0000000035
000004	Chau	0820	568149	000034	000016	12	0000000004
000005	Rhoades	0900	247115	000023	000013	26	0000000025
000006	Temple	0850	078251	000032	000012	23	0000000021
000007	Lerma	0835	274910	000005	000003	16	0000000011
000008	Cota	0810	774514	000017	000004	19	0000000001
000009	Spear	0840	921683	000019	000014	27	0000000009
000010	Merchant	0825	627712	000034	000013	31	0000000027
000011	Tovar	0855	417352	000013	000032	15	0000000011
000012	Fenton	0905	710080	000012	000018	18	0000000022
000013	Conroy	0815	557523	000010	000019	8	0000000014
000014	Burgos	0925	536264	000035	000014	13	0000000028
000015	Dodd	0800	633751	000027	000010	21	0000000033

Department			
Name	Department_ID	Description	Address_ID
San Diego TS	000001	Tech Support	0000000051
San Antonio ACC	000002	Accounting	0000000052
Detroit ADMS	000003	Admissions	0000000053
Suffolk DEVL	000004	Development	0000000054
Fairbanks COMMS	000005	Communications	0000000055
Charlotte MAIL	000006	Mail Services	0000000056
San Francisco FRL	000007	Food Research Lab	0000000057
Muskegon OR	000007	Outreach	0000000058
Lexington HR	000009	Human Resources	0000000059
Charleston KT	000010	Knowledge Transfer	0000000060
Salt Lake City CD	000011	Consumer Design	0000000061
Anchorage MAIL	000012	Mail Services	0000000062
Peoria ADMS	000013	Admissions	0000000063
Bradenton ACC	000014	Accounting	0000000064
Washington TS	000015	Tech Support	0000000065

Contracts				
Contract_ID	Frequency	Start_Date	End_Date	Client_ID
0000000001	177	2/25/2016	3/11/2016	000001
0000000002	7	5/23/2017	5/23/2017	000008
0000000003	5	1/19/2018	5/2/2018	000014
0000000004	14	3/30/2018	1/22/2019	000010
0000000005	31	6/13/2018	6/25/2018	000005
0000000006	60	1/22/2019	3/14/2019	000012
0000000007	1	11/28/2018	12/12/2018	000007
0000000008	2	9/14/2018	2/21/2019	000002
0000000009	5	4/27/2018	7/2/2018	000009
0000000010	7	3/16/2016	9/12/2016	000004
0000000011	7	7/5/2017	11/29/2017	000011

0000000012	21	10/28/2016	3/14/2017	000006
0000000013	5	7/10/2018	9/3/2018	000013
0000000014	7	12/29/2016	8/25/2017	000003
0000000015	31	3/30/2018	4/6/2018	000015
0000000016	90	10/16/2018	9/15/2020	000002
0000000017	7	5/25/2016	9/14/2021	000007
0000000018	1	8/5/2017	9/4/2020	000011
0000000019	1	5/3/2019	6/6/2019	000014
0000000020	2	10/6/2018	12/13/2018	000009
0000000021	5	5/13/2019	9/16/2019	000005
0000000022	31	5/3/2017	2/28/2019	000007
0000000023	7	8/28/2018	3/30/2019	000013
0000000024	5	6/27/2017	12/6/2018	000015
0000000025	2	11/12/2018	2/28/2019	000001
0000000026	1	6/29/2017	3/23/2018	000006
0000000027	9	10/16/2017	6/6/2018	000004
0000000028	7	5/29/2017	3/13/2019	000012
0000000029	6	6/7/2017	4/4/2018	000003
0000000030	60	9/15/2017	10/4/2017	000011
0000000031	31	12/26/2018	3/10/2019	000015
0000000032	2	8/29/2017	5/10/2018	000008
0000000033	1	10/14/2017	12/11/2018	000010
0000000034	5	3/22/2018	1/9/2019	000002
0000000035	7	12/19/2017	6/9/2018	000003

Products			
Product_ID	Product_Name	Sale_Price	Purchase_Price
000001	Fried Rice	7.99	5.99
000002	Chicken Breast	19.99	14.99
000003	Bacon	17.99	12.49
000004	Pork Sausage	11.99	5.99
000005	Breaded Chicken Patty	11.99	5.99
000006	Beef Hot Dog	5.99	2.49
000007	Sirloin Steak	26.99	18.99

000008	Pulled Pork	13.99	7.99
000009	Honey Ham	44.99	28.99
000010	Pineapple Chunks	6.49	2.99
000011	Coconut Strips	8.99	4.49
000012	Strawberry Oatmeal	5.99	2.49
000013	Baby Carrots	4.99	1.99
000014	Green Peas	6.49	2.99
000015	Broccoli Florets	6.99	3.49
000016	Crispy Fish Strips	12.99	10.49
000017	Cod Fillet	17.99	12.99
000018	Breaded Haddock Sticks	12.99	9.49
000019	Macaroni and Cheese	3.99	1.49
000020	Green Peas	6.49	3.29
000021	Whole Blueberries	7.49	3.99
000022	Broccoli and Cauliflower	6.99	2.99
000023	Asparagus Spears	10.99	5.99
000024	Whole Strawberries	6.49	3.49
000025	Steak Fries	6.99	3.99
000026	Vegetable Fried Rice	5.99	2.49
000027	Homestyle Waffles	5.99	1.99
000028	Buttermilk Pancakes	6.99	2.49
000029	Mini Donuts	8.99	4.99
000030	Blueberry Scone Dough	14.99	9.49
000031	Lemon Meringue Pie	11.19	8.99
000032	Chocolate Brownie Bites	9.99	6.49
000033	Chocolate Crème Pie	10.99	8.49
000034	Monster Cookie Dough	10.99	7.99
000035	Frozen Pizza	10.99	7.99

Warehouse	
Ware_ID	Address_ID
000001	0000000066
000002	0000000067
000003	0000000068
000004	0000000069
000005	0000000070
000006	0000000071
000007	0000000072
000008	0000000073
000009	0000000074
000010	0000000075

Supplier	
S_ID	S_Name
000001	Lopez Foods
000002	Keystone Foods
000003	Gavina Gourmet
000004	Groupe Danone
000005	Brake Foods
000006	HKTDC
000007	Atlas Wholesale
000008	Smart Foods
000009	EVCO
000010	Symrise

000011	0000000076
000012	0000000077
000013	0000000078
000014	0000000079
000015	0000000080

000011	ADM
000012	MGP Ingredients
000013	AAK
000014	Food Lion
000015	100KM Foods

Purchase_Order			
Order_ID	Date_Submitted	Date_Fulfilled	S_ID
0000000001	2/20/2015	3/10/2015	000005
0000000002	2/14/2019	3/4/2019	000015
0000000003	2/19/2016	3/9/2016	000011
0000000004	4/18/2018	4/24/2018	000009
0000000005	3/15/2017	4/7/2017	000012
0000000006	12/14/2015	12/22/2015	000007
0000000007	1/18/2017	2/9/2017	000013
0000000008	12/17/2018	2/11/2019	000008
0000000009	12/14/2015	12/22/2015	000004
0000000010	8/15/2017	9/8/2017	000010
0000000011	1/19/2015	1/26/2015	000003

0000000012	2/2/2018	2/27/2018	000001
0000000013	3/10/2016	6/24/2016	000006
0000000014	10/3/2018	10/9/2018	000014
0000000015	11/19/2015	12/8/2015	000002

Works_for			
Department_I D	Employee_I D	Start_Date	End_Date
000008	0000000001	6/27/1958	9/26/1959
000015	0000000030	10/1/1994	10/13/2018
000012	0000000005	11/9/1983	2/3/1986
000004	0000000024	10/3/1987	12/17/2011
000010	0000000012	1/30/1975	5/19/1983
000006	0000000006	6/30/1958	5/29/1959
000007	0000000008	8/24/1989	9/15/1989
000001	0000000014	4/9/1968	10/9/1970
000009	0000000009	8/2/1958	3/10/1960
000003	0000000023	8/6/1975	10/12/2010
000011	0000000003	8/8/1971	2/10/1974
000003	0000000002	4/3/1967	5/5/1967
000013	0000000013	8/31/1994	10/30/1994
000014	0000000007	11/21/1971	10/30/1972
000002	0000000011	11/25/1970	7/17/1971
000001	0000000035	1/2/1996	11/16/2000
000008	0000000017	7/14/1969	12/15/1972
000010	0000000033	8/27/2008	1/22/2019
000015	0000000004	11/27/1981	1/26/1983
000005	0000000016	12/12/1970	9/21/1974
000012	0000000019	4/3/1979	4/23/1981
000003	0000000007	7/4/1973	10/30/1978
000004	0000000018	3/26/1980	3/31/1980
000009	0000000001	2/2/1961	3/26/1961
000002	0000000031	6/3/1997	11/17/2018
000010	0000000020	11/24/2001	4/25/2003
000013	0000000015	11/4/1968	7/27/1969
000006	0000000017	5/29/1973	7/27/1974
000015	0000000022	7/30/1982	3/20/2002
000007	0000000032	6/24/1999	4/11/2018
000010	0000000009	3/6/1961	5/26/1962
000001	0000000019	7/1/1985	10/25/1989

Manages			
Department_I D	Employee_I D	Start_Date	End_Date
000007	0000000008	8/24/1989	9/15/1989
000013	0000000015	11/4/1968	7/27/1969
000008	0000000029	7/4/1990	10/13/2017
000015	0000000004	11/27/1981	1/26/1983
000015	0000000033	8/2/1999	3/20/2006
000009	0000000034	9/17/1997	3/21/2018
000012	0000000019	4/3/1979	4/23/1981
000008	0000000001	6/27/1958	9/26/1959
000009	0000000009	8/2/1958	3/10/1960
000011	0000000003	8/8/1971	2/10/1974
000002	0000000011	11/25/1970	7/17/1971
000012	0000000035	3/24/1998	3/15/2018
000013	0000000013	8/31/1994	10/30/1994
000001	0000000035	1/2/1996	11/16/2000
000010	0000000003	6/9/1967	5/31/1968
000010	0000000033	8/27/2008	1/22/2019
000010	0000000020	11/24/2001	4/25/2003
000001	0000000019	7/1/1985	10/25/1989
000001	0000000014	4/9/1968	10/9/1970
000006	0000000010	6/28/1980	2/23/1983
000010	0000000012	1/30/1975	5/19/1983
000006	0000000006	6/30/1958	5/29/1959
000015	0000000030	10/1/1994	10/13/2018
000011	0000000015	9/10/1969	7/8/1972
000014	0000000007	11/21/1971	10/30/1972
000006	0000000012	12/21/1988	9/14/1993
000008	0000000017	7/14/1969	12/15/1972
000014	0000000010	6/28/1984	7/14/1986
000005	0000000016	12/12/1970	9/21/1974
000003	0000000002	4/3/1967	5/5/1967
000004	0000000018	3/26/1980	3/31/1980
000014	0000000001	6/3/1961	3/18/1965

000005	0000000009	6/10/1962	6/2/1964
000009	0000000034	9/17/1997	3/21/2018
000012	0000000018	5/13/1980	4/7/1982
000003	0000000017	7/20/1976	11/22/1978
000006	0000000010	6/28/1980	2/23/1983
000012	0000000035	3/24/1998	3/15/2018
000014	0000000001	6/3/1961	3/18/1965
000011	0000000016	12/15/1976	5/20/1977
000008	0000000029	7/4/1990	10/13/2017
000013	0000000021	12/18/1978	3/10/1984
000006	0000000018	2/23/1987	5/26/1989
000014	0000000010	6/28/1984	7/14/1986
000011	0000000015	9/10/1969	7/8/1972
000002	0000000027	11/1/1994	1/21/2019
000004	0000000013	8/26/1999	1/7/2002
000008	0000000026	9/26/1991	7/27/2016
000010	0000000003	6/9/1967	5/31/1968
000015	0000000033	8/2/1999	3/20/2006
000001	0000000019	2/27/1991	4/2/1995
000012	0000000011	2/1/1973	1/15/1975
000007	0000000028	1/28/1989	9/3/2017
000001	0000000009	7/1/1964	10/5/1965
000013	0000000013	8/18/2007	12/7/2007
000004	0000000025	9/16/1991	2/14/2015
000011	0000000006	7/4/1968	11/24/1969
000002	0000000009	4/12/1966	12/2/1966
000006	0000000012	12/21/1988	9/14/1993
000015	0000000014	9/13/1974	5/15/1981

000009	0000000001	2/2/1961	3/26/1961
000006	0000000017	5/29/1973	7/27/1974
000012	0000000018	5/13/1980	4/7/1982
000015	0000000022	7/30/1982	3/20/2002
000012	0000000005	11/9/1983	2/3/1986
000010	0000000009	3/6/1961	5/26/1962
000013	0000000021	12/18/1978	3/10/1984
000006	0000000018	2/23/1987	5/26/1989
000003	0000000007	7/4/1973	10/30/1978
000012	0000000011	2/1/1973	1/15/1975
000003	0000000017	7/20/1976	11/22/1978
000002	0000000027	11/1/1994	1/21/2019
000005	0000000009	6/10/1962	6/2/1964
000004	0000000013	8/26/1999	1/7/2002
000004	0000000024	10/3/1987	12/17/2011
000002	0000000031	6/3/1997	11/17/2018
000001	0000000009	7/1/1964	10/5/1965
000015	0000000014	9/13/1974	5/15/1981
000003	0000000023	8/6/1975	10/12/2010
000002	0000000009	4/12/1966	12/2/1966
000011	0000000006	7/4/1968	11/24/1969
000008	0000000026	9/26/1991	7/27/2016
000001	0000000019	2/27/1991	4/2/1995
000007	0000000032	6/24/1999	4/11/2018
000004	0000000025	9/16/1991	2/14/2015
000007	0000000028	1/28/1989	9/3/2017
000011	0000000016	12/15/1976	5/20/1977
000013	0000000013	8/18/2007	12/7/2007

Supervises			
Super_Emp_ID	Employee_ID	Start_Date	End_Date
000014	0000000001	6/3/1961	3/18/1965
000009	0000000001	2/2/1961	3/26/1961
000006	0000000017	5/29/1973	7/27/1974
000012	0000000018	5/13/1980	4/7/1982
000015	0000000022	7/30/1982	3/20/2002
000012	0000000005	11/9/1983	2/3/1986

Provisioned	
Contract_ID	Loc_ID
0000000026	000006
0000000019	000013
0000000032	000014
0000000025	000003
0000000028	000015
0000000016	000009
0000000013	000005

000010	0000000009	3/6/1961	5/26/1962
000013	0000000021	12/18/1978	3/10/1984
000006	0000000018	2/23/1987	5/26/1989
000003	0000000007	7/4/1973	10/30/1978
000012	0000000011	2/1/1973	1/15/1975
000003	0000000017	7/20/1976	11/22/1978
000002	0000000027	11/1/1994	1/21/2019
000005	0000000009	6/10/1962	6/2/1964
000004	0000000013	8/26/1999	1/7/2002
000004	0000000024	10/3/1987	12/17/2011
000002	0000000031	6/3/1997	11/17/2018
000001	0000000009	7/1/1964	10/5/1965
000015	0000000014	9/13/1974	5/15/1981
000003	0000000023	8/6/1975	10/12/2010
000002	0000000009	4/12/1966	12/2/1966
000011	0000000006	7/4/1968	11/24/1969
000008	0000000026	9/26/1991	7/27/2016
000001	0000000019	2/27/1991	4/2/1995
000007	0000000032	6/24/1999	4/11/2018
000004	0000000025	9/16/1991	2/14/2015
000007	0000000028	1/28/1989	9/3/2017
000011	0000000016	12/15/1976	5/20/1977
000013	0000000013	8/18/2007	12/7/2007
000007	0000000008	8/24/1989	9/15/1989
000013	0000000015	11/4/1968	7/27/1969
000008	0000000029	7/4/1990	10/13/2017
000015	0000000004	11/27/1981	1/26/1983
000015	0000000033	8/2/1999	3/20/2006
000009	0000000034	9/17/1997	3/21/2018
000012	0000000019	4/3/1979	4/23/1981
000008	0000000001	6/27/1958	9/26/1959
000009	0000000009	8/2/1958	3/10/1960
000011	0000000003	8/8/1971	2/10/1974
000002	0000000011	11/25/1970	7/17/1971
000012	0000000035	3/24/1998	3/15/2018
000013	0000000013	8/31/1994	10/30/1994
000001	0000000020	1/2/1996	11/16/2000
000010	0000000002	6/9/1967	5/31/1968
000010	0000000033	8/27/2008	1/22/2019
000010	0000000020	11/24/2001	4/25/2003
000001	0000000019	7/1/1985	10/25/1989
000001	0000000014	4/9/1968	10/9/1970
000006	0000000010	6/28/1980	2/23/1983
000010	0000000012	1/30/1975	5/19/1983

0000000008	000008
0000000015	000010
0000000029	000011
0000000004	000002
0000000033	000012
0000000034	000004
0000000019	000001
0000000001	000007
0000000009	000035
0000000003	000017
0000000011	000018
0000000035	000022
0000000013	000005
0000000020	000009
0000000002	000021
0000000033	000018
0000000020	000007
0000000019	000011
0000000014	000017
0000000010	000027
0000000012	000009
0000000006	000013
0000000030	000024
0000000015	000031
0000000007	000009
0000000012	000014
0000000017	000023
0000000010	000009
0000000001	000006
0000000035	000035
0000000017	000019
0000000018	000017
0000000022	000035
0000000005	000013
0000000009	000020
0000000021	000002
0000000018	000033
0000000007	000020
0000000011	000019
0000000017	000014
0000000027	000010
0000000009	000012
0000000013	000006
0000000024	000030

000006	0000000006	6/30/1958	5/29/1959
000015	0000000030	10/1/1994	10/13/2018
000011	0000000015	9/10/1969	7/8/1972
000014	0000000007	11/21/1971	10/30/1972
000006	0000000012	12/21/1988	9/14/1993
000008	0000000017	7/14/1969	12/15/1972
000014	0000000010	6/28/1984	7/14/1986
000005	0000000016	12/12/1970	9/21/1974
000003	0000000002	4/3/1967	5/5/1967
000004	0000000018	3/26/1980	3/31/1980

0000000031	000015
0000000009	000007
0000000014	000012
0000000023	000017
0000000009	000010
0000000006	000001
0000000035	000035
0000000019	000013
0000000017	000020

Contains		
Contract_ID	Product_ID	Quantity
0000000008	000006	14
0000000015	000013	12
0000000029	000014	3
0000000004	000003	7
0000000033	000015	19
0000000034	000009	23
0000000019	000005	1
0000000001	000008	8
0000000009	000010	4
0000000003	000011	7
0000000011	000002	5
0000000035	000012	15
0000000013	000004	18
0000000020	000001	13
0000000002	000007	9
0000000033	000026	18
0000000020	000017	12
0000000019	000028	17
0000000014	000019	5
0000000010	000033	13
0000000012	000021	18
0000000006	000030	14
0000000030	000023	13
0000000015	000028	3
0000000007	000025	6

Distributed	
Department_ID	Product_ID
0000000001	000026
0000000013	000017
0000000003	000028
0000000014	000019
0000000005	000033
0000000006	000021
0000000014	000030
0000000008	000023
0000000012	000028
0000000010	000006
0000000011	000019
0000000002	000026
0000000013	000017
0000000012	000028
0000000014	000019
0000000005	000033
0000000010	000021
0000000011	000030
0000000015	000011
0000000005	000002
0000000006	000012
0000000001	000004
0000000013	000001
0000000014	000007
0000000010	000026

0000000012	000034	10
0000000017	000027	20
0000000010	000024	12
0000000016	000029	3
0000000002	000022	5
0000000018	000031	1
0000000001	000032	14
0000000001	000020	17
0000000017	000026	19
0000000018	000006	21
0000000022	000013	8
0000000005	000014	16
0000000009	000003	5
0000000021	000015	13
0000000018	000009	15
0000000007	000005	2
0000000011	000008	8
0000000017	000010	6
0000000027	000011	9
0000000009	000002	14
0000000013	000012	12
0000000024	000004	18
0000000031	000001	5
0000000009	000007	14
0000000014	000026	7
0000000023	000017	16
0000000009	000028	20
0000000006	000019	11
0000000026	000033	17
0000000019	000021	1
0000000032	000030	4
0000000025	000023	21
0000000028	000028	14
0000000016	000006	3
0000000013	000019	7

0000000002	000017
0000000008	000028
0000000012	000019
0000000010	000033
0000000011	000021
0000000002	000030
0000000013	000023
0000000012	000028
0000000009	000025
0000000008	000034
0000000013	000027
0000000003	000024
0000000010	000029
0000000004	000022
0000000005	000031
0000000015	000032
0000000006	000020
0000000011	000026
0000000001	000006
0000000014	000013
0000000013	000014
0000000007	000003
0000000015	000015
0000000005	000009
0000000006	000005
0000000001	000008
0000000013	000010
0000000005	000006
0000000013	000013
0000000005	000014
0000000012	000003
0000000009	000022
0000000008	000031
0000000013	000032
0000000003	000020

Stores		
Ware_ID	Product_ID	Quantity
000003	000006	14
000004	000013	12
000009	000014	3
000002	000003	7

Creates	
Employee_ID	Order_ID
0000000001	000006
0000000002	000003
0000000003	000004
0000000004	000008

000010	000015	19
000013	000009	23
000006	000005	1
000015	000008	8
000007	000010	4
000010	000011	7
000001	000002	5
000005	000012	15
000009	000004	18
000012	000001	13
000003	000007	9
000006	000026	18
000012	000017	12
000014	000028	17
000011	000019	5
000008	000033	13
000013	000021	18
000006	000030	14
000014	000023	13
000011	000028	3
000002	000025	6
000004	000034	10
000008	000027	20
000010	000024	12
000015	000029	3
000001	000022	5
000012	000031	1
000007	000032	14
000001	000020	17
000013	000026	19
000004	000006	21
000011	000013	8
000002	000014	16
000006	000003	5
000003	000015	13
000004	000009	15
000008	000005	2
000015	000008	8
000012	000010	6
000004	000011	9
000010	000002	14
000006	000012	12
000007	000004	18
000001	000001	5

0000000005	000015
0000000006	000012
0000000007	000004
0000000008	000010
0000000009	000006
0000000010	000007
0000000011	000001
0000000012	000009
0000000013	000003
0000000014	000011
0000000015	000003
0000000013	000013
0000000009	000014
0000000007	000002
0000000012	000001
0000000006	000008
0000000013	000010
0000000014	000015
0000000002	000005
0000000001	000006
0000000009	000015
0000000004	000007
0000000012	000010
0000000010	000001
0000000015	000005
0000000006	000009
0000000002	000012
0000000001	000003
0000000009	000006
0000000004	000012
0000000012	000014
0000000008	000011
0000000007	000008
0000000012	000003
0000000014	000004
0000000009	000009
0000000011	000002
0000000013	000010
0000000003	000013
0000000016	000006
0000000008	000003
0000000007	000004
0000000002	000009
0000000006	000006

000009	000007	14
000003	000026	7
000011	000017	16
000003	000028	20
000013	000019	11
000014	000033	17
000002	000021	1
000001	000030	4
000008	000023	21
000010	000028	14
000015	000006	3
000005	000019	7

0000000012	000007
0000000014	000001
0000000013	000009
0000000008	000003
0000000002	000011
0000000006	000003
0000000010	000013
0000000011	000014
0000000015	000002
0000000007	000001
0000000011	000008
0000000009	000006

Supplies			
S_ID	Ware_ID	Product_ID	Quantity
000003	000006	000009	182
000004	000013	000005	84
000009	000014	000008	112
000002	000003	000010	71
000010	000015	000011	38
000013	000009	000002	24
000006	000005	000012	73
000015	000008	000004	7
000007	000010	000001	46
000010	000011	000007	146
000001	000002	000026	93
000005	000012	000017	88
000009	000004	000028	181
000012	000001	000019	52
000003	000007	000033	54
000006	000003	000021	76
000012	000004	000008	49
000014	000009	000023	90
000011	000002	000028	96
000006	000010	000006	17

000012	000013	000019	43
000010	000006	000009	37
000003	000015	000005	80
000004	000007	000008	120
000009	000010	000010	41
000002	000001	000011	145
000010	000005	000006	152
000013	000009	000013	66
000006	000012	000014	40
000015	000003	000003	67
000007	000012	000015	177
000007	000014	000009	59
000001	000011	000005	79
000013	000008	000008	109
000004	000013	000010	194
000011	000006	000011	115
000002	000014	000002	197
000006	000011	000012	91
000003	000002	000004	100
000004	000004	000001	155
000008	000008	000007	108
000015	000010	000026	173
000012	000015	000017	99
000004	000001	000028	165
000010	000012	000019	51
000006	000007	000033	69
000013	000001	000021	186
000006	000013	000014	94
000015	000004	000003	25
000007	000011	000015	183
000010	000002	000009	75
000001	000006	000005	151
000005	000003	000014	141
000009	000004	000003	153
000012	000008	000015	14
000003	000015	000009	45
000006	000012	000005	8
000012	000015	000033	175
000014	000007	000021	36
000011	000010	000030	89

2.4 Sample Queries to Database

In the following sections, we will be discussing the sample queries that will be corresponding to our database system. These queries will be converted to relational algebra, tuple relational calculus, and domain relational calculus. A description of each of these languages will be in their respective sections along with the queries and their converted forms.

2.4.1 Design of Queries

The following is a list of the sample queries for our database system.

1. List all of the employees that have delivered to a location for every client.
 2. List every employee that has worked in every department.
 3. List all of the suppliers that have supplied “frozen pizza” to warehouse 50.
 4. List the customers with year long contracts of weekly deliveries.
 5. List the employees that have managed at least two departments.
 6. List the employees that have delivered to all the locations that employee John Doe has.
 7. List all of the suppliers that supply the product that has the highest sale price.
 8. List the product that makes the second most profit of those that are in an active contract.
 9. List the employees who have submitted purchase orders to every supplier.
 10. List the item that “John Doe” has ordered the least of, but has ordered at least once.
- Queries #1, 2, 9, and 10 are those that need to use the division operator.

2.4.2 Relational Algebra Expressions for Queries

Relational algebra is the basic set of operations for the relational model which allows a user to specify basic retrieval requests as relational algebra expressions. The results of the

operations create a new relation which may have been formed from one or more relations.

These relations can be further manipulated using operations of the same algebra. A sequence of these relations using relational algebra are then considered relational algebra expressions whose result will then be the relation that represents the result of the database query.

Relational algebra is important for different reasons. First, it provides a foundation for relational model operations. Second, it is used to implement and optimize queries for the relational database. Third, some of the concepts are used in the SQL for relational database management systems. Although most RDBMSs do not have user interfaces for relational algebra queries, the core operations and functions in the internal modules of most relational systems are based on relational algebra operations.

Below are the sample queries for our database and the relational algebra expression that corresponds to them.

1. List all of the employees that have delivered to a location for every client.

$$\Pi_{\text{Employee_ID}} (\sigma_{\text{Provisioned.Loc_ID} = \text{Routes.First_Address}} (\text{Routes} * \sigma_{\text{Provisioned.Contract_ID} = \text{Contract.Contract_ID}} (\text{Provisioned} * \sigma_{\text{Client.Client_ID} = \text{Contracts.Client_ID}} (\text{Clients} * \text{Contracts})))) \div \Pi_{\text{Client.Client_ID}} (\text{Clients}))$$

2. List every employee that has worked in every department.

$$\Pi_{\text{Works_for.Employee_ID}} (\text{Works_for} \div \Pi_{\text{Department.Department_ID}} (\text{Department}))$$

3. List all of the suppliers that have supplied “frozen pizza” to a warehouse.

$$\Pi_{\text{Supplies.S_ID}} (\sigma_{\text{Warehouse.Ware_ID} = \text{Supplies.Ware_ID}} (\text{Warehouse} * \sigma_{\text{Supplies.Product_ID} = \text{Products.Product_ID}} (\text{Supplies} * \sigma_{\text{Product.Name} = \text{“frozen pizza”}} (\text{Products}))))$$

4. List the customers with year long contracts of weekly deliveries.

$$\Pi_{\text{Contracts.Client_ID}} (\sigma_{\text{Contract.Frequency} = 7 \wedge \text{Difference}(\text{Start_Date}, \text{End_Date}) > 1 \text{ year}} (\text{Contracts}))$$

5. List the employees that have managed at least two departments.

$$\Pi_{w1.Employee_ID} (\sigma_{w1.Employee_ID = w2.Employee_ID \wedge w1.Department_ID \neq w2.Department_ID} (Manages(w1) * Manages(w2)))$$

- 6. List the employees that have delivered to all the locations that employee “John Doe” has.**

$$\Pi_{Employee.Employee_ID} (\sigma_{Employee.F_Name \neq "John" \wedge Employee.L_Name \neq "Doe"} ((Routes * Employee) \div \Pi_{Routes.First_Address} (\sigma_{(Routes.Employee_ID = Employee.Employee_ID) \wedge (Employee.F_Name = "John") \wedge (Employee.L_Name = "Doe")} (Routes * Employee))))$$

- 7. List all of the suppliers that supply the product that has the highest sale price.**

$$\Pi_{Supplies.S_ID} (Supplies \bowtie_{Supplies.Product_ID = P2.Product_ID} (\sigma_{P2.Sale_Price > P1.Sale_Price} (Products(P1) * Products(P2)))$$

- 8. List the product that has the second lowest sale price of those that are in an active contract.**

$$\Pi_{P2.Product_ID} (\sigma_{(P1.Sale_Price < P2.Sale_Price) \wedge (P2.Sale_Price < P3.Sale_Price) \wedge (end_date > CURRENT_DATE)} (Contracts * ((Products(P1) \bowtie_{P1.Product_ID = Contains.Product_ID} Contains) * (Products(P2) \bowtie_{P2.Product_ID = Contains.Product_ID} Contains) * (Products(P3) \bowtie_{P3.Product_ID = Contains.Product_ID} Contains))))$$

- 9. List the employees who have submitted purchase orders to every supplier.**

$$\Pi_{Employee_ID} ((\sigma_{(Employee.Employee_ID = Creates.Employee_ID) \wedge (Creates.Order_ID = Purchase_Order.Order_ID)} (Employee * Creates * Purchase_Order)) \div \Pi_{S_ID}(Supplier))$$

- 10. List the item that “John Doe” has ordered the least of, but has ordered at least once.**

$$\Pi_{C1.Product_ID} (\sigma_{C1.Quantity < C2.Quantity} ((Contains(C1) * Contains(C2)) \div \Pi_{Contract_ID} (\sigma_{Employee.F_Name = "John" \wedge Employee.L_Name = "Doe"} (Contracts * Contains))))$$

$\text{Contracts.Client_ID} = \text{Clients.Client_ID} (\text{Contracts} * (\sigma_{\text{Clients.F_Name} = \text{"John"} \wedge \text{Clients.L_Name} = \text{"Doe"}} (\text{Clients}))))))$

2.4.3 Relational Calculus for Queries

In general, relational calculus provides a higher-level declarative language for specifying relational queries. There are two variations of relational calculus: tuple relational calculus and domain relational calculus. In both variations of relational calculus, we write one declarative expression to specify a retrieval request. There is no description of how, or in what order, to evaluate a query. A relational calculus expression specifies what is to be retrieved rather than how to retrieve it. Relational calculus is considered to be a nonprocedural language. In section 2.4.4 we will discuss tuple relational calculus and in section 2.4.5 will discuss domain relational calculus.

2.4.4 Tuple Relational Calculus Expressions for Queries

Tuple relational calculus is based on specifying a number of tuple variables. A tuple needs to be defined and the relation name in which the tuple is to be searched for must be specified followed by a condition/formula. A formula is made up of predicate calculus atoms. Atoms can be in the form of $R(t_i)$ where R is a relation name and t_i is a tuple variable. Or in the form $t_i.A \text{ op } t_j.B$, where **op** is the comparison operators and A is an attribute of the relation on which t_i ranges and B is an attribute of the relation on which t_j ranges. Or, atoms can be in the form of $t_i.A \text{ op } c$ or $c \text{ op } t_i.A$ where c is a constant. Each tuple variable may take as its value any individual tuple from that relation. More than one tuple variable can be specified using existential and universal qualifiers. The following are the sample queries used in section 2.4.2 in their tuple relational calculus form.

1. List all of the employees that have delivered to a location for every client.

$$\{E \mid \text{Employee}(E) \wedge (\forall c)(\text{Clients}(c) \rightarrow (\exists o \exists r \exists p \exists t) (\text{Routes}(r) \wedge \text{Locations}(o) \wedge \text{Provisioned}(p) \wedge \text{Contracts}(t) \wedge t.\text{Client_ID} = c.\text{Client_ID} \wedge p.\text{Contract_ID} = t.\text{Contract_ID} \wedge p.\text{Loc_ID} = r.\text{First_Address} \wedge r.\text{Employee_ID} = E.\text{Employee_ID}))\}$$

2. List every employee that has worked in every department.

$\{e \mid \text{Employee}(e) \wedge (\forall d) (\text{Department}(d) \rightarrow (\exists w) (\text{Works_for}(w) \wedge w.\text{Employee_ID} = e.\text{Employee_ID} \wedge w.\text{Department_ID} = d.\text{Department_ID}))\}$

3. List all of the suppliers that have supplied “frozen pizza” to warehouse 50.

$\{s \mid \text{Suppliers}(s) \wedge (\exists w \exists p) (\text{Warehouse}(w) \wedge \text{Supplies}(s) \wedge \text{Products}(p) \wedge p.\text{Name} = \text{“frozen pizza”} \wedge p.\text{Product_ID} = s.\text{Product_ID} \wedge s.\text{Ware_ID} = w.\text{Ware_ID})\}$

4. List the customers with year long contracts of weekly deliveries.

$\{c \mid \text{Clients}(c) \wedge (\exists t) (\text{Contracts}(t) \wedge t.\text{Frequency} = 7 \wedge \text{Difference}(t.\text{Start_Date}, t.\text{End_Date}) > 1 \text{ year} \wedge t.\text{Client_ID} = c.\text{Client_ID})\}$

5. List the employees that have managed at least two departments.

$\{e \mid \text{Employee}(e) \wedge (\exists w1 \exists w2) (\text{Manages}(w1) \wedge \text{Manages}(w2) \wedge w1.\text{Employee_ID} = w2.\text{Employee_ID} \wedge w1.\text{Department_ID} \neq w2.\text{Department_ID} \wedge w1.\text{Employee_ID} = e.\text{Employee_ID})\}$

6. List the employees that have delivered to all the locations that employee John Doe has.

$\{e \mid \text{Employee}(e) \wedge \neg (\exists j \exists r1 \exists r2) (\text{Employee}(j) \wedge \text{Routes}(r1) \wedge \text{Routes}(r2) \wedge j.\text{F_Name} = \text{“John”} \wedge j.\text{L_Name} = \text{“Doe”} \wedge r1.\text{Employee_ID} = j.\text{Employee_ID} \wedge r2.\text{Employee_ID} = e.\text{Employee_ID} \wedge r1.\text{First_Address} \neq r2.\text{First_Address})\}$

7. List all of the suppliers that supply the product that has the highest sale price.

$\{s \mid \text{Suppliers}(s) \wedge (\exists u \exists p1 \exists p2) (\text{Supplies}(u) \wedge \text{Products}(p1) \wedge \text{Products}(p2) \wedge p2.\text{Sale_Price} > p1.\text{Sale_Price} \wedge u.\text{Product_ID} = p2.\text{Product_ID} \wedge u.\text{S_ID} = s.\text{S_ID})\}$

8. List the product that has the second lowest sale price of those that are in an active contract.

$\{p \mid \text{Products}(p) \wedge (\exists p1 \exists p3 \exists c1 \exists c2 \exists c3 \exists d) (\text{Products}(p1) \wedge \text{Products}(p3) \wedge \text{Contains}(c1) \wedge \text{Contains}(c2) \wedge \text{Contains}(c3) \wedge \text{Contracts}(d) \wedge d.\text{end_date} >$

$d.current_date \wedge p3.Product_ID = c3.Product_ID \wedge p.Product_ID = c2.Product_ID \wedge$
 $p1.Product_ID = c1.Product_ID \wedge p1.Sale_Price < p.Sale_Price \wedge p.Sale_Price <$
 $p3.Sale_Price\}}$

9. List the employees who have submitted purchase orders to every supplier.

$\{e \mid Employee(e) \wedge (\forall s) (Supplier(s) \rightarrow (\exists c \exists p) (Creates(c) \wedge Purchase_Order(p) \wedge$
 $e.Employee_ID = c.Employee_ID \wedge c.Order_ID = p.Order_ID \wedge p.S_ID = s.S_ID))\}$

10. List the item that “John Doe” has ordered the least of, but has ordered at least once.

$\{p \mid Product(p) \wedge (\exists c1 \exists c2 \exists t \exists s) (Contains(c1) \wedge Contains(c2) \wedge Contracts(t) \wedge$
 $Clients(s) \wedge s.F_Name = "John" \wedge s.L_Name = "Doe" \wedge s.Client_ID = t.Client_ID \wedge$
 $c1.Contract_ID = t.Contract_ID \wedge c2.Contract_ID = t.Contract_ID \wedge C1.Quantity <$
 $C2.Quantity \wedge C1.Product_ID = p.Product_ID)\}$

2.4.5 Domain Relational Calculus Expressions for Queries

Domain relational calculus is similar to tuple relational calculus but differs in the type of variables used in formulas. Instead of variables ranging over tuples, the variables range over single values from domains of attributes. The formulas are slightly different from tuple relational calculus as well. Atoms can be of the form $R(x_1, x_2, \dots, x_j)$ where R is the name of the relation of degree j and each $x_i, 1 \leq i \leq j$, is a domain variable. This atom states that a list of values $\langle x_1, x_2, \dots, x_j \rangle$ must be a tuple in the relation whose name is R , where x_i is the value of the i th attribute value of the tuple. An atom can also be in the form $x_i \text{ op } x_j$, where **op** is one of the comparison operators, and x_i and x_j are domain variables. An atom can also be in the form $x_i \text{ op } c$ or $c \text{ op } x_j$, where c is a constant value. Similar to tuple relational calculus, formulas are made up of atoms, variables, and quantifiers. The following are the sample queries used in section 2.4.2 and 2.4.4 in their domain relational calculus form.

1. List all of the employees that have delivered to a location for every client.

$$\{ \langle e \rangle \mid \text{Employee}(e, _, _, _, _, _, _, _, _) \wedge (\forall c) (\text{Clients}(c, _, _, _, _, _, _, _, _) \rightarrow (\exists i \exists t) (\text{Routes}(_, _, _, _, i, _, _, e) \wedge \text{Locations}(i, _, _, _, _, _, _, _) \wedge \text{Provisioned}(t, i) \wedge \text{Contracts}(t, _, _, _, c))) \}$$

2. List every employee that has worked in every department.

$$\{ \langle e \rangle \mid \text{Employee}(e, _, _, _, _, _, _, _, _) \wedge (\forall d) (\text{Department}(_, d, _, _) \rightarrow \text{Works_for}(d, e, _, _)) \}$$

3. List all of the suppliers that have supplied “frozen pizza” to warehouse 50.

$$\{ \langle s \rangle \mid \text{Suppliers}(s, _) \wedge (\exists w \exists p) (\text{Warehouse}(w, _) \wedge \text{Supplies}(s, w, p, _) \wedge \text{Products}(p, \text{“frozen pizza”}, _, _)) \}$$

4. List the customers with year long contracts of weekly deliveries.

$$\{ \langle c \rangle \mid \text{Clients}(c, _, _, _, _, _, _, _, _) \wedge (\exists x) (\text{Contracts}(_, >7, x, > x+1 \text{ year}, c)) \}$$

5. List the employees that have managed at least two departments.

$$\{ \langle e \rangle \mid \text{Employee}(e, _, _, _, _, _, _, _, _) \wedge (\exists d) (\text{Department}(_, d, _, _) \wedge \text{Manages}(d, e, _, _) \wedge \text{Manages}(\neg d, e, _, _)) \}$$

6. List the employees that have delivered to all the locations that employee John Doe has.

$$\{ \langle e \rangle \mid \text{Employee}(e, _, _, _, _, _, _, _, _) \wedge \neg (\exists j \exists r) (\text{Employee}(j, \text{“John”}, _, \text{“Doe”}, _, _, _, _, _) \wedge \text{Routes}(_, _, _, _, r, _, _, j) \wedge \text{Routes}(_, _, _, _, \neg r, _, _, e)) \}$$

7. List all of the suppliers that supply the product that has the highest sale price.

$$\{ \langle s \rangle \mid \text{Supplier}(s, _) \wedge (\exists u \exists p \exists r) (\text{Supplies}(s, _, p, _) \wedge \text{Products}(p, _, r, _) \wedge \text{Products}(\neg p, _, < r, _)) \}$$

8. List the product that has the second lowest sale price of those that are in an

active contract.

$$\{ \langle p, s \rangle \mid \text{Products}(p, _, s, _) \wedge (\exists p2 \exists p3 \exists c1 \exists c2 \exists c3 \exists d) (\text{Products}(p2, _, <s, _) \wedge \text{Products}(p3, _, >s, _) \wedge \text{Contains}(c1, p, _) \wedge \text{Contains}(c2, p2, _) \wedge \text{Contains}(c3, p3, _) \wedge \text{Contracts}(_, _, _, d, _)) \}$$

9. List the employees who have submitted purchase orders to every supplier.

$$\{ \langle e \rangle \mid \text{Employee}(e, _, _, _, _, _, _, _, _) \wedge (\forall s) (\text{Supplier}(s, _) \rightarrow (\exists c \exists o) (\text{Creates}(e, o) \wedge \text{Purchase_Order}(o, _, _, s))) \}$$

10. List the item that “John Doe” has ordered the least of, but has ordered at least once.

$$\{ \langle p \rangle \mid \text{Product}(p, _, _, _) \wedge (\exists s \exists t \exists q) (\text{Contains}(t, p, q) \wedge \text{Contains}(t, _, >q) \wedge \text{Contracts}(t, _, _, s) \wedge \text{Client}(s, \text{“John”}, _, \text{“Doe”}, _, _, _, _, _)) \}$$

Phase 3: Logical and Conceptual Database with Postgres DBMS

3.1 Normalization of Relations

The normalization of relations, as proposed by Codd in 1972, takes a relation schema through a series of tests to determine whether it satisfies a certain normal form. The process is through a top-down fashion and decomposes a relation as needed to satisfy a normal form. This method of decomposing relations is seen as a relational design by analysis. Throughout this section, we will discuss what normalization is along with the first, second, third, and Boyce-Codd normal forms are. We will also discuss the methods of decomposing a relation so that it may satisfy a normal form. Lastly, we will show our relations and which normal form they fall under.

3.1.1 Normalization of Relations

The normal form of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized. Normalization tests a relation schema to certify whether it satisfies a certain normal form. The normalization of data can be considered a process of analyzing the given relation schema based on their functional dependencies and primary keys to achieve the desirable properties of minimizing redundancy and minimizing the insertion, deletion, and update anomalies. Any conditions that do not pass the normal form tests are decomposed into smaller relation schemas that meet the tests.

First Normal Form

The first normal form states that the domain of an attribute must include only atomic values and that the value of any attribute in a tuple must be a single value from the domain of that attribute. The first normal form does not allow having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. If a relation has an attribute that is multivalued, composite, or a combination, then the relation does not pass and must be normalized through other means. One way would be to remove the attribute that violates the first normal form and place it in a separate relation along with the primary key of the first relation. This decomposes the non-1NF relation into two 1NF relations. Another is to expand the key so that there will be a separate tuple in the original relation. The primary key would then be a combination of the original primary key and the attribute that does not pass. However, this would introduce redundancy in the relation. The final way to solve the problem would be to replace the attribute with multiple atomic attributes. For example, an attribute called department_locations could be replaced by d_location1 and d_location2. The amount replacing the original attribute would depend on the maximum number of values that is known for the attribute. However, this would introduce more NULL values for the departments that have less locations than the maximum.

The first normal form also does not allow multivalued attributes that are themselves composite. These are called nested relations because each tuple can have a relation within it. To normalize these into 1NF, remove the nested relation attributes into a new relation and propagate the primary key into it. The primary key of the new relation will combine the partial key with the primary key of the original relation. This procedure can be done multiple times to a relation with multiple-level nesting to make a

set of 1NF relations.

Second Normal Form

A relation schema R is in the second normal form if every non-prime attribute A in R is fully functionally dependent on the primary key of R . A functional dependency $X \rightarrow Y$ is a full functional dependency if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y . A functional dependency $X \rightarrow Y$ is a partial dependency if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test does not need to be applied. If a relation schema is not in 2NF, it can be second normalized or 2NF normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent.

Third Normal Form

A relation schema R is in the third normal form if it satisfies 2NF and no non-prime attribute of R is transitively dependent on the primary key. The third normal form is based on the concept of transitive dependency. A functional dependency $X \rightarrow Y$ in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R , and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. If a relation is a transitive dependency, the way to normalize the relation would be to

decompose the relation into two 3NF relation schemas. For example, picture a relation that has two functional dependencies where FD1 holds an attribute of FD2. The first relation would hold FD1 while the second relation would hold FD2 and the common attribute would of would be the primary key of FD2. The relation would then be in the third normal form. A NATURAL JOIN operation on new relations would recover the original relation without generating spurious tuples.

Boyce-Codd Normal Form

The Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. A relational schema R is in BCNF if whenever a non-trivial functional dependency $X \rightarrow A$ holds in R , then X is a superkey of R . This definition of BCNF differs from the definition of 3NF in that condition of 3NF, which allows A to be prime, is absent from BCNF. An attribute of relation schema R is called a prime attribute of R if it is a member of some candidate key of R .

The Three Different Types of Update Anomalies

Normalization exists to remove redundant data and allows the database to be consistent. If the relations aren't normalized then anomalies will appear which affect the

relation when changes are made. Anomalies can only appear if two relations are joined together, such as the natural join operation.

Insertion Anomalies

Insertion anomalies occur whenever a record is created for a relation that is joined with another relation. One way for an insertion anomaly to occur would be if a tuple is inserted with attribute values that aren't consistent with the attribute values for another relation. To avoid this instance of an anomaly, NULL values may be used or the attribute values must correspond correctly to the joined relations.

Another form of the insertion anomaly occurs when a record is inserted that corresponds to a relation that has no records/contains NULL values. Since the relation is joined with another relation there exists the possibility that the primary key has a NULL value. Primary keys can't contain NULL values so this would result as a different form of an insertion anomaly;

Deletion Anomalies

Deletion anomalies is related to the second form of the insertion anomalies, specifically NULL values for attributes. Deleting a tuple from a joined relation or deletion of certain attributes of a tuple will result in the tuple and all existing information to be destroyed. To overcome this anomaly, instead of deleting attributes or the tuple, the record can instead house NULL values.

Modification Anomalies

In a joined relation, if a value for a record is changed, then all occurrences of that tuple must be updated as well or else the database will become inconsistent and thus have no integrity.

3.1.2 Relation Analysis and Updates

Most of the relations in our database already pass the first or second normal form because of the conversion from the E-R model to a Relational model. The following is a list of relations that pass the first, second, and third normal forms.

Employee: Employee_ID, F_Name, M_Init, L_Name, Address_ID, Phone, DOB, salary, SSN.

Candidate Keys:

Employee_ID, Address_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Department: Name, Department_ID, Description, Address_ID

_____Candidate Keys:

Department_ID, Address_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Stores: Ware_ID, Product_ID, Quantity

Candidate Keys:

Ware_ID, Product_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is NOT passed.

Explanation:

There is a partial dependency between Quantity and Product_ID. Because Quantity is a non prime attribute which depends on Product_ID, a subset of a candidate key, Quantity can't be determined without Product_ID.

Possible Anomalies:

There exists the possibility that a Product_ID is not listed in a customers receipt.

This is a result from the INSERT anomaly where the Product_ID contains a NULLvalue.

Solution:

The record containing a NULL Product_ID must also have Quantity contain a NULL value as well to maintain consistency.

Routes: Path_ID, Path_Name, Start_Time, Vehicle_Num, First_Address, Next_Address, Scheduled_Stops, Employee_ID

____Candidate Keys:

Path_ID, First_Address, Next_Address, Employee_ID, Address_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Locations: Loc_ID, City_Name, State, Zip_Code, Longitude, Latitude

Candidate Keys:

Loc_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Contracts: Contract_ID, Frequency, Start_Date, End_Date, Client_ID

_____Candidate Keys:

Contract_ID, Client_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Clients: Client_ID, F_Name, M_Init, L_Name, Address_ID, Phone, DOB, Credit_Card, E-mail

Candidate Keys:

Client_ID, Address_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Products: Product_ID, Product_Name, Sale_Price, Purchase_Price

Candidate Keys:

Product_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Warehouse: Ware_ID, Address_ID

____ Candidate Keys:

Ware_ID, Address_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Supplier: S_ID, S_Name

____ Candidate Keys:

S_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Purchase_Order: Order_ID, Date_Submitted, Date_Fulfilled, S_ID

____ Candidate Keys:

Order_ID, S_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial

dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Works_For: Department_ID, Employee_ID, Start_Date, End_Date

Candidate Keys:

Department_ID, Employee_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Manages: Department_ID, Employee_ID, Start_Date, End_Date

Candidate Keys:

Department_ID, Employee_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Contains: Contract_ID, Product_ID, Quantity

Candidate Keys:

Contract_ID, Product_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is NOT passed.

Explanation:

There is a partial dependency between Quantity and Product_ID. Because Quantity is a non prime attribute which depends on Product_ID, a subset of a candidate key, Quantity can't be determined without Product_ID.

Possible Anomalies:

There exists the possibility that a Product_ID is not listed in a customers receipt. This is a result from the INSERT anomaly where the Product_ID contains a NULLvalue.

Solution:

The record containing a NULL Product_ID must also have Quantity contain a NULL value as well to maintain consistency.

Supervises: Super_Emp_ID, Employee_ID, Start_Date, End_Date

Candidate Keys:

Super_Emp_ID, Employee_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Provisioned: Contract_ID, Loc_ID

_____Candidate Keys:

Contact_ID, Loc_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive

dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Distributed: Department_ID, Product_ID

____ Candidate Keys:

Department_ID, Product_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Supplies: S_ID, Ware_ID, Product_ID, Quantity

Candidate Keys:

S_ID, Ware_ID, Product_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is NOT passed.

Explanation:

There is a partial dependency between Quantity and Product_ID. Because Quantity is a non prime attribute which depends on Product_ID, a subset of a candidate key, Quantity can't be determined without Product_ID.

Possible Anomalies:

There exists the possibility that a Product_ID is not listed in a customers receipt. This is a result from the INSERT anomaly where the Product_ID contains a NULL value.

Solution:

The record containing a NULL Product_ID must also have Quantity contain a NULL value as well to maintain consistency.

Address: Address_ID, Street, City, State, Zip

Candidate Keys:

Address_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

Creates: Employee_ID, Order_ID

_____Candidate Keys:

Employee_ID, Order_ID

Reviewing Normal Forms:

1NF for the relation is passed; relation only contains atomic values

2NF for the relation is passed; passes 1NF and doesn't contain partial dependencies

3NF for the relation is passed; passes 2NF and contains no transitive dependencies

BNF for the relation is passed; passes 3NF and no attribute is functionally dependent on multiple candidate keys

3.2 Purpose and Functionality of PSQL

In order to understand the purposes and functionality of PostgreSQL, it is necessary to first understand the Structured Query Language and how it relates to databases. SQL is a cross between the Relational Algebra and Tuple Relational Calculus highlighted at the end of the last phase. In a more loose sense, it's a programming language used to manage data held in a relational database management system. This part is key—the data held in a relational database management system.

From this, it can be understood that SQL is a language used to design databases, but is not a full-fledged database-management system. Before explaining the explicit tie-in to PostgreSQL and how it relates, it's first necessary to dispel a rather common misconception: the Microsoft SQL Server. The name itself is a bit of a misnomer, as it is not the 'iconic' SQL server. From a technical standpoint, the vast majority of database systems use SQL, despite the implication that would otherwise be drawn from Microsoft's naming conventions.

To circle back around, PostgreSQL is a relational database management system which utilizes SQL. The software itself is free and open source, being maintained by the

PostgreSQL Global Development Group. As is the case with more open source software, it also has high compatibility with a variety of operating systems, such as FreeBSD, Linux, OpenBSD, OS X, Solaris, Unix, Windows, and a few others.

However, all of this circulates around the purpose of PostgreSQL, and not the functionality, so let's address that as well. PostgreSQL allows for user-defined functions (stored procedures) in a proprietary language, `pgSQL`, as well as in a few common programming languages, like Perl, Python, and Tcl. Since the 10.0 update, PostgreSQL also uses declarative partitioning, which involves partitioning either via a list or a range. It's important to consider that these are some of the more explicit differences behind the scenes between PostgreSQL and a few of the other common database management systems. One final note of difference is the ability to run PostgreSQL solely from the command line, in contrast with a database management system such as Microsoft SQL Server, which involves more use of a GUI.

In a more generalized sense, PostgreSQL also incorporates the majority of the basic expectations of a database management system. These include operations such as data definition, a.k.a. creating, changing, and removing definitions defining the structure of the data. PostgreSQL also handles updating, namely insertion, deletion, and modification of the actual data. One of the prime uses—retrieval, otherwise known as querying the database, which we'll explore in context of our own queries later this phase. Lastly, there is the administration aspect of a database management system, which involves the constraints, referential integrity, concurrency control, and security of the database.

3.3 Schema Objects in PostgreSQL

The following section will be about existing schema objects that relate to our database. The format will be presented in the following order: we will define the object and describe what their purpose is for, next the syntax of these objects will be stated and finally a list of the schema objects that exist in our database will be presented.

3.3.1 Schema Object Definitions

Tables

Tables are a type of object that is used to store information and their purpose is to be what houses records and how they would be organized. Tables would have attributes for columns, those columns are defined by a datatype. A record of data exists as a row with values for every column. In PostgreSQL tables are able to inherit from another table; the designations for these tables would be the child and parent table. Inheritance allows the parent table to appear in conjunction with the child table when a query is made to the child table. Optional constraints can also be made for columns as well as the table; if a constraint is breached then the name of the constraint will appear in error messages.

Syntax:

```
CREATE TABLE [GLOBAL | LOCAL] TABLE [IF NOT EXISTS] table_name  
_____  
(  
    column_name-1 datatype column_constraint,  
    ....
```

column_name-n datatype,

table_constraint

....

]);

[INHERITS (parent_table)]

[ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]

[TABLESPACE tablespace_name]

where column_constraint is:

[CONSTRAINT constraint_name]

{

NOT NULL | NULL

| CHECK (expression) [NO INHERIT]

| DEFAULT default_expr

| GENERATED

{ALWAYS | BY DEFAULT} AS IDENTITY [(sequence_options)]

| UNIQUE index_parameters

| PRIMARY KEY index_parameters

| REFERENCES reftable [(refcolumn)]

[MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]

[ON DELETE action] [ON UPDATE action]

}

and table_constraint is:

```
[CONSTRAINT constraint_name]  
{  
    CHECK (expression) [NO INHERIT]  
    | UNIQUE (column_name) index_parameters  
    | PRIMARY KEY (column_name) index_parameters  
    | EXCLUDE [USING index_method] (exclude_element WITH operator )  
index_parameters WHERE (predicate)  
    | FOREIGN KEY (column_name ) REFERENCES reftable [(refcolumn)]  
    [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE  
action] [ON UPDATE action]  
}
```

Tables from our database

- address
- employee
- department
- locations
- routes
- clients
- contracts
- products

- ware
- supplier
- purchase
- works
- manages
- supervise
- provision
- contains
- distributed
- stores
- supplies
- creates

Tablespace

A tablespace is where data is stored and allows super-users to move data, including objects such as tables) to a different physical location. Tablespaces are tied to a database since they are dependent on the databases' metadata therefore tablespaces can't be tied to a different database. The purpose of a tablespace is to allow database administrators to choose a location in the file system where the files (database objects) will be kept, an advantage to choosing locations would be that commonly used indexes,

tables, etc. can be placed into faster drives such as solid state drives (SSDs).

Syntax:

```
CREATE TABLESPACE tablespace_name  
[OWNER {new_owner | CURRENT_USER | SESSION_USER}]  
LOCATION 'directory'  
[WITH (tablespace_name = value)]
```

Schema

A database may contain one or more schemas which are logical containers of tables and other objects such as data types, functions...etc inside the database. Schemas are allowed to have the same object names which would be independent from another. Schemas allow multiple users utilize a database without interfering with another. The purpose of the schema is to provide a foundation for the database, organize objects into groups and provide ownership/restrict publicity of a schema to users.

Syntax:

```
CREATE SCHEMA schema_name [AUTHORIZATION role_specification]  
[schema_element]
```


CREATE SCHEMA AUTHORIZATION role_specification [schema_element]

CREATE SCHEMA IF NOT EXISTS schema_element [AUTHORIZATION
role_specification]

CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification

where role_specification can be the user name of:

| CURRENT_USER

| SESSION_USER

View

The view is a virtual table that is able to define and simplify large queries. When a view is created, a query is created which could be assigned a name. A view represents a subset of a table including joined tables. Views are able to be restricted which prevents defined users accessing certain rows or columns. The purpose of a view is to manage commonly used complex queries, structure data to improve accessibility and restrict access of certain records or columns to users.

Syntax:

CREATE VIEW [OR REPLACE] [TEMP] [RECURSIVE] view_name

_____ [WITH (view_option_name [= view_option_value])]

AS

query

_____ [WITH [CASCADED | LOCAL] CHECK OPTION]

Functions

Functions are sets of SQL statements that must have input parameters and would output at least one value. The main difference between a function and a procedure would be the functions outputs values while a procedure simply executes, however, functions may also have a set of commands though they must return a value. Users can also create their own functions where they are able to define their functions to return sets of base or composite values. The purpose of a function is to pass parameters where some sort of computational or mathematical operation will execute which returns a desired value.

Syntax:

CREATE [OR REPLACE] FUNCTION

```
name (  
    [[argmode] [argname] argtype [{DEFAULT | =} default_expr]]  
)  
  
[RETURNS ret_type  
    | RETURNS TABLE (column_name column_type)]  
  
{  
  
    LANGUAGE lang_name
```

```
| TRANSFORM {FOR TYPE type_name}  
| WINDOW  
  
| IMMUTABLE | STABLE | VOLATILE | [NOT] LEAKPROOF  
  
| CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT |  
  
STRICT  
  
| [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY  
DEFINER  
  
}
```

Procedures

Procedures have the ability to execute many commands that would normally be performed individually, they also can be a set of commands in a specified order. Because these procedures are stored, one can run them manually as well as other programs being able to utilize them. These procedures can be created in procedural languages such as Perl, Python, PL/pgSQL...etc. The difference between a procedure and a function would be that procedures may or may not return a value while a function must return a value. The purpose of procedures is to perform beyond SQL standard operations, such as looping and other complex calculations as well as significantly saving time by executing many repetitive commands/operations.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE
```

```

        name ([argmode ] [argname] argtype [{DEFAULT | = }
default_expr]))
    {
        LANGUAGE lang_name
        | TRANSFORM {FOR TYPE type_name}
        | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY
DEFINER
        | SET configuration_parameter
            {TO value | = value | FROM CURRENT}
        | AS 'definition'
        | AS 'obj_file', 'link_symbol'
    }

```

Collation

Collations defines how data can be compared and sorted inside a database. It can allow a sort order and character classification behavior for a piece of data which can go through per-column or per-operation. The purpose of collations is to determine how data is sorted based on the properties of a data such as case sensitivity, accents, and language characters.

Syntax:

```
CREATE COLLATION [IF NOT EXISTS] name
```

```
(
    [LOCALE = locale]
    [LC_CTYPE = lc_ctype]
    [PROVIDER = provider]
)

CREATE COLLATION [IF NOT EXISTS] name FROM existing_collation
```

Domains

A domain is a data type which can be a default type or user-defined that may have constraints which is used by column. The view of domains provides a list of all domains in the database if the user has access to those domains, otherwise some domains may not be listed for security purposes. The purpose of domains is to improve efficiency where a data type with constraints would be used often so a user-defined domain would be created that would encompass the data type as well as the constraint cleanly.

Syntax:

```
CREATE DOMAIN name [AS] data_type
    [COLLATE collation]
    [DEFAULT expression]
    [constraint]
```

where constraint is:

[CONSTRAINT constraint_name]
{NOT NULL | NULL | CHECK (expression)}

Sequences

Sequences (sequence generators/sequence objects) are tables with only one row that generate unique identifiers, mostly integers, for tuples of a particular table. As in the name, these newly created integers are in a sequential order. Because the integers are unique, the purpose of utilizing sequences are to create unique primary keys very easily due to the automation as well as correspond those keys to many rows across many tables.

Syntax:

```
CREATE [TEMPORARY | TEMP] SEQUENCE name [INCREMENT [BY]
increment]
[MINVALUE minvalue | NO MINVALUE] [MAXVALUE maxvalue | NO MAXVALUE]
[START [WITH] start ] [CACHE cache] [[NO] CYCLE]
```

Indexes

Indexes are a specific type of table which point to data in a table; indexes function similarly to a books index which direct a user to a different section of content. The purpose of indexes is to significantly speed up retrieving data as well as improve the the speed of SELECT queries and WHERE clauses. Downsides to indexes would be that updating a table with indexes takes more time than updating a table without indexes since the indexes themselves need to be updated as well, as such UPDATE and INSERT statements would have reduced speed in data modification.

Syntax:

```
CREATE [UNIQUE] INDEX index_name  
ON table_name (column-1, column-2, ... column-n)
```

Triggers

A trigger (trigger function) is a group of actions that are automatically executed when a specified event occurs, such as INSERT, UPDATE or DELETE statements. The trigger will correspond to a determined table, view, or foreign table. Triggers may also execute functions, from this triggers can be used for a variety of purpose such as: accessing system functions, validate input data, replicate data to different files to maintain integrity and consistency within a database...etc.

Syntax:

```
CREATE [CONSTRAINT] TRIGGER name {BEFORE | AFTER | INSTEAD OF}
{event [OR ...]}
ON table_name
[FROM referenced_table_name]
[NOT DEFERRABLE | [DEFERRABLE] [INITIALLY IMMEDIATE | INITIALLY
DEFERRED]]
[REFERENCING {{OLD | NEW} TABLE [AS] transition_relation_name}]
[FOR [EACH] {ROW | STATEMENT}]
[WHEN (condition)]
EXECUTE {FUNCTION | PROCEDURE} function_name (arguments)
```

where event can be one of:

INSERT

UPDATE [OF column_name]

DELETE

Packages

Packages group together logically related objects, functions, procedures, PL/SQL types...etc. A package encapsulates this grouping and is split among two parts, the package specification and the body/definition. The specification of the package

provides an interface where types, variables, exceptions, variables....etc. are declared. Any objects inside the specification are public objects whereas in the body, objects and other declarations are defined, rather than declared, and concealed from the code outside the package. The purpose of a package is to be used many times when developing applications since they house exceptions, types, procedures and functions that are all concealed from low privileged users.

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
[AUTHID {CURRENT_USER | DEFINER}]
{IS | AS}
[
  [definitions of public TYPES]
  ,[declarations of public variables, types, and objects]
  ,[PROCEDURE prodedure_name]
  ,[FUNCTION function_name]
]
END [package_name]
```

3.4 Relation Schema and Content

3.4.1 Relation Schema

Below is the relation schema of our database. Each relation has the attributes that correspond to them along with the type of attribute they are and if they allow null values. The contents of the relation schema will be below the schema corresponding to it.

Address Schema:

delivery = \d address

Table "address"		
Column	Type	Nullable
address_id	integer	not null
street	character varying(50)	
state	character(2)	
city_name	character varying(75)	
zip_code	integer	not null

Address Contents:

address_id	street	city_name	state	zip_code
1	670 Nonulen Crescent	Batavia	New Jersey	38186
2	240 Orin Gardens	Breckenridge	Washington	56681
3	326 Dorothy Close	Canton	Ohio	43152
4	326 Chesie Way	Seaside	Delaware	38144
5	314 Plendin Mead	Jacksonville	North Carolina	78676
6	409 Franie Way	Temple	Indiana	55406
7	554 Ginasea Rise	Pompano Beach	Kansas	19527
8	240 Sinibe Rise	Orlando	Kansas	73886
9	170 Arin Gardens	Aberdeen	Mississippi	56326
10	976 Susan Place	Priest River	Utah	48956
11	956 Stetie Hill	Lexington	Texas	90325
12	559 Sinolea Avenue	Durant	Utah	75695
13	862 Spondan Mead	Tallahassee	Montana	28364
14	331 Elizabeth Gardens	Alcoa	Utah	82469
15	284 Tetron Street	Michigan	Texas	96850
16	715 Ceafle Rise	Oakland	Ohio	92005
17	848 Arin Hill	Green River	Kansas	91612
18	946 Conanee Vale	Petaluma	Montana	77391
19	939 Stadie Place	Royal Oak	Illinois	50622
20	180 Stogie Street	Altoona	Nebraska	59277
21	128 Jeremiah Road	Lower Southampton	Oregon	74837
22	540 Oren Place	Summit	California	22359
23	508 Ninahea Grove	New Rochelle	Utah	44413
24	297 Banelan Rise	Mansfield	Wisconsin	21688
25	156 Feneye Way	Shreveport	Illinois	49117
26	762 Taspan Grove	Mississippi	Mississippi	68163
27	374 Wanameo Rise	Coupeville	Indiana	15839
28	969 Stelie Vale	Michigan City	Oregon	26997
29	650 Torsen Road	Sandwich	Louisiana	83406
30	295 Arin Mews	Millville	California	82608
31	892 Tanenei Mews	Winslow	North Carolina	69066
32	508 Tanetee Rise	Muskogee	Virginia	13791
33	381 Tochen Close	Portsmouth	Washington	43662
34	670 Taspin Road	French Lick	Louisiana	43081
35	645 Ellen Place	Hingham	Florida	85996
36	441 Eugene Square	Norwood	Nebraska	15080
37	755 Aron Street	Paterson	Nebraska	84155
38	655 Tieben Gardens	Mount Vernon	West Virginia	92460
39	325 Arin Way	Spring Green	Virginia	18399
40	285 Dineme Lane	Guntersville	Louisiana	70063
41	220 Doflon Hill	West Point	Wisconsin	93359
42	698 Britie Square	Bismarck	Louisiana	44569
43	953 Ceugon Road	Oxnard	Montana	93181
44	827 Eran Drive	Middletown	Mississippi	49409
45	379 Tispon Drive	Little Falls	South Carolina	81740
46	842 Arin Place	Barnstable	Illinois	53093
47	770 Teninee Mead	Lake Charles	Oregon	47430

48	792 Chagie Place	Mesa	Virgin Island	92038
49	885 Lanefe Vale	West Bridgewater	Virginia	48917
50	637 Denelan Mead	Rhode Island	Arizona	41982
51	5 Debs Park	San Diego	California	92160
52	42 Johnson Plaza	San Antonio	Texas	78260
53	435 Lighthouse Bay Pass	Detroit	Michigan	48275
54	08387 Autumn Leaf Circle	Suffolk	Virginia	23436
55	26 Holmberg Street	Fairbanks	Alaska	99790
56	7434 Fuller Lane	Charlotte	North Carolina	28263
57	4846 Iowa Center	San Francisco	California	94177
58	45990 Fremont Plaza	Muskegon	Michigan	49444
59	10 Maple Point	Lexington	Kentucky	40524
60	03 6th Pass	Charleston	West Virginia	25336
61	448 Pawling Lane	Salt Lake City	Utah	84145
62	019 Weeping Birch Pass	Anchorage	Alaska	99599
63	29 Judy Terrace	Peoria	Illinois	61656
64	4996 East Point	Bradenton	Florida	34210
65	8 Elgar Hill	Washington	District of Columbia	20010

Clients Schema:

delivery=# \d clients

Table "clients"		
Column	Type	Nullable
client_id	integer	not null
f_name	character varying(50)	
m_init	character varying(1)	
l_name	character varying(50)	
address_id	integer	not null
phone	bigint	
dob	date	not null
credit_card	bigint	not null
e_mail	character varying(75)	

Clients Contents:

client_id	f_name	m_init	l_name	address_id	phone	dob	credit_card	e_mail
1	Gustav	B	Ganelea	36	7241882710	1959-07-24	4981937169983648	stapie@lenfle.edu
2	Bonnie	G	Nathaniel	37	1095258344	1993-02-08	4556716572149046	manken@nathaniel.edu
3	Jasmin	Y	Donese	38	8386648846	1969-07-01	4716059369083798	daflen@huneneo.net
4	Nensen	T	Funuse	39	2783455415	1969-11-12	4556786613061187	tenonee@honsie.com
5	Conton	Q	Antonia	40	2162493276	1971-05-08	4024007133247139	herbert@darsie.net
6	Seymour	D	Monehe	41	9683095421	1971-05-08	4532592267648140	frodie@thinden.com
7	Benonee	F	Diblin	42	6801917898	1992-02-20	4189886715543515	hinewei@nicholas.com
8	Henilon	Y	Stedie	43	4698552894	1996-02-10	4716235268530234	oscar@jeffrey.edu
9	Lunahei	J	Mancen	44	4813982138	1992-10-31	4929376077162190	spendon@karfie.net
10	Shendin	A	Minhen	45	3121516206	2001-03-05	4539355619459290	dinome@braisie.net
11	Beverley	I	Cecilia	46	1881895455	1980-12-11	4024007109050335	harry@minose.com
12	Arin	G	Aron	47	4175458663	1974-07-17	4024007181213470	tispan@nenamee.edu
13	Cinosea	U	Oran	48	4297231985	1993-05-30	4485208570821020	danamei@spashon.com
14	Naneheo	O	Spendin	49	9937592607	2001-06-09	4929486784662570	tenoteo@loneneo.edu
15	John	P	Doe	50	2739776529	1962-10-06	4539819351632109	oran@glendan.net

Contains Schema:

delivery=# \d contains

Table "contains"		
Column	Type	Nullable

```

-----+-----+-----
contract_id | integer | not null |
product_id  | integer | not null |
quantity    | integer | not null |

```

Contains Contents:

```

contract_id | product_id | quantity
-----+-----+-----
8 | 6 | 14
15 | 13 | 12
29 | 14 | 3
4 | 3 | 7
33 | 15 | 19
34 | 9 | 23
19 | 5 | 1
1 | 8 | 8
9 | 10 | 4
3 | 11 | 7
11 | 2 | 5
35 | 12 | 15
13 | 4 | 18
20 | 1 | 13
2 | 7 | 9

```

Contracts Schema:

delivery=# \d contracts

Table "contracts"

```

Column | Type | Nullable |
-----+-----+-----
contract_id | integer | not null |
frequency   | integer |          |
start_date  | date   | not null |
end_date    | date   | not null |
client_id   | integer | not null |

```

Contracts Contents:

```

contract_id | frequency | start_date | end_date | client_id
-----+-----+-----+-----+-----
1 | 177 | 2016-02-25 | 2016-03-11 | 1
2 | 7 | 2017-05-23 | 2017-05-23 | 8
3 | 5 | 2018-01-19 | 2018-05-02 | 14
4 | 14 | 2018-03-30 | 2019-01-22 | 10
5 | 31 | 2018-06-13 | 2018-06-25 | 1
6 | 60 | 2019-01-22 | 2019-03-14 | 12
7 | 1 | 2018-11-28 | 2018-12-12 | 7

```

8	2	2018-09-14	2019-02-21	2
9	5	2018-04-27	2018-07-02	9
10	7	2016-03-16	2016-09-12	15
11	7	2017-07-05	2017-11-29	11
12	21	2016-10-28	2017-03-14	6
13	5	2018-07-10	2018-09-03	13
14	7	2016-12-29	2017-08-25	3
15	31	2018-03-30	2018-04-06	15
16	90	2018-10-16	2020-09-15	2
17	7	2016-05-25	2021-09-14	7
18	1	2017-08-05	2020-09-04	11
19	1	2019-05-03	2019-06-06	14
20	2	2018-10-06	2018-12-13	1
21	5	2019-05-13	2019-09-16	2
22	31	2017-05-03	2019-02-28	3
23	7	2018-08-28	2019-03-30	4
24	5	2017-06-27	2018-12-06	5
25	2	2018-11-12	2019-02-28	10
26	1	2017-06-29	2018-03-23	6
27	9	2017-10-16	2018-06-06	9
28	7	2017-05-29	2019-03-13	7
29	6	2017-06-07	2018-04-04	8
30	60	2017-09-15	2017-10-04	11
31	31	2018-12-26	2019-03-10	15
32	2	2017-08-29	2018-05-10	12
33	1	2017-10-14	2018-12-11	13
34	5	2018-03-22	2019-01-09	14
35	7	2017-12-19	2018-06-09	11

Creates Schema:

delivery=# \d creates

Table "creates"

Column	Type	Nullable
employee_id	integer	not null
order_id	integer	not null

Creates Contents:

employee_id | order_id

1	6
2	3
3	4
4	8
5	15
6	12
7	4
8	10
9	6
10	7
11	1
12	9

13	3
14	11
15	15
15	16
15	17
15	18
15	19
15	20
15	21
15	22
15	23
15	24
15	25
15	26
15	27
15	28
15	29

Department Schema:

delivery=# \d department

Table "department"			
Column	Type	Nullable	
department_id	integer	not null	
name	character varying(50)		
description	character varying(255)		
address_id	integer	not null	

Department Contents:

department_id	name	description	address_id
1	San Diego TS	Tech Support	51
2	San Antonio ACC	Accounting	52
3	Detroit ADMS	Admissions	53
4	Suffolk DEVL	Development	54
5	Fairbanks COMMS	Communications	55
6	Charlotte MAIL	Mail Services	56
7	San Francisco FRL	Food Research Lab	57
8	Muskegon OR	Outreach	58
9	Lexington HR	Human Resources	59
10	Charleston KT	Knowledge Transfer	60

11	Salt Lake City CD	Consumer Design		61
12	Anchorage MAIL	Mail Services		62
13	Peoria ADMS	Admissions		63
14	Bradenton ACC	Accounting		64
15	Washington TS	Tech Support		65

Distributed Schema:

delivery=# \d distributed

Table "distributed"

Column	Type	Nullable
department_id	integer	not null
product_id	integer	not null

Distributed Contents:

department_id | product_id

1		26
13		17
3		28
14		19
5		33
6		21
14		30
8		23
12		28
10		6
11		19
2		26
13		20
12		7
14		14

Employee Schema:

delivery=# \d employee

Table "employee"

Column	Type	Nullable
employee_id	integer	not null
f_name	character varying(50)	
m_init	character varying(1)	
l_name	character varying(50)	
address_id	integer	not null
phone	double precision	not null

dob	date	not null
salary	integer	not null
ssn	integer	not null

Employee Contents:

employee_id	f_name	m_init	L_name	address_id	phone	dob	salary	ssn
1	John	M	Doe	1	4996829581	2001-01-29	223283	310-35-7362
2	Detren	J	Nonodeo	2	4646535216	1967-02-15	57176	324-28-6325
3	Stolie	S	Curtis	3	9927477279	1992-07-14	850267	219-19-0535
4	Spondan	O	Oswald	4	2412281786	1973-05-01	27181	041-78-2212
5	Samuel	F	Nenlon	5	7932032122	1991-04-04	100146	206-22-1009
6	Spondin	U	Eron	6	7721422228	1984-05-18	79888	517-44-7960
7	Frewie	F	Aron	7	3065163876	1987-05-09	126567	217-56-3375
8	Cenoseo	E	Sineme	8	8296403566	1983-09-23	73624	481-86-6431
9	Binane	P	Oron	9	1806304101	1988-07-22	92057	489-74-8719
10	Tadren	T	Thonie	10	7496254607	1991-03-19	28806	182-78-4361
11	Tanalan	C	Maifan	11	3983211521	1983-09-11	35380	619-32-0231
12	Amanda	N	Marion	12	6815181326	1991-12-11	115015	023-86-1995
13	Oron	Q	Nichole	13	6031763277	1968-08-26	127564	252-12-7388
14	Roberta	K	Fralie	14	9525257019	2000-01-24	31679	264-15-4502
15	Spandan	H	Sonalan	15	5826364768	1958-06-11	85571	411-17-6684
16	Edmund	A	Nanilon	16	3613919197	1997-04-23	90423	051-72-7948
17	Nonolen	K	Freddie	17	6944129268	1958-06-03	185908	227-75-1598
18	Tenaye	D	Henile	18	4471804921	1966-08-12	140749	506-13-7306
19	Sincle	J	Cinekeo	19	7882545823	1983-01-02	135117	253-75-4298
20	Thetie	F	Lanlin	20	4433068345	1967-04-30	45703	382-33-1747
21	Nanelen	H	Orin	21	2803511226	1983-01-02	90209	088-56-8748
22	Shindan	Q	Stemie	22	6518232731	1987-07-07	32636	258-36-6861
23	Nanahei	M	Teflon	23	7657454751	1982-01-20	42664	374-24-3586
24	Spendan	E	Tonilen	24	2402929588	1970-08-10	121732	574-09-3088
25	Trendan	B	Sharon	25	8341329383	1991-12-03	161018	228-88-3661
26	Hunewee	T	Flindan	26	2161293222	1992-12-22	125337	491-18-9722
27	Brivie	V	Spendon	27	2743538808	1967-11-10	67829	098-70-7953
28	Isaac	U	Rodney	28	8771809848	1991-12-03	113587	245-36-5284
29	Baniwea	C	Calvin	29	2333186403	1991-06-14	28029	473-05-7669
30	Monele	I	Tardie	30	1872148264	1976-05-25	54090	308-44-3885
31	Sanelen	X	Daihin	31	6371213707	1965-09-08	109669	516-43-5332
32	Nanohe	O	Frocie	32	9766701150	1993-11-07	44290	562-65-2471
33	Sorgle	Y	Flinden	33	7435818391	1962-07-28	85141	549-68-3320
34	Hanenee	R	Tonsin	34	2801372695	1995-06-17	113871	117-86-2311
35	Nonelon	B	Stayie	35	9951761553	1958-03-31	74305	411-12-7030

Locations Schema:

delivery=# \d locations

Table "locations"

Column	Type	Nullable
loc_id	integer	not null
state	character varying(50)	
city_name	character varying(75)	
zip_code	integer	
longitude	numeric	not null
latitude	numeric	not null

Locations Contents:

loc_id	state	city_name	zip_code	longitude	latitude
--------	-------	-----------	----------	-----------	----------

1	Batavia	New Jersey	38186	-115.26693737875792	34.59256079081971
2	Breckenridge	Washington	56681	-100.06185925375792	47.809499046844714
3	Canton	Ohio	43152	-120.54037487875792	40.876179237371524
4	Seaside	Delaware	38144	-108.14779675375792	36.20443106831824
5	Jacksonville	North Carolina	78676	-97.90853894125792	28.415603850501267
6	Temple	Indiana	55406	-82.39584362875792	27.054278360760208
7	Pompano Beach	Kansas	19527	-98.74349987875792	35.16935892993562
8	Orlando	Kansas	73886	-76.46322644125792	39.87184204561187
9	Aberdeen	Mississippi	56326	-69.82748425375792	44.17830040661964
10	Priest River	Utah	48956	-84.54916394125792	44.74286725545292
11	Lexington	Texas	90325	-97.46908581625792	47.54319742359321
12	Durant	Utah	75695	-100.45736706625792	41.701664694161906
13	Tallahassee	Montana	28364	-115.31088269125792	43.28923837422243
14	Alcoa	Utah	82469	-116.98080456625792	33.02252386425122
15	Michigan	Texas	96850	-123.61654675375792	47.09633931054157
16	Nashville	Tennessee	62701	-86.78156921872693	36.14845021775153
17	Atlanta	Georgia	14425	-84.42126551875668	33.724997707193125
18	Jackson	Mississippi	23421	-90.1566499631287	32.324420285439615
19	Amarillo	Oklahoma	13634	-101.82310707040921	35.22888891792396
20	Santa Fe	New Mexico	16635	-105.92347839668582	35.66330613906651
21	Los Angeles	California	90263	-118.21893641546558	34.030385446752355
22	Las Vegas	Nevada	93725	-115.13353505924533	36.17719947062528
23	Fort Worth	Texas	78402	-97.36563825462917	32.75768762742407
24	Houston	Texas	11418	-95.44478767965893	29.780370747552833
25	Alcoa	Utah	82469	-113.59049447822667	37.081399339143324
26	Lincoln	Nebraska	90806	-96.6516833886675	40.774384265345326
27	Denver	Colorado	65211	-104.9598083436714	39.738667964585645
28	Sioux Falls	South Dakota	3054	-96.67207732240877	43.55407743718912
29	Des Moines	Iowa	96766	-93.56470330993852	41.603732215276025
30	Springfield	Illinois	27888	-89.63511153499803	39.79447795630532
31	Madison	Wisconsin	80024	-89.39618297435527	43.07368279950761
32	Columbia	Missouri	54902	-92.3728554558395	38.92425734119287
33	Detroit	Michigan	61354	-83.08248022673194	42.35819404793734
34	Cleveland	Ohio	17777	-81.66030145534046	41.48485014806882
35	Louisville	Kentucky	16433	-85.71330345029355	38.20573669391505

Manages Schema:

delivery=# \d manages

Table "manages"

Column | Type | Nullable |

```

-----+-----+-----
department_id | integer | not null |
employee_id   | integer | not null |
start_date    | date   | not null |
end_date      | date   |         |

```

Manages Contents:

department_id | employee_id | start_date | end_date

```

-----+-----+-----
7 |      8 | 1989-08-24 | 1989-09-15
13 |     15 | 1968-11-04 | 1969-07-27
8 |     29 | 1990-07-04 | 2017-10-13
15 |      4 | 1981-11-27 | 1983-01-26
15 |     33 | 1999-08-02 | 2006-03-20
9 |     34 | 1997-09-17 | 2018-03-21

```

```

12 | 19 | 1979-04-03 | 1981-04-23
8 | 1 | 1958-06-27 | 1959-09-26
9 | 9 | 1958-08-02 | 1960-03-10
11 | 3 | 1971-08-08 | 1974-02-10
2 | 11 | 1970-11-25 | 1971-07-17
12 | 35 | 1998-03-24 | 2018-03-15
13 | 13 | 1994-08-31 | 1994-10-30
1 | 35 | 1996-01-02 | 2000-11-16
10 | 3 | 1967-06-09 | 1968-05-31

```

Products Schema:

delivery=# \d products

Table "products"			
Column	Type	Nullable	
product_id	integer	not null	
product_name	character varying(50)		
sale_price	numeric		
purchase_price	numeric		

Products Contents:

product_id	product_name	sale_price	purchase_price
1	Fried Rice	7.99	5.99
2	Chicken Breast	19.99	14.99
3	Bacon	17.99	12.49
4	Pork Sausage	11.99	5.99
5	Breaded Chicken Patty	11.99	5.99
6	Beef Hot Dog	5.99	2.49
7	Sirloin Steak	26.99	18.99
8	Pulled Pork	13.99	7.99
9	Honey Ham	44.99	28.99
10	Pineapple Chunks	6.49	2.99
11	Coconut Strips	8.99	4.49
12	Strawberry Oatmeal	5.99	2.49
13	Baby Carrots	4.99	1.99
14	Green Peas	6.49	2.99
15	Broccoli Florets	6.99	3.49
16	Crispy Fish Strips	12.99	10.49
17	Cod Fillet	17.99	12.99
18	Breaded Haddock Sticks	12.99	9.49
19	Macaroni and Cheese	3.99	1.49
20	Green Peas	6.49	3.29
21	Whole Blueberries	7.49	3.99
22	Broccoli and Cauliflower	6.99	2.99
23	Asparagus Spears	10.99	5.99
24	Whole Strawberries	6.49	3.49
25	Steak Fries	6.99	3.99
26	Vegetable Fried Rice	5.99	2.49
27	Homestyle Waffles	5.99	1.99

28		Buttermilk Pancakes		6.99		2.49
29		Mini Donuts		8.99		4.99
30		Blueberry Scone Dough		14.99		9.49
31		Lemon Meringue Pie		11.19		8.99
32		Chocolate Brownie Bites		9.99		6.49
33		Chocolate Crème Pie		10.99		8.49
34		Monster Cookie Dough		10.99		7.99
35		Frozen Pizza		10.99		7.99

Provisioned Schema:

delivery=# \d provisioned

Table "provisioned"

Column	Type	Nullable
contract_id	integer	not null
loc_id	integer	not null

Provisioned Contents:

contract_id	loc_id
26	6
19	13
32	14
25	3
28	15
16	9
13	5
8	8
15	10
29	11
4	2
33	12
34	4
19	1
20	13
21	14
22	15
23	16
24	17
25	18
26	19
27	20
29	21
30	22
31	23
32	24
33	25
34	26

Purchase_Order Schema:

```
delivery=# \d purchase_order
          Table "purchase_order"
   Column   | Type   | Nullable |
-----+-----+-----
order_id    | integer | not null |
date_submitted | date   | not null |
date_fulfilled | date   |          |
s_id        | integer | not null |
```

Purchase_Order Contents:

```
order_id | date_submitted | date_fulfilled | s_id
-----+-----+-----+-----
1 | 2015-02-20 | 2015-03-10 | 5
2 | 2019-02-14 | 2019-03-04 | 15
3 | 2016-02-19 | 2016-03-09 | 11
4 | 2018-04-18 | 2018-04-24 | 9
5 | 2017-03-15 | 2017-04-07 | 12
6 | 2015-12-14 | 2015-12-22 | 7
7 | 2017-01-18 | 2017-02-09 | 13
8 | 2018-12-17 | 2019-02-11 | 8
9 | 2015-12-14 | 2015-12-22 | 4
10 | 2017-08-15 | 2017-09-08 | 10
11 | 2015-01-19 | 2015-01-26 | 3
12 | 2018-02-02 | 2018-02-27 | 1
13 | 2016-03-10 | 2016-06-24 | 6
14 | 2018-10-03 | 2018-10-09 | 14
15 | 2015-11-19 | 2015-12-08 | 2
16 | 2015-11-19 | 2015-12-08 | 1
17 | 2015-11-19 | 2015-12-08 | 3
18 | 2015-11-19 | 2015-12-08 | 4
19 | 2015-11-19 | 2015-12-08 | 5
20 | 2015-11-19 | 2015-12-08 | 6
21 | 2015-11-19 | 2015-12-08 | 7
22 | 2015-11-19 | 2015-12-08 | 8
23 | 2015-11-19 | 2015-12-08 | 9
24 | 2015-11-19 | 2015-12-08 | 10
25 | 2015-11-19 | 2015-12-08 | 11
26 | 2015-11-19 | 2015-12-08 | 12
27 | 2015-11-19 | 2015-12-08 | 13
28 | 2015-11-19 | 2015-12-08 | 14
29 | 2015-11-19 | 2015-12-08 | 15
```

Routes Schema:

delivery=# \d routes

Table "routes"			
Column	Type	Nullable	
path_id	integer	not null	
path_name	character varying(50)		
start_time	integer		
vehicle_num	integer	not null	
first_address	integer	not null	
next_address	integer	not null	
scheduled_stops	integer		
employee_id	integer	not null	

Routes Contents:

path_id	path_name	start_time	vehicle_num	first_address	next_address	scheduled_stops	employee_id
1	Pfeifer	930	310440	20	32	17	31
2	Lackey	805	479697	11	12	13	23
3	Rea	910	196655	8	18	9	35
4	Chau	820	568149	34	16	12	4
5	Rhoades	900	247115	23	13	26	25
6	Temple	850	78251	32	12	23	21
7	Lerma	835	274910	5	3	16	11
8	Cota	810	774514	17	4	19	1
9	Spear	840	921683	19	14	27	9
10	Merchant	825	627712	34	13	31	27
11	Tovar	855	417352	13	32	15	11
12	Fenton	905	710080	12	18	18	22
13	Conroy	815	557523	10	19	8	14
14	Burgos	925	536264	35	14	13	28
15	Dodd	800	633751	27	10	21	33
16	Bob	800	633751	26	10	21	33
17	Norbert	800	633751	25	10	21	33
18	Old River	800	633751	24	10	21	33
19	White	800	633751	23	10	21	33
20	Panama	800	633751	22	10	21	33
21	Buena Vista	800	633751	21	10	21	33
22	Truxtun	800	633751	20	10	21	33
23	Cheerio	800	633751	19	10	21	33
24	Captain	800	633751	18	10	21	33
25	Minfilia	800	633751	17	10	21	33
26	Eris	800	633751	16	10	21	33
27	Mountain Vista	800	633751	15	10	21	33
28	Rodgers	800	633751	14	10	21	33
29	Adams	800	633751	13	10	21	33

Stores Schema:

delivery=# \d stores

Table "stores"			
Column	Type	Nullable	
ware_id	integer	not null	
product_id	integer	not null	
quantity	integer	not null	

Stores Contents:

ware_id	product_id	quantity
3	6	14
4	13	12
9	14	3
2	3	7
10	15	19
13	9	23
6	5	1
15	8	8
7	10	4
10	11	7
1	2	5
5	12	15
9	4	18
12	1	13
3	7	9

Supervises Schema:

delivery=# \d supervises

Table "supervises"

Column	Type	Nullable
super_emp_id	integer	not null
employee_id	integer	not null
start_date	date	not null
end_date	date	

Supervises Contents:

super_emp_id	employee_id	start_date	end_date
14	1	1961-06-03	1965-03-18
9	1	1961-02-02	1961-03-26
6	17	1973-05-29	1974-07-27
12	18	1980-05-13	1982-04-07
15	22	1982-07-30	2002-03-20
12	5	1983-11-09	1986-02-03
10	9	1961-03-06	1962-05-26
13	21	1978-12-18	1984-03-10
6	18	1987-02-23	1989-05-26
3	7	1973-07-04	1978-10-30
12	11	1973-02-01	1975-01-15
3	17	1976-07-20	1978-11-22
2	27	1994-11-01	2019-01-21
5	9	1962-06-10	1964-06-02
4	13	1999-08-26	2002-01-07

Supplier Schema:

```
delivery=# \d supplier
          Table "supplier"
Column |      Type      | Nullable |
-----+-----+-----
s_id   | integer         | not null |
s_name | character varying(75) |      |
```

Supplier Contents:

```
s_id | s_name
-----+-----
1 | Lopez Foods
2 | Keystone Foods
3 | Gavina Gourmet
4 | Groupe Danone
5 | Brake Foods
6 | HKTDC
7 | Atlas Wholesale
8 | Smart Foods
9 | EVCO
10 | Symrise
11 | ADM
12 | MGP Ingredients
13 | AAK
14 | Food Lion
15 | 100KM Foods
```

Supplies Schema:

```
delivery=# \d supplies
          Table "supplies"
Column |      Type      | Nullable |
-----+-----+-----
s_id   | integer         | not null |
ware_id | integer         | not null |
product_id | integer       | not null |
quantity | integer        | not null |
```

Supplies Contents:

```
s_id | ware_id | product_id | quantity
-----+-----+-----+-----
3 | 6 | 9 | 182
```

4	13	35	84
9	14	8	112
2	3	10	71
10	15	11	38
13	9	2	24
6	5	12	73
15	8	35	7
7	10	1	46
10	11	7	146
1	2	35	93
5	12	17	88
9	4	28	181
12	1	19	52
3	7	33	54

Warehouse Schema:

```
delivery=# \d warehouse
        Table "warehouse"
   Column | Type   | Nullable |
-----+-----+-----
ware_id  | integer | not null |
address_id | integer | not null |
```

Warehouse Contents:

```
ware_id | address_id
-----+-----
1 | 51
2 | 52
3 | 53
4 | 54
5 | 55
6 | 56
7 | 57
8 | 58
9 | 59
10 | 60
11 | 61
12 | 62
13 | 63
14 | 64
15 | 65
```

Works_For Schema:

```
delivery=# \d works_for
        Table "works_for"
```


Column	Type	Nullable
department_id	integer	not null
employee_id	integer	not null
start_date	date	not null
end_date	date	

Works_For Contents:

department_id	employee_id	start_date	end_date
8	1	1958-06-27	1959-09-26
15	30	1994-10-01	2018-10-13
12	5	1983-11-09	1986-02-03
4	24	1987-10-03	2011-12-17
10	12	1975-01-30	1983-05-19
6	6	1958-06-27	1959-05-29
7	8	1989-08-24	1989-09-15
1	14	1968-04-09	1970-10-09
9	9	1958-08-02	1960-03-10
3	23	1975-08-06	2010-10-12
11	3	1971-08-08	1974-02-10
3	2	1967-04-03	1967-05-05
13	13	1994-08-31	1994-10-30
14	7	1971-11-21	1972-10-30
2	11	1970-11-25	1971-07-17
1	11	1970-11-25	1971-07-17
3	11	1970-11-25	1971-07-17
4	11	1970-11-25	1971-07-17
5	11	1970-11-25	1971-07-17
6	11	1970-11-25	1971-07-17
7	11	1970-11-25	1971-07-17
8	11	1970-11-25	1971-07-17
9	11	1970-11-25	1971-07-17
10	11	1970-11-25	1971-07-17
11	11	1970-11-25	1971-07-17
12	11	1970-11-25	1971-07-17
13	11	1970-11-25	1971-07-17
14	11	1970-11-25	1971-07-17
15	11	1970-11-25	1971-07-17

3.5 Query Conversion to SQL

From the previous phase we will convert the previous queries into the SQL format. All queries are using SQL features such as (GROUP BY, HAVING, FULL OUTER JOIN, INNER JOIN and aggregate functions). Each query will be presented, shown in the SQL format and finally the data retrieval of the query will be outputted.

1. List all of the employees that have delivered to a location for every client.

```
SELECT e.Employee_ID, e.F_Name, e.L_Name FROM employee e INNER JOIN
      (SELECT Employee_ID FROM (routes r INNER JOIN locations l
      ON r.First_Address = l.Loc_ID) as p1
      INNER JOIN (provisioned d INNER JOIN contracts c
      ON d.Contract_ID = c.Contract_ID) as p2
      ON p1.Loc_ID = p2.Loc_ID) as o
      ON e.Employee_ID = o.Employee_ID
      GROUP BY e.Employee_ID
      HAVING count(e.Employee_ID) >= 15;
```

```
-- Query #1
employee_id | f_name | l_name
-----+-----+-----
33 | Sorgle | Flinden
(1 row)
```

2. List every employee that has worked in every department.

```
SELECT e.Employee_ID, e.F_Name, e.L_Name FROM employee e INNER JOIN
      (SELECT Employee_ID FROM works_for natural join department) as d
      ON d.Employee_ID = e.Employee_ID
      GROUP BY e.Employee_ID HAVING count(e.Employee_ID) = 15;
```

```

-- Query #2
employee_id | f_name | l_name
-----+-----+-----
11 | Tanalan | Maifan
(1 row)

```

3. List all of the suppliers that have supplied “frozen pizza” to a warehouse.

```

SELECT s.S_ID, s.S_Name FROM supplier s INNER JOIN
    (SELECT S_ID FROM (warehouse w INNER JOIN supplies p
        ON w.Ware_ID = p.Ware_ID) as p1
    INNER JOIN products d ON d.Product_ID = p1.Product_ID
    AND d.Product_Name = 'Frozen Pizza') as out
    ON s.S_ID = out.S_ID ORDER BY s.S_ID;

```

```

-- Query #3
s_id | s_name
-----+-----
1 | Lopez Foods
4 | Groupe Danone
15 | 100KM Foods
(3 rows)

```

4. List the customers with year long contracts of at least weekly deliveries.

```

SELECT c.Client_ID, C.F_Name FROM clients c INNER JOIN
    (SELECT * FROM ((SELECT t.CONTRACT_ID, (DATE_PART('year', t.End_Date) -
        DATE_PART('year', t.Start_Date)) as Years
    FROM contracts t) as o
    NATURAL JOIN contracts) WHERE o.Years >= 1) as o1
    ON c.Client_ID = o1.Client_ID WHERE o1.Frequency >= 7
    ORDER BY c.Client_ID;

```

```

-- Query #4
client_id | f_name
-----+-----
         2 | Bonnie
         3 | Jasmin
         3 | Jasmin
         4 | Nensen
         6 | Seymour
         7 | Benonee
         7 | Benonee
         9 | Lunahei
        10 | Shendin
        11 | Beverley
        15 | John
(11 rows)

```

5. List the employees that have managed at least two departments.

```

SELECT e.Employee_ID, e.F_Name, e.L_Name FROM employee e NATURAL JOIN
      (SELECT m.Employee_ID FROM manages m GROUP BY m.Employee_ID HAVING
count(m.Employee_ID) >= 2) as o
ORDER BY e.Employee_ID;

```

```

-- Query #5
employee_id | f_name | l_name
-----+-----+-----
         3 | Stolie | Curtis
        35 | Nonelon | Stayie
(2 rows)

```

6. List the employees that have delivered to all the locations that employee John Doe has.

```

WITH T AS
      (SELECT Employee_ID FROM employee
      WHERE F_Name = 'John' AND L_Name = 'Doe')
SELECT Employee_ID, F_Name, L_Name FROM (SELECT Employee_ID FROM

```

(SELECT First_Address FROM T NATURAL JOIN routes) as o

NATURAL JOIN routes) as o2 NATURAL JOIN employee;

-- Query #6

employee_id	f_name	l_name
1	Yanegea	Edward
33	Sorgle	Flinden
36	John	Doe

(3 rows)

7. List all of the suppliers that supply the product that has the highest sale price.

SELECT s.s_id, S_Name, Product_Name FROM supplier s FULL OUTER JOIN

(SELECT s_id, Product_Name FROM (supplies FULL OUTER JOIN

(SELECT product_id, Purchase_Price, product_name FROM ((SELECT

max(Purchase_Price)

FROM products) as out INNER JOIN products ON out.max =

products.Purchase_Price)) as o1

on supplies.Product_ID = o1.product_ID) WHERE purchase_Price IS NOT

NULL) as o2

ON s.S_ID = o2.S_ID WHERE Product_Name IS NOT NULL;

-- Query #7

s_id	s_name	product_name
3	Gavina Gourmet	Honey Ham

(1 row)

8. List the product that has the second lowest sale price of those that are in an active contract.

WITH T AS

```
(SELECT * FROM (SELECT p1.product_id, DATE_PART('day', c.end_date)
- (DATE_PART('day', CURRENT_DATE)) AS days FROM (products p1 natural join
contracts c)) as o
NATURAL JOIN products p2 WHERE o.days > 0)
SELECT p.product_id, p.product_name, p.sale_price FROM ((SELECT product_id FROM T
WHERE Sale_Price IN (SELECT MIN(Sale_Price) FROM T WHERE Sale_Price NOT
IN
(SELECT MIN(Sale_Price) FROM T)) GROUP BY product_id) as out NATURAL
JOIN products p);
```

-- Query #8

product_id	product_name	sale_price
13	Baby Carrot	4.99

(1 row)

9. List the employees who have submitted purchase orders to every supplier.

```
SELECT e.Employee_ID, e.F_Name, e.L_Name FROM employee e INNER JOIN
(SELECT Employee_ID FROM creates natural join purchase_order
natural join supplier) as s
ON e.Employee_ID = s.Employee_ID
GROUP BY e.Employee_ID HAVING count(e.Employee_ID) = 15;
```

-- Query #9

employee_id	f_name	l_name
15	Spandan	Sonalan

(1 row)

10. List the item that “John Doe” has ordered the least of, but has ordered at least once.

```
SELECT product_ID, product_name, min(Sale_Price) FROM products NATURAL JOIN
      (SELECT product_ID FROM contains NATURAL JOIN (SELECT * FROM contracts
      NATURAL JOIN (SELECT Client_ID FROM clients
      WHERE F_Name = 'John' AND L_Name = 'Doe') as o) as p) as q
GROUP BY product_ID, product_name;
```

```
-- Query #10
product_id | product_name | min
-----+-----+-----
      13 | Baby Carrots | 4.99
      (1 row)
```

3.6 Data Loader

As with every database, data entry plays a key part in the building of the initial database, as well as when there needs to be any new insertions, updates, or deletions to said database. There's a few different methods for handling said insertion. The most simplistic of those would be the generalized 'insert' statements, which we favored heavily for our setup. However, as the simplistic adjective implies, it scales rather poorly as all of the statements are created by hand, which really is not realistic for any live database. In that sense, a Data Loader program would be most ideal, as we'll see when looking at Dr. Huaqing Wang's java implementation of a Data Loader for an Oracle Database.

SQL Insert Statements

As was mentioned before, the insert statement approach was fairly simplistic, and that played a large part in the reasoning for why we favored using it so much in the entry of our data. Another prime reason for our use was that we did not have our data structured in a uniform format under which a Data Loader program would have been preferable. As such, we deemed

the time trade off in the use of the Insert statements to be equal to the time investment of trying to reformat our data to a suitable form. However, more closely related to the insert statements is the two general approaches for its use.

1. INSERT INTO (table) VALUES (value, value...);
2. INSERT INTO (table) (query);

Both of these approaches are fairly self-explanatory. The former allows for inserting explicit data into a specified table. This was the format we chose to use for our insertions, as can be seen in the accompanying insert files for our database. The latter we did not choose to use, as the queries were more for tests of integrity and accuracy than they were for data creation. However, if necessary, the secondary approach would insert the output of a query into the specified table.

Java Data Loader

Unfortunately, we weren't overly familiar with the concept of data loaders during our work on Phase II, or we would have likely placed a stronger emphasis on the formatting of the data we created, to facilitate for easier usage of a data loader program. Specifically using Dr. Wang's java implementation of a Data Loader, the concept seems to revolve around processing an input file of a specified format (txt in this case), with a particular formatting within the file itself as well. The data loader would then parse through the file based upon these parameters and then construct insert statements similar to those discussed previously. While performance-wise the difference would be negligible, the difference in time is the prime factor. For a live database, you're likely dealing with thousands of entries for tables, creating an insert statement would be a painstaking task in such a situation, giving the Data Loader program approach of automating that aspect of data entry a very significant edge over directly creating an insert statement. Therefore, it can be concluded that the pros and cons of each method would likely come down to the scale of the database, but with the Data Loader generally being the preferred option.

Phase 4: DBMS Procedural Language, Stored Procedures, and Triggers

Nearly all of this phase will be spent analyzing more of the syntactical aspects of the languages behind the various database management systems. In particular, the first section will be spent looking into some of the intricacies of the procedural language utilized by Postgres: PL/pgSQL. While the bulk of it will be spent looking at the syntax of the language, the section will also briefly offer some highlights on procedures, functions, packages, and triggers.

Moving into the second section we'll be taking a live look at several procedures, functions, and triggers within the scope of our own database. The section will include sample code, as well as the effects they have on the data itself. Lastly, the final section will feature a high level overview of several of the other popular database management systems, such as Oracle, MySQL, and Microsoft SQL Server.

4.1 Postgres PL/pgSQL

SQL is the language that most databases systems use as their query language. It's easy to learn and portable. However, every SQL statement must be executed individually by the database server. The individual queries must be sent one at a time, wait for the queries to be processed, receive the results, and do some computation. The solution is to use PL/pgSQL to group a block of computation and a series of queries inside the database server. Throughout this section, we will explain the syntax needed to create functions, procedures, packages, triggers and cursors along with their benefits.

4.1.1 Postgres PL/pgSQL Language

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. It is mostly used to create functions, procedures, triggers and packages.

PL/pgSQL is a block-structured language and a block is defined as

```
[ <<label>> ]  
[ DECLARE  
  declarations ]  
BEGIN  
  statements  
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon. The label is only used if you want to identify the block for use in an EXIT statement, or to qualify the names of the variables declared in the block. The label given after block must match the label at the beginning of the block. Now that a block has been defined, the following subsections will be about the components that can be created using PL/pgSQL.

Functions

Functions written in PL/pgSQL are defined to the server by executing CREATE FUNCTION commands. This will create a new stored function for the database. 'Function body text' is the block that was shown above. The block would change in appearance depending on the function that is being created. Having stored functions in the database reduce the time and effort of having to parse queries. Stored functions also accept any scalar or array data types, and can return a result of any of these data types.

Syntax:

```
CREATE FUNCTION function_name(integer, text) RETURNS integer AS
```

'function body text'

LANGUAGE plpgsql;

Procedures

In Postgres, procedural languages such as PL/pgSQL, C, Python, and Tcl are referred to as stored procedures. Stored procedures define functions for creating triggers or custom aggregate functions. Using a stored procedure instead sending SQL statements from a front-end/client software to a DBMS server reduces the number of round trips between applications and database servers. All SQL statements are inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call. Stored procedures increase application performance as well because the user-defined functions and stored procedures pre-compiled and stored in the database server. The following is the syntax for creating or replacing a procedure.

Syntax:

CREATE [OR REPLACE] PROCEDURE procedure_name (parameter_list)

LANGUAGE language_name

AS \$\$

stored_procedure_body;

\$\$;

Cursors

Cursors are typically used within applications that maintain a persistent connection to the PostgreSQL backend. A cursor allows us to encapsulate a query and process each individual row at a time. Cursors are used when we want to divide a large result set into parts and process each part individually. A function can also be developed that returns a reference to a cursor. The caller of the function can process the result set based on the cursor reference which allows the caller to read the rows. This provides an efficient way to return large row sets from functions. Below is the syntax used to create a cursor. A cursor is both created and executed with the DECLARE statement, which is also known as opening a cursor. The optional BINARY keyword causes output to be retrieved in the form of binary instead of ASCII. The INSENSITIVE keyword exists to ensure that all data retrieved from the cursor remains unchanged from other cursors or connections. The SCROLL keyword exists to specify that multiple rows can be selected at a time from the cursor. CURSOR FOR query is the complete query whose result set will be accessible by the cursor when executed. FETCH is used to retrieve rows from a cursor.

Syntax:

DECLARE

cursor_name

[BINARY] [INSENSITIVE] [SCROLL]

CURSOR FOR

query

[FOR { READ ONLY | UPDATE [OF
column
[...]] }]

FETCH [FORWARD | BACKWARD | RELATIVE]
[# | ALL | NEXT | PRIOR]
{ IN | FROM }

cursor

Triggers

CREATE TRIGGER creates a DML, DDL, or a logon trigger. A trigger is a special type of stored procedure that automatically runs when an event occurs in the database server. A **DML (Data Manipulation Language)** trigger runs when a user tries to modify data using INSERT, DELETE, or UPDATE statements on a table or view. **DDL (Data Definition Language)** triggers run in response to a variety of to a variety of DDL events. These events primarily correspond to Transact-SQL CREATE, ALTER, DROP statements, and certain system stored procedures that perform DDL-like operations. Logon triggers fire in response to the LOGON event that's raised when a user's session is being established. Only the syntax that corresponds to PL/SQL will be presented while Transact-SQL syntax will be used in a later section.

Syntax:

CREATE [CONSTRAINT] TRIGGER name {BEFORE | AFTER | INSTEAD OF} {event
[OR ...]}

ON table_name

[FROM referenced_table_name]

[REFERENCING {{OLD | NEW} TABLE [AS] transition_relation_name}]

[FOR [EACH] {ROW | STATEMENT}]

[WHEN (condition)]

EXECUTE {FUNCTION | PROCEDURE} function_name (arguments)

where event can be one of:

INSERT

UPDATE [OF column_name]

DELETE

Packages

For the sake of describing what a package is, this section will be referring to the Oracle Database language PL/SQL, instead of PL/pgSQL. The equivalent of a package for PL/pgSQL is a schema which would need to be changed in certain parts in order to work. A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications share its contents. A package always has a specification, which

declares the public items that can be referenced from outside the package. If the public items include cursors or subprograms, then the package must also have a body. The package body will be described later.

In either the package specification or package body, you can map a package subprogram to an external Java or C subprogram by using call specification which maps the external subprogram name, parameter types, and return type to their SQL counterparts. The AUTHID clause of the package specification determines whether the subprograms and cursors in the package run with the privileges of the definer or the invoker. It also determines whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. The ACCESSIBLE BY clause of the package specification lets you specify a white list of PL/SQL units that can access the package.

Package Body

The body must define queries for public cursors and code for public subprograms. The body can also declare and define private items that cannot be referenced from outside the package, but are necessary for the internal workings of the package. Finally, the body can have an initialization part, whose statements initialize variables and do other one-time setup steps, and an exception-handling part. You can change the body without changing the specification or the references to the public items. The following is the syntax for a schema, which is the equivalent to PL/SQL's package with a few changes.

Syntax:

CREATE SCHEMA schema_name

```
[ AUTHORIZATION username ] [ schema_element [ ... ] ]
```

```
CREATE SCHEMA AUTHORIZATION username
```

```
[ schema_element [ ... ] ]
```

4.2 Postgres PL/pgSQL Subprograms

This section will feature some sample procedures, functions, and triggers designed around our database. The majority of the functionality of the procedures and functions will focus on some of the more base operations of a database, such as record insertion, deletion, or applying SQL functions, such as average, over some variable number of entries.

Procedure 1 - Add Employee:

As the name would imply, this procedure was intended to facilitate an easier way to add an entry to the employee relation. One key thing to note, is the how the procedure handles the creation. The procedure was designed from the viewpoint of the company hiring a new employee. As such, a company would not hire an employee endlessly. Therefore, this procedure also handles the creation of a works_for relation, using the current date for the starting date, and a null value for the ending (indicating they are actively working for said department. While possible, it was unnecessary to take into account an input for the dates, as past records should theoretically already be within the scope of the database, and would not need to be entered after the fact by the business.


```

1  -- Procedure 1 - Add an employee
2  CREATE OR REPLACE PROCEDURE addEmployee(
3    empID      Integer,
4    first      VARCHAR,
5    minit      VARCHAR,
6    last       VARCHAR,
7    addressID  Integer,
8    phone      Float,
9    dob        Date,
10   salary     Integer,
11   SSN        VARCHAR,
12   depNo      Integer)
13  LANGUAGE plpgsql
14  AS $$
15  BEGIN
16    SET Transaction READ WRITE;
17    insert into employee values(empID, first, minit, last, addressID,
18    phone, dob, salary, SSN);
19    insert into works_for values(depNo, empID, CURRENT_DATE, NULL);
20  EXCEPTION
21    WHEN others THEN
22    raise 'addEmployee() failed due to %', SQLERRM;
23  END;
24  $$;

```

Procedure 1 - Output:

Both of the relations that were created can be seen by the relatively simplistic queries after the procedure was called. The prime concern with the procedure, as can be seen by the multiple casts within the procedure call, was that postgres had a hard time deciphering type an input was, when used on the command line. Without the multiple casts seen here, the input were instead read as an 'unknown' type.

```

delivery=# CALL addEmployee(100, CAST('Billy' as VARCHAR), CAST('B' as VARCHAR), CAST('Joe' as VARCHAR), 30,
delivery=# CAST(6619894453 as FLOAT), to_date('2/25/1990', 'mm/dd/yyyy'), 250000, CAST('620856130' as VARCHAR), 10);
delivery=# SELECT * FROM employee WHERE Employee_ID = 100;
 employee_id | f_name | m_init | l_name | address_id | phone | dob | salary | ssn
-----+-----+-----+-----+-----+-----+-----+-----+-----
100 | Billy | B | Joe | 30 | 6619894453 | 1990-02-25 | 250000 | 620856130
(1 row)

delivery=# SELECT * FROM works_for WHERE Employee_ID = 100;
 department_id | employee_id | start_date | end_date
-----+-----+-----+-----
10 | 100 | 2019-04-07 |
(1 row)

```

Procedure 2 - Remove a purchase_order:

We also came at this procedure with a similar perspective of, what would this

mean within the context of the business. While the first procedure was the rough idea of hiring a new employee, this procedure was more along the lines of an employee making a mistake when creating a purchase order for a supplier and needing to delete said order. As such, the procedure itself is relatively simplistic, with the only thing of note being the dependency in creates that needed to be deleted before the purchase order was able to be deleted.

```
31  -- Procedure 2 - Remove a purchase_order (employee filed wrong order)
32  CREATE OR REPLACE PROCEDURE removePOOrder(INT)
33  LANGUAGE plpgsql
34  AS $$
35  BEGIN
36  »   delete from creates where Order_ID = $1;
37  »   delete from purchase_order where Order_ID = $1;
38  EXCEPTION
39  »   WHEN others THEN
40  »   RAISE 'Deleting purchase order % failed due to %', $1, SQLERRM;
41  END;
42  $$;
```

Procedure 2 - Output

As is seen in the before-after format of the included screenshot, the procedure properly handles the removal of the purchase order with the given order id. One thought we had that would be slightly more complex would be finding minimum id not in use for the insertions, such as in procedure 1, to account for situations like this, where some of the ids would be deleted and as such, not in use.

```

delivery=# SELECT * FROM purchase_order WHERE Order_ID=5;
 order_id | date_submitted | date_fulfilled | s_id
-----+-----+-----+-----
          5 | 2017-03-15     | 2017-04-07     | 12
(1 row)

delivery=# CALL removePOOrder(5);
CALL
delivery=# SELECT * FROM purchase_order WHERE Order_ID=5;
 order_id | date_submitted | date_fulfilled | s_id
-----+-----+-----+-----
(0 rows)

```

Function 1 - Average Lowest Salaries

This was likely the hardest to design of the procedures and functions covered thus far. Admittedly, a large part of the difficulty came from confusing some syntactic structures between Oracles and Postgres. Once we ironed out the differences in syntax, this was a relatively straight-forward function to create. The function by design takes in an integer value, then uses that to calculate the average salary from the N employees making the least amount of money.

```

48 CREATE OR REPLACE FUNCTION lowestAvgSalary(Integer)
49 RETURNS FLOAT AS $body$
50 DECLARE
51     avgSal FLOAT;
52 BEGIN
53     SELECT AVG(Salary)
54     INTO avgSal
55     FROM (SELECT * FROM employee ORDER BY Salary DESC) AS lowSalaries
56     LIMIT $1;
57     RETURN avgSal;
58 EXCEPTION
59     WHEN others THEN
60     RAISE 'lowestAvgSalary of % salaries failed due to %', $1, SQLERRM;
61 END;
62 $body$ LANGUAGE plpgsql;

```

Function 1 - Output

As evidenced once again by the accompanying screenshot, it can be seen that the function did run successfully. It should be noted that although the output value seems rather high, it is likely due to the tendency for the bulk of our salaries for employees to be a little unrealistically high.

```
CREATE FUNCTION
delivery=# SELECT * FROM lowestAvgSalary(10);
lowestavgsalary
-----
115736.416666667
(1 row)
```

Trigger 1 - Before Update Trigger

The first of the triggers we implemented was a trigger encompassing the logic of the before update function. In context of the business, this trigger references when a customer chooses to update a location provisioned by their contract. Since the location being provisioned is being updated, the routes for deliveries must also reflect that adjustment, and so this trigger updates the reference to that location within the routes before turning and updating the reference in the provisioned table.

```

68 CREATE OR REPLACE FUNCTION update_add()
69 RETURNS trigger AS
70 $body$
71 BEGIN
72     UPDATE routes
73     SET Next_Address = NEW.Loc_ID
74     WHERE Next_Address = OLD.Loc_ID;
75     RETURN NEW;
76 END;
77 $body$ LANGUAGE plpgsql;
78
79 CREATE TRIGGER update_nextAdd
80 BEFORE UPDATE OF Loc_ID
81 ON provisioned
82 FOR EACH ROW
83 EXECUTE FUNCTION update_add();

```

Trigger 1 - Output

As has been the trend, the output shown here was also done in the before after template to confirm the trigger was functioning correctly. In the before, we take a look at the only relation using the Next_Address of 4. In the after, it can be seen that although the update call was done on provisioned, the changes carried across to routes, where the Next_Address of path 8 is now the 7 from the update.

```

delivery=# SELECT * FROM routes WHERE Next_Address = 4;
 path_id | path_name | start_time | vehicle_num | first_address | next_address | scheduled_stops | employee_id
-----+-----+-----+-----+-----+-----+-----+-----
      8 | Cota      |          810 |    774514 |          17 |           4 |           19 |           1
(1 row)

delivery=# UPDATE provisioned SET Loc_ID = 7 WHERE Loc_ID = 4;
UPDATE 1
delivery=# SELECT * FROM routes WHERE Next_Address = 7;
 path_id | path_name | start_time | vehicle_num | first_address | next_address | scheduled_stops | employee_id
-----+-----+-----+-----+-----+-----+-----+-----
      8 | Cota      |          810 |    774514 |          17 |           7 |           19 |           1
(1 row)

delivery=#

```

Trigger 2 - Cascaded Deletion Trigger

This trigger from a surface value, seemed intended to be a replacement for a

cascaded deletion. For our sample trigger shown below, we implemented a trigger if a client wished to cancel all contracts and remove themselves from our system. Here's hoping for the business it never comes to that, but we assumed that could be one eventuality.

```
89 CREATE OR REPLACE FUNCTION fun_delete_client()
90 RETURNS trigger
91 AS $body$
92 BEGIN
93     DELETE FROM provisioned
94     WHERE Contract_ID = (SELECT contract_id FROM contracts c
95     WHERE c.Client_ID = OLD.Client_ID);
96     DELETE FROM contains
97     WHERE Contract_ID = (SELECT contract_id FROM contracts c
98     WHERE c.Client_ID = OLD.Client_ID);
99     DELETE FROM contracts
100     WHERE Client_ID = OLD.client_id;
101     RETURN OLD;
102 END;
103 $body$
104 LANGUAGE plpgsql;
105
106 CREATE TRIGGER delete_client
107 BEFORE DELETE
108 ON clients
109 FOR EACH ROW
110 EXECUTE FUNCTION fun_delete_client();
```

Trigger 2 - Output

In order to demonstrate the full scope of the functionality for this trigger, instead of looking at the clients, the output instead looked at one of the dependencies of the client's primary key. In this particular case, the contract was analyzed, as the contract for the client was deleted before the client associated with that primary key was deleted. Without this preliminary step, it wouldn't be possible to delete the client without a cascade command.

```

delivery=# SELECT * FROM contracts WHERE client_id = 5;
 contract_id | frequency | start_date | end_date | client_id
-----+-----+-----+-----+-----
          24 |         5 | 2017-06-27 | 2018-12-06 |         5
(1 row)

delivery=# DELETE FROM clients where client_id = 5;
DELETE 1
delivery=# SELECT * FROM contracts WHERE client_id = 5;
 contract_id | frequency | start_date | end_date | client_id
-----+-----+-----+-----+-----
(0 rows)

```

Trigger 3 - Update Contract Locations in a View

This trigger is designed to interact with a view in postgres. Given that a view is reformatting some of the base tables of a database, it doesn't really make sense to interact with a different representation of said data. As such, postgres does restrict you from attempting to insert or update directly on a view. With that in mind, the following view was created from four of the relations to test the accompanying trigger with. For this trigger the idea was if a customer ever wanted to make a change to where a current contract was being delivered to. With that idea in mind, the trigger allows for updates to the location of an associated contract.

View Creation:

```

68 CREATE VIEW activeContracts AS
69 » SELECT c.Client_ID as id,
70 » c.F_Name || ' ' || c.M_Init || ' ' || c.L_Name as name,
71 » o.Contract_ID as contract,
72 » o.Frequency,
73 » o.Start_Date,
74 » l.City_Name || ', ' || l.State || ', ' || l.Zip_Code as location,
75 » l.Loc_ID
76 » FROM (clients c INNER JOIN contracts o ON c.Client_ID = o.Client_ID)
77 » INNER JOIN (locations l INNER JOIN provisioned p ON p.Loc_ID = l.Loc_ID)
78 » ON o.Contract_ID = p.Contract_ID
79 » ORDER BY id, contract;

```

Trigger and Associated Function Creation:

```
81 CREATE TRIGGER update_location
82 INSTEAD OF UPDATE on activeContracts
83 for each row
84 EXECUTE PROCEDURE insert_contract_location();
85
86 CREATE OR REPLACE FUNCTION insert_contract_location()
87 » RETURNS trigger AS
88 $func$
89 BEGIN
90 » UPDATE provisioned
91 » » SET Loc_ID = NEW.Loc_ID
92 » » WHERE Loc_ID = OLD.Loc_ID;
93 » RETURN NEW;
94 END;
95 $func$ LANGUAGE plpgsql;
```

Trigger 3 - Output

The following screen depicts a before action of one particular row from the view. In between them, a singular update call can be seen running on the location id. After the update call, the location id and the location that is tied to it are both seen to be updated, as the update took place within the original relation containing the link to the relation. In this particular case, the relation that was updated was the provisioned relation. For the pure intent of a sanity check, we have included an alternative look at the record being changed within the original relation, while using the same update call on the view.

View Update:


```

delivery=# SELECT * FROM activeContracts WHERE id = 1;
 id |      name      | contract | frequency | start_date |      location      | loc_id
-----+-----+-----+-----+-----+-----+-----
  1 | Gustav B Ganelea |      20 |         2 | 2018-10-06 | Montana, Tallahassee, 28364 |    13
(1 row)

delivery=# UPDATE activeContracts SET loc_id = 2 WHERE id = 1;
UPDATE 1
delivery=# SELECT * FROM activeContracts WHERE id = 1;
 id |      name      | contract | frequency | start_date |      location      | loc_id
-----+-----+-----+-----+-----+-----+-----
  1 | Gustav B Ganelea |      20 |         2 | 2018-10-06 | Washington, Breckenridge, 56681 |     2
(1 row)

delivery=#

```

Relation after View Update:

```

delivery=# SELECT * FROM provisioned WHERE Contract_ID = 20;
 contract_id | loc_id
-----+-----
          20 |     13
(1 row)

delivery=# UPDATE activeContracts SET loc_id = 2 WHERE id = 1;
UPDATE 1
delivery=# SELECT * FROM provisioned WHERE Contract_ID = 20;
 contract_id | loc_id
-----+-----
          20 |     2
(1 row)

```

4.3 PL/pgSQL-Like Languages in Microsoft SQL Server, MySQL and Oracle DBMS

While all of data, procedures, and functions described within the documentation was implemented using PostgreSQL, it's still very useful to analyze some of the differences seen in some of the other popular alternatives that exist, as well as the accompanying syntax. Among the market, some of the most popular alternatives to

PostgreSQL include Microsoft SQL Server, MySQL, and Oracle. Each of these features their own benefits and drawbacks, which will be looked at in this section.

4.3.1 Microsoft SQL Server: T-SQL

Microsoft SQL Server uses the procedural language T-SQL, or Transact-SQL, to operate its Database Management System. While on the surface, most of the functionality is shared between the various procedural languages, T-SQL features a few unique traits in comparison to its competitors. Some of the more notable in contrast with languages such as MySQL would be how it handles try-catch blocks in context of a procedure. Unlike MySQL, T-SQL allows for multiple try-catch blocks within the procedure itself, whereas MySQL has to handle it within a separate section altogether.

Another notable difference is the relative ease with which T-SQL functions can return tables or scalar values. Admittedly, this is possible in languages such as Oracle, but it is far more difficult to accomplish. On the topic of loops, unlike Oracle, T-SQL does not have for-loops. T-SQL instead has to handle all of the loops with while loops or some basic loops. The last notable difference of T-SQL from its peers is its ability to encrypt the text of procedures and restricting user permissions.

T-SQL Select Statement Syntax

```
<SELECT statement> ::=  
    [ WITH { [ XMLNAMESPACES , ] [ <common_table_expression> [,...n] ] } ]  
    <query_expression>  
    [ ORDER BY { order_by_expression | column_position [ ASC | DESC ] }  
    [,...n ] ]  
    [ <FOR Clause>]
```

```

[ OPTION ( <query_hint> [ ,...n ] ) ]

<query_expression> ::=

{ <query_specification> | ( <query_expression> ) }

[ { UNION [ ALL ] | EXCEPT | INTERSECT }

    <query_specification> | ( <query_expression> ) [...n] ]

<query_specification> ::=

```

T-SQL Loop Statement

```

WHILE Boolean_expression

    { sql_statement | statement_block | BREAK | CONTINUE }

```

T-SQL Trigger Creation

```

CREATE [ OR ALTER ] TRIGGER [ schema_name . ]trigger_name

ON { table | view }

[ WITH <dml_trigger_option> [ ,...n ] ]

{ FOR | AFTER | INSTEAD OF }

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }

AS { sql_statement [ ; ] [ ,...n ] [ ; ] > }

```

```

<dml_trigger_option> ::=

    [ EXECUTE AS Clause ]

```

4.3.2 MySQL

MySQL is the most widely used standard database system for websites that store extremely large amounts of data and users such as most social media sites

(Twitter, Facebook, Wikipedia etc.). MySQL is the most basic version compared to other more robust RDBMSs'. Many of its features are not unique and are present in other DBMSs' what differentiates it the most would be its lack of features. For instance MySQL doesn't follow complete SQL standards for a portion of its functionality, this includes a lack of foreign key references. Another limitation would be that triggers can't correspond to views and that these triggers are limited by a single action, such as a trigger BEFORE and AFTER an event on the same table.

Syntax for Procedures, Functions & Triggers:

CREATE

[DEFINER = user]

PROCEDURE sp_name ([proc_parameter[,...]])

[characteristic ...] routine_body

CREATE

[DEFINER = user]

FUNCTION sp_name ([func_parameter[,...]])

RETURNS type

[characteristic ...] routine_body

CREATE

[DEFINER = user]

TRIGGER trigger_name

trigger_time trigger_event

ON tbl_name FOR EACH ROW

[trigger_order]

trigger_body

Syntax for Selective & Repetitive Statements:

SELECT

[ALL | DISTINCT | DISTINCTROW]

[STRAIGHT_JOIN]

select_expr [, select_expr ...]

[FROM table_references

[WHERE where_condition]

[GROUP BY {col_name | expr | position}, ... [WITH ROLLUP]]

[HAVING where_condition]

[ORDER BY {col_name | expr | position}

[ASC | DESC], ... [WITH ROLLUP]]

[LIMIT {[offset,] row_count | row_count OFFSET offset}]

CASE case_value

WHEN when_value THEN statement_list

[WHEN when_value THEN statement_list] ...

[ELSE statement_list]

END CASE

[begin_label:] LOOP

statement_list

END LOOP [end_label]

[begin_label:] WHILE search_condition DO

statement_list

END WHILE [end_label]

4.3.3 Oracle DBMS: PL/SQL

The Oracle Database Management System uses the procedural language PL/SQL. This language is a combination of SQL along with the procedural features of programming languages. SQL is the standard database language so PL/SQL is strongly integrated with SQL and supports both static and dynamic SQL. PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database. The applications written in PL/SQL are fully portable while providing support for developing web application and server pages. These are only a few advantages and features of PL/SQL. A main difference between PL/SQL and T-SQL would be the way they handle variables, stored procedures, and built-in functions. A difference between PL/SQL and MySQL would be that PL/SQL is a SQL extension while MySQL is a database itself that uses the SQL language. Below are some examples of how the syntax would look using PL/SQL.

Syntax for Procedures, Functions & Triggers:

Function

```
CREATE [OR REPLACE] FUNCTION function_name  
[(parameter_name [IN | OUT | IN OUT] type [, ...])]  
RETURN return_datatype  
{IS | AS}  
BEGIN  
< function_body >  
END [function_name];
```

Procedure

```
CREATE [OR REPLACE] PROCEDURE procedure_name  
[(parameter_name [IN | OUT | IN OUT] type [, ...])]  
{IS | AS}  
BEGIN  
< procedure_body >  
END procedure_name;
```

Trigger

```
CREATE [OR REPLACE] TRIGGER trigger_name  
{BEFORE | AFTER | INSTEAD OF }
```

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col_name]

ON table_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Syntax for Selective & Repetitive Statements:

For-Loop

FOR loop_counter IN [REVERSE] lowest_number..highest_number

LOOP

{...statements...}

END LOOP;

While-Loop

WHILE condition

LOOP

{...statements...}

END LOOP;

Repeat Until Loop

LOOP

{...statements...}

EXIT [WHEN boolean_condition];

END LOOP;

IF-THEN-ELSE Statement

IF condition THEN

{...statements to execute when condition is TRUE...}

LSE

{...statements to execute when condition is FALSE...}

END IF;

CASE Statement

CASE [expression]

WHEN condition_1 THEN result_1

WHEN condition_2 THEN result_2

...

WHEN condition_n THEN result_n

ELSE result

END

Phase 5: Graphical User Interface

The last part of the phase involves creating some type of graphical user interface of our choosing. This GUI will be specifically for a certain type of user groups which will be detailed. Unique as well as standardized features of Postgres will be described in how they were able to aid in the creation of the database as well as GUI. These features will be detailed with a photo and summary of what the feature is handling.

5.1 Functionalities and User Groups of the GUI

Application

Creating a database for a frozen food delivery would mean that the focus would be on the deliveries and the clients. Because of this, the focus of my user groups will be those who are directly and indirectly involved with the routes and customers. Descriptions of the user groups will detailed along with their purpose.

5.1.1 User Group Descriptions

Route Overseers

Route Overseers is the user group which keeps track of what is currently occurring for routes that are active, that is to say where drivers are currently on a delivery.

Routine Activities:

- View activities of on-going routes
- Create addendums for route activities
- Contact clients through their preferred method

Depot Managers

Depot Managers is the user group which would be most involved with the database. These managers are involved in the creation of new routes which is a rare occurrence because of the convenience of appending stops to a route.

Routine Activities:

- Append routes
- Assign employees to a route
- View every drivers routes as well as available vehicles
- Append stops to a route with full details
- View overall route report details

Drivers

Drivers are the most common type of user group. Drivers are the currently employed workers who are assigned a route and vehicle. Their level of access to the database is similar to a Depot Manager, however, they have limited privileges on the

types of changes.

Routine Activities:

- Append stops
- View their currently assigned routes
- Append customers into the database
- View route manifests
- Append activities for a stop

5.1.2 Tables and Views Descriptions

Referencing back onto Phase IV, we've utilized some of the features PL/SQL provides in order to allow these user groups easy navigation through the database. A significant portion of the navigation lies through the views which will be detailed further below.

View: drivers

This view is a full outer join between tables Routes and Employee. It is displayed as `full_name, Employee_ID`. The view displays all employees who are listed in the routes table, which is to say that these employees are drivers. This view is used most of the time, it's great functionality is that it appends the employees separated name attributes and joins them into a single column. This is view is most often accessed by a dropdown menu list.

View: driversRoutes

This view is a simple full inner join between tables Routes and Employee. It grabs necessary information so that a table will display all routes that an employee is currently assigned to.

View: routeMan

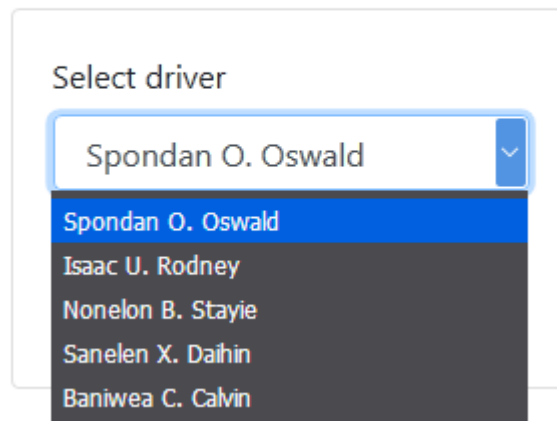
This view is the most complex as it performs an inner join on five different tables. It is displayed as any relevant information of a single route which would include, the stops, location information, clients, etc. A great feature this view provides is that users won't be making a mistake in inserting an already established primary key. Although it isn't as frequently seen as the view drivers, the functionality it provides for the database is invaluable.

5.2 Programming

The previous functionalities have been given a description on what they are and how they function in the database. Now we will showcase how they function in a typical use case scenario.

5.2.1 Server-Side Programming

Choose a Driver



Select driver

Spondan O. Oswald

- Spondan O. Oswald
- Isaac U. Rodney
- Nonelon B. Stayie
- Sanelen X. Daihin
- Baniwea C. Calvin

The forefront of our database is presented to us as the view `drivers` showcases which employees are currently assigned to a route.

Nonelon B. Stayie's Routes

Driver	Route Name	Vehicle
Nonelon B. Stayie	Merchant	627712
Nonelon B. Stayie	Tovar	417352
Nonelon B. Stayie	Fenton	710080

After selecting an employee the view `driversRoutes` populates just below the Driver list. This view grabs necessary information to inform a user what routes they are assigned to with the provided vehicle.

Driver

Spondan O. Oswald

Route Name

Vehicle

310440

310440

479697

196655

Insert Route

Just below the routes managers are able to create a route. Again we use the view `drivers` to present a dropdown list of the employees who are assigned to a route. We can also see another list for all the vehicles that are available for use. The same vehicle can be used on many different routes.

Edit a Route

Route Name

Merchant

▼

Edit Route

Below the route insertion form, we use `driversRoutes` to showcase a dropdown list of the routes the employee is currently assigned to.

Merchant's Stops

Stop	Stop Path	Street	City	State	Zip Code	Long	Lat	Frequency	Client
1	10	794 Fratie Mews	Batavia	New Jersey	35770	178.38631	-54.506	177	Gustav B. Ganelea
2	10	752 Tispen Vale	Darlington	New Jersey	92290	-167.44232	172.82288	7	Bonnie G. Nathaniel
3	10	138 Senolan Mead	Bay Head	New Jersey	38755	68.92246	23.95493	5	Jasmin Y. Donese
4	10	236 Fresie Mead	Lake Geneva	New Jersey	46049	78.46483	26.55073	14	Nensen T. Funuse
5	10	480 Spanden Rise	Batavia	New Jersey	65531	-26.45152	-22.68526	31	Conton Q. Antonia
6	10	549 Stutie Mead	Bay Head	New Jersey	65772	123.94066	-167.45464	60	Seymour D. Monehe
7	10	853 Penelope Crescent	Darlington	New Jersey	31781	-170.41735	-63.81688	1	Benonee F. Diblin
8	10	814 Eren Place	Lake Geneva	New Jersey	38183	6.64605	2.90659	2	Henilon Y. Stedie

Stop

Contract Location Number

Stop Path

200

10

Street

City

417 Nonilea Vale

Batavia

State

Zip Code

Longitude

New Jersey

29069

84.13115

Latitude

Frequency

Client

-75.68758

5

Lonene F. Nenalonherhie

Add Stop

In this menu we utilized `manRoute` to grab all the information shown. We have all the stops of a route, the routes path, the location, longitude and latitude of the client, and the daily interval of when the stop should be visited. We are also able to append all the information that has been mentioned. Default values are populated where necessary.

5.2.2 Middle-Tier Programming

The functionality and purposes of the code have been explained but not shown; this section would be showing what the back end is seen as.

Server Hosting

```
<?php
$db=pg_connect("host=localhost dbname=delivery user=delivery password=frozen");
if(!$db)
{
    echo "connection failed";
}
?>
```

All of the front end that was seen is not a program. The GUI was created utilizing PHP 10. A program called XAMPP was utilized to host the server for us to navigate. In the code above, a separate file name connect.php held the following connection information. We include this file on all pages so that we may retain the connection without having to increase the file size of our document pages.

Database Querying

```
<form action="change-routes.php" method="post">
  <div class="form-group">
    <label for="sel1">Select driver</label>
    <select class="form-control" name="employee_name">
      <?php // Performing SQL query

      $result = pg_query($db, "SELECT * FROM drivers");
      if (!$result)
      {
        echo "Failed query.\n";
        exit;
      }

      while ($row = pg_fetch_assoc($result))
      {
        // $employee_name = $row['full_name'];
        // $employee_id = $row['Employee_ID'];
        echo '<option value="'. $row['full_name']. '">'. $row['full_name']. '</option>';
      }
      $employee_name = $_POST['employee_name'];
    </select>
    <div class="text-center">
      <br>
      <input type="submit" class="btn btn-primary" name="action" value="Choose Driver">
    </div>
  </div>
</form>
```

The above is what we are first presented with when interacting with the database, the driver selection. A form is given which would loop through the query results, the drivers, that we have sent to the database.

```

if ($_POST['action'])
{
    if ($_POST['action'] == 'Add Stop')
    {
        $Stop_Address = $_POST['Stop_Address'];
        $Stop_Address = sani($Stop_Address);
        settype($Stop_Address, "integer");

        $Stop_Path = $_POST['Stop_Path'];
        $Stop_Path = sani($Stop_Path);
        settype($Stop_Path, "integer");
    }
}

```

Once we submit a form with parameters, in this case once we add a stop, all the values given are then sanitized and set to their correct values as stated in the table creation of these values. This is to prevent any failed queries that a user may make.

```

CREATE SEQUENCE IF NOT EXISTS pathInc START 16;

CREATE TABLE IF NOT EXISTS routes (
    Path_ID integer primary key not null DEFAULT nextval('pathInc'),

```

Another instance of utilizing postgresql's features would be sequences. Sequences allows for a set value an attribute should start with that increments whenever a parameter is not called in. Sequences are useful for primary integer keys since queries would have a more difficult time failing from the fail-safe increments.

5.3 Learning and Implementing

Every information has been explained and detailed throughout the document. We will briefly go over what has been covered.

Data Collection and Conceptual Database Design

We have analyzed what a specific companys components may be structured as and how its entities relate to another.

Conceptual Collection and Logical Database

Creating a visualization model of how a database may be connected and formed. This is done by a thorough examination of the relationships between the many relations.

Physical Database Design

A DBMS may differ greatly between it's features and slightly different functionality. Functionality such as views and sequences are signifcant for speeding up the back end implementation.

Software Interface

A good GUI application should never get in the way of one's work flow. It should be easily read and have some logical structure.

5.4 Survey

Outcome	Mitchell Alvarado	Sean Fontes
An ability to analyze a problem, and identify and define the computing requirements and specifications appropriate to its solution.	7	10
An ability to design, implement and evaluate a computer-based system, process, component, or program to meet desired needs, An ability to understand the analysis, design, and implementation of a computerized solution to a real-life problem.	5	9
An ability to communicate effectively with a range of audiences. An ability to write a technical document such as a software specification white paper or a user manual	7	9
An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems in a way that demonstrates comprehension of the tradeoffs involved in design choices.	3	10