# UPServiceDB

§

*A database system for a multinational package delivery and supply chain management company.*

developed by ***Tiziano Franza***.

# 1. Requirements Analysis

In the preliminary phase of a database software development, it is necessary to gather both the functional and non-functional requirements in order to spot all the obstacles to be overcome with correctness and completeness. It is also important to take into consideration all the static and dynamic aspects of a database that can be directly translated as data and operations on data. Here the ***Requirements Specification*** are presented:

*"UP is a multinational package delivery and supply management company. UP prides itself on having up-to-date information on the processing and current location of each shipped item.*

*Shipped items are the heart of the UP product tracking information system. Shipped items can be characterized by item number (unique), weight, insurance amount, destination, and final delivery date. In addition, parcels are also characterized by their dimension.*

*Shipped items are received into the UP system at a single retail center. Retail centers are characterized by their type, unique ID, and address.*

*Shipped items make their way to their destination via one or more standard UP transportation events (i.e., flights, truck deliveries). These transportation events are characterized by a unique schedule number, a type (e.g., flight, truck), and a delivery route.*

*The delivery route is based on the position of the used mean of transport: when items are on board of a means of transport, they take the same position of the means of transport."*

In this specific context several queries were considered:

- Op1. Collect a new item (5000 times per day).
- Op2. Track the item (5 times per item).
- Op3. Add a new transportation event (5 times per item).
- Op4. Add a position for a means of transport (once per minute per vehicle).
- Op5. Produce a weekly report, with the average delivery time and the number of accomplished deliveries (once a week).

Furthermore it is necessary to take into consideration that *"there are 1000 vehicles"* available for the transportation events.

The main aim of this phase is to create a clear and solid structure of information starting from the just provided requirements. The hidden information is difficult to code in the way they are provided because these are expressed in natural language that is naturally ambiguous and subject to free interpretation.

The process of ***Requirements Analysis*** includes a phase of linearization, consisting in the splitting of the requirements in simple paragraphs, choice of the entities with the appropriate level of abstraction, adaptation of the same semantic style, unification of synonyms and elicitation of the references between entities.

For this purpose a table called ***Glossary of Terms*** is generated. In this glossary every term is explained with the help of a short description, attributes, possible elicited synonyms and relationships with the other terms:

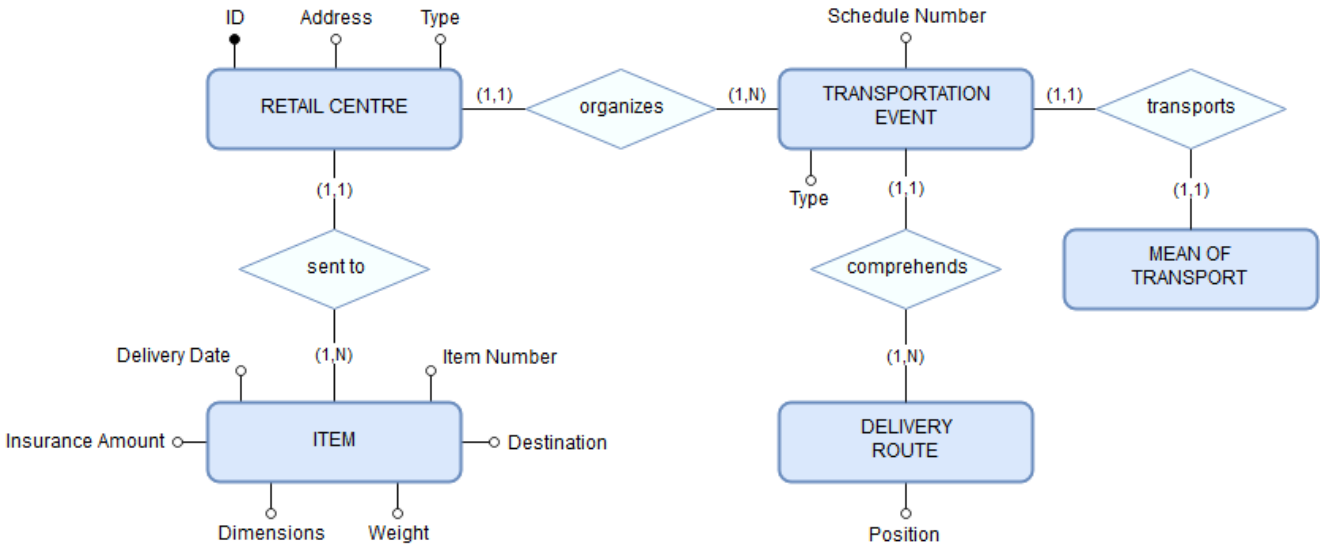
| TERM | DESCRIPTION | RELATIONSHIPS |
|---|---|---|
| Item | The object that has to be shipped. It can be delivered with the help of a *Mean of Transport* along different *Transportation Event*s.<br>Synonyms: parcel.<br>Attributes: item number, weight, insurance amount, destination, final delivery date, dimensions. | Retail Center. |
| Retail Center | The place where *Item* is received to be dispatched for the shipping by a *Mean of Transport* through *Transportation Event*s.<br>Attributes: type, ID (unique), address. | Item, Transportation Event. |
| Transportation Event | Event that is organized with the help of the *Retail Center* and involves a *Delivery Route* to make the shipment.<br>Attributes: schedule number, type (e.g. flight, truck). | Retail Center, Delivery Route. |
| Delivery Route | Path to follow to reach the destination with the *Mean of Transport* to deliver the *Item*.<br>Attributes: position. | Transportation Event, Mean of Transport. |
| Mean of Transport | It follows a *Delivery Route* to accomplish the *Transportation Event*.<br>Synonyms: vehicle. | Delivery Route. |

Regarding the non-functional requirements, the system object of development of this project needs to satisfy certain properties:

# 2. Design of the Database

The Requirements Analysis revealed the candidate entities to be considered in the database system in order to accomplish the requested queries that need to be carried out by the client during the usage of the system.

Following we can observe an early representation, called **Conceptual Model**, that fully reflects the Requirements Analisys just carried out in the previous section:




In order to make the structure thicker new entities were added:

- ***DEALER***, which represents the sender and the receiver of the *ITEM*. This entity has the following attributes:
  - *ID*, indicates the identifier of the *DEALER*.
  - *Name*: indicates the name of the *DEALER* in terms of name and surname.
  - *Address*: indicates where the *DEALER* lives. It is composed of a *ZIP* code, a *Country*, a *City* and a *Street*.
  - *Phone Number*: indicates the *DEALER*'s phone number, in case of problems.
  - *Mail*: indicates the *DEALER*'s email contact, in case of problems.
  - *Kind*: specifies if the *DEALER* is the sender or the receiver of the *ITEM*.

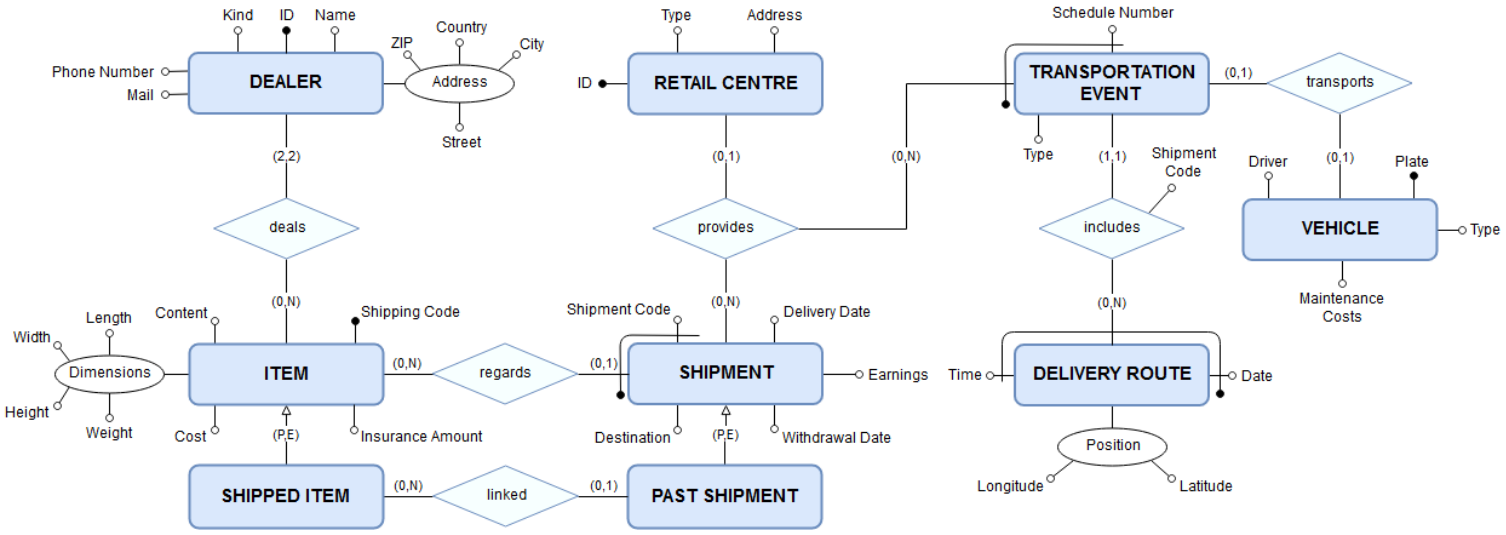
- ***SHIPMENT***, which represents the operation of shipping an *ITEM*. This entity has the following attributes:
  - *Shipment Code*, indicates the identifier of the *SHIPMENT.*
  - *Delivery Date*, moved from the entity *ITEM*.
  - *Withdrawal Date*, indicates the date when the *ITEM* will be taken from the sender.
  - *Earning*, earning of the company by the shipment.
  - *Destination*, moved from the entity *ITEM*.

The entity *Mean of Transport* was renominated *Vehicle*.

Hence, to further improve the representative quality of the model it was useful to adopt specific patterns, also known as ***Project Patterns***, that are common solutions to frequent problems in the context of projects and that have already been solved. The adopted patterns are listed here following:

- Part-Of: applied to the relationship
  - *regards*, towards the entity *Shipment*, since its existence directly depends on the existence of an *Item* and a couple of instances of *Dealer*.
  - *organizes*, towards the entity *Transportation Event*, since its existence directly depends on the existence of a *Retail Centre*.
  - *comprehends*, towards the entity *Delivery Route*, since its existence direcly depends on the existence of a Transportation Event.
- Archiving a Concept: applied to the entity
  - *Shipment*, to keep track of all the shipments up to the last six months.
  - *Item*, to keep track of all the items that have been shipped up to the last six months.

The result of the refinement of the raw Conceptual Model is the following:




In this work we have considered for hypothesis that the database system in development will have a lifecycle of 10 years, hence it has to be able to contain the information of the same period of time.

In order to fully understand an idea of the dimensions of the database, the ***Volumes Table*** gets drawn up. In this table the approximate maximum number of tuples for each entity or relationship is analyzed.

| CONCEPT | TYPE | VOLUME |
|---|---|---|
| Dealer | E | 800.000 |
| deals | R | 7.000 |
| Item | E | 3.500 |
| regards | R | 3.500 |
| Shipment | E | 20 |
| provides | R | 100 |
| Retail Centre | E | 200 |
| Transportation Event | E | 100 |
| includes | R | 14.400 |
| Delivery Route | E | 14.400 |
| transports | R | 100 |
| Vehicle | E | 100 |
| Shipped Item | E | 1.600.000 |
| linked | R | 1.600.000 |
| Past Shipment | E | 10.000 |

Therefore to fill the Volumes Table it was useful to foresee an approximative estimate for the quantities in play in the period of 10 years.

Here's following a list of the assumptions for the considered value averages to generate the values in the table:

- 1 vehicle has a capacity of 200 items on average.
- 1 week on average is necessary for each item to be shipped to destination.
- 1 transportation event lasts 24 hours on average.
- Operation 4 is applied once per 10 minutes.
- The number of items collected per day is 500 instead of 5000.
- The number of available vehicles is 100 instead of 1000.

Supplied with the Volumes Table is the ***Operations Table***, that allows us to have insights about the operations to be carried out and the relative frequencies.

The Volumes Table allows us to obtain an approximation of the dimensions to be used to evaluate the efficiency of the operations.

| OPERATION | TYPE | FREQUENCY |
|---|---|---|
| Op.1 | Interactive | 500/day |
| Op.2 | Interactive | 14/day |
| Op.3 | Interactive | 14/day |
| Op.4 | Interactive | 100/hour |
| Op.5 | Batch | 1/week |

In this context studying the possible addition of redundancy reveals particularly useful to increment the runtime performance of the information retrieval. The step in which these features are analyzed is called ***Redundancy Analysis***.

The candidate redundancy is the ***Shipment Code*** attribute on the entity *Transportation Event*. It contains the shipment code for the specific shipment, equal to the omonym attribute on the entity *Shipment*.

This redundancy has effect on Operation 2: "*Track the item*".

| ACCESSES WITHOUT REDUNDANCY | | | |
|---|---|---|---|
| **CONCEPT** | **CNSTR** | **ACC** | **TYPE** |
| Item | E | 1 | R |
| regards | R | 1 | R |
| Shipment | E | 1 | R |
| provides | R | 5 | R |
| Transportation E. | E | 5 | R |
| includes | R | 120 | R |
| Delivery Route | E | 120 | R |

| ACCESSES WITH REDUNDANCY | | | |
|---|---|---|---|
| **CONCEPT** | **CNSTR** | **ACC** | **TYPE** |
| Item | E | 1 | R |
| regards | R | 1 | R |
| Transportation E. | E | 5 | R |
| includes | R | 120 | R |
| Delivery Route | E | 120 | R |

Here's following the number of total accesses:

- N. accesses without redundancy: 1+1+1+5+5+120+120 = **253**.
- N. accesses with redundancy: 1+1+5+120+120 = **247**.

Adding the redundancy, response time of the system are lowered by 2.5%. Since the earned time is very little, it was not added to the conceptual model, even because this would means a synchronization between the *Transportation Event* entity and the *Shipment* entity that would result in a worse performance considering the high frequency of addition of an item (500/minute).

The development strategy adopted for the creation of the Conceptual Model, that we have already shown, is called ***Top-Down Strategy*** and consists in the proposition of a initial raw model, that describes the specifications with few abstract concepts. Therefore the raw model is improved for successive refinements, that hyphen the level of detail of the various present concepts, until a better schema is obtained as result.

In our specific case, first of all we considered a conceptual model directly obtained by the requirements specification, hence from the Glossary of Terms. Then we have transformed during the whole reconstruction phase through the usage of Design Patterns and the adoption and modification of identifiers, in order to make a much solid structure of the system.

The next step was the optimization of the system throughout the search of candidate redundancies candidate and the acceptance/rejection of those on the basis of an deep analysis considering the number of accesses with or without the redundancies themselves and a consequent qualitative evaluation considering the operations influenced by them. Since there are no further improvements with redundancies, the final model remains the same.

# 3. Implementation of the Database

The next step is the construction of a model that could act as an interface to the real implementation of the database system. This is called ***Logical Model*** and condenses the conceptual model under the shape of tables representing entities and relationships with the corresponding attributes and identifiers (that in this model we will call as *keys*).

The logical model developed for this system is the following:

**ADDRESS** (*zip NUMBER, country STRING, city STRING, street STRING*).

**DIMENSIONS** (*width NUMBER, length NUMBER, height NUMBER, weight NUMBER*).

**POSITION** (*latitude NUMBER, longitude NUMBER*).

**DEALER** (*id STRING, kind STRING, name STRING, phone STRING, mail STRING, address ADDRESS*).

**SHIPMENT** (*shipmentCode STRING, destination STRING, withdrawalDate DATE, deliveryDate DATE, earnings NUMBER*).

**PASTSHIPMENT** (*shipmentCode STRING, destination STRING, withdrawalDate DATE, deliveryDate DATE, earnings NUMBER*).

**CENTRE** (*id NUMBER, kind STRING, address STRING*).

**VEHICLE** (*driver STRING, plate STRING, kind STRING, costs NUMBER*).

**ITEM** (*shippingCode STRING, content STRING, insurance NUMBER, dimensions DIMENSIONS, seller REF DEALER, buyer REF DEALER, shipment REF SHIPMENT*).

**PASTITEM** (*shippingCode STRING, content STRING, insurance NUMBER, dimensions DIMENSIONS, seller REF DEALER, buyer REF DEALER, shipment REF SHIPMENT,pastShipment REF PASTSHIPMENT*).

**EVENT** (*id NUMBER, kind STRING, vehicle REF VEHICLE, shipment REF SHIPMENT, centre REF CENTRE*).

**ROUTE** (*id NUMBER, stamp DATE, position POSITION, event REF EVENT*).

The implementation of the Logical Model has been developed in PL/SQL in concomitance with the kind of utilized DBMS, that is Oracle Database 11g (release 2).

The exploited paradigm was the object oriented one without external referential constraint but using OIDs to reference the other tables in the database. Moreover no primary keys can be found while those indicated in the Logical Model were constrainted to unique values in order to maintain the unicity property of the fields.

In order to evaluate the performances of the system during the execution of the operations, the step of early-generated typed ***Table Population*** was fundamental for this purpose at the same time taking into consideration the supposed volumes. This operation can be condensed in adding random values for dates, numbers and strings but, in order to make results for the requests the most realistic possible, a ***Vocabulary*** from where choosing randomly the previously-added strings was implemented.

It was ideated to contain the following semantic keys:

- *names*, containing 1000 voices for first names for people, to generate the field *name* of the table DEALER and the field *driver* of the table VEHICLE;
- *surnames*, containing 1000 voices for surnames for people, to generate the field *name* of the table DEALER and the field *driver* of the table VEHICLE;
- *cities*, containing 1000 voices for denomination of english cities, to generate the field *city* of the table DEALER.ADDRESS;
- *countries*, containing about 300 voices for countries, to generate the field *country* of the table DEALER.ADDRESS, the field *destination* of the table SHIPMENT and the field *destination* of the table PASTSHIPMENT.
- *streets*, containing more than 1000 voices for streets, to generate the field *street* of the table DEALER.ADDRESS.
- *vehicles*, containing 5 different voices for mean of transport types, to generate the field *kind* of the table VEHICLE.

| TABLE | POPULATION TIME (sec) |
|---|---|
| Dealer | 210,12 |
| Shipment | 0.54 |
| PastShipment | 2.05 |
| Centre | 0.55 |
| Vehicle | 0.52 |
| Item | 1.55 |
| PastItem | 494.73 |
| Event | 0.56 |
| Route | 4.06 |
| **Total** | **714.68** |

The voices contained inside the lists associated to the previous semantic keys, have been scraped from the web (sources: https://*www2.census.gov*, www.*coffeeghost.net* , *www.searchify.ca*), fixed to 1000 elements and standardized by a simple script written in Java (*VocabStandardizer*) for the specific kind of data structure implemented in the database. Each voice was assigned an incrementing value that identifies them and the relative value.

Each table is populated by a specific procedure, named *populate<nameTable>* and that carry out their job prefixed batch size (1000 like the size of dictionary) and a number of batches. The cardinality of tables has a correspondence to the values indicated in the Volumes Table and the number of batches is calculated dividing the volume per the batch size.

The choice of having a batchSize value is due to the fact that the cardinality of the tables is superior to the cardinality of the vocabulary in a lot of cases. However the values contained in the Volume Tables do not correspond to the exact cardinalities but have an approximation high enough to guarantee an adequate simulation of the database system after a period of 10 years of life.

This step of population is followed up by the step of the **Implementation of Operations**. Tipically in a database the main operations to be carried out are four:

- *Create Operations*: for each table a procedure to insert tuples was created, trying to reduce the number of needed inputs and automatizing the insertion of the remaining fields. The only tables that cannot be created are the historic ones, respectively PASTITEM and PASTSHIPMENT. Their fillment is strictly automatic and will be explained further in the document.

- *Read Operations*: for each table a procedure to read tuples was created, however procedure do not return values. Functions return values but no more than a tuple at a time, so each table has a corresponding *temp<nameTable>* with the same shape, in which read values are stored. The body of these operations is parametrized so that it is also possible to fill even partially the fields and return a wider set of information (even select all the tuples selecting anything). The null string value is the space while the null number value is -1.
- *Update Operations*: for each table a procedure to update tuples was created. The body of these operations is parametrized so that it is also possible to fill even partially the fields and return a wider set of information (even select all the tuples selecting anything). The null string value is the space while the null number value is -1. The only tables that cannot be created are the historic ones, respectively PASTITEM and PASTSHIPMENT. Their update is strictly automatic and will be explained further in the document.
- *Delete Operations*: for each table a procedure to delete tuples was created. The only input requirement for deletion is the conceptual key value of the specific table. The process of deletion is automatized but these provided operations come in use when mistakes happen.
- *Custom Operations*: these are the operations required by default and listed on the specifications. Most of them are just implemented as a call of the CRUD operations (operations 1, 3, 4), others are implemented from scratch (operations 2, 5).

Sometimes operations on execution need to access across tables to retrieve the stored data and for this reason the process of ***Data Indexing*** comes in handy. Data Indexing is an operation that consists in adding indexes on fields of specific tables to further optimize the processes of information retrieval among tuples of the tables in terms of passed time. In the case of a database of few hundreads of tuples there is no problem, but in a million-tuples environment (such is this), retrieving a tuple may have a much higher computational cost in absence of indexes.

In any case, it's a fair habit not to randomly choose indexes because these occupy memory and require update for each operation of insert or deletion. For this reason it's important to choose appropriately only those indexes really useful to reduce high computational retrieval costs respect to the required operations.

| OPERATION | SELECTION AND JOIN OPERATIONS |
|---|---|
| Create Centre | Selection on MAX(CENTRE.id); |
| Create Event | Selection on MAX(EVENT.id); |
| Create Route | Selection on MAX(ROUTE.id);<br>Selection on MAX(EVENT.id); |
| Read Dealer | Selection on DEALER.*; |
| Read Shipment | Selection on SHIPMENT.*; |
| Read PastShipment | Selection on PASTSHIPMENT.*; |
| Read Centre | Selection on CENTRE.*; |
| Read Vehicle | Selection on VEHICLE.*; |
| Read Item | Selection on ITEM.*; |
| Read PastItem | Selection on PASTITEM,*; |

| OPERATION | SELECTION AND JOIN OPERATIONS |
|---|---|
| Read Event | Selection on EVENT.*; |
| Read Route | Selection on ROUTE.*; |
| Update Dealer | Selection on DEALER.id; |
| Update Shipment | Selection on SHIPMENT.shipmentCode; |
| Update Centre | Selection on CENTRE.id; |
| Update Vehicle | Selection on VEHICLE.plate; |
| Update Item | Selection on SUM(ITEM.calculateCosts()); |
| Update Event | Selection on EVENT.id; |
| Update Route | Selection on ROUTE.id; |
| Collect Item (Op1) | // see “Create Item” |
| Track An Item (Op2) | Selection on EVENT.id, EVENT.kind;<br>Selection on ROUTE.*;<br>Selection on ITEM.shippingCode;<br>Ordering on ROUTE.id, ROUTE.stamp; |
| Add Transportation Event (Op3) | // see “Create Event” |
| Add Position (Op4) | // see “Create Route” |
| Produce Weekly Report (Op5) | Selection on WEEKLYREPORT.id;<br>Selection on AVG(SHIPMENT.calculatePassedTime());<br>Selection on COUNT(SHIPMENT);<br>Selection on SUM(SHIPMENT.earnings);<br>Selection on SUM(VEHICLE.costs); |

The analysis will proceed only for the operations for which frequency is available, that are the custom ones.

*Operation 1*: has no optimizable operation.

*Operation 2*: has three optimizable selection and an optimizable ordering. Since this is an interactive operation, we will be considering potential indexes. It interests the following tables:

**ITEM** (*shippingCode STRING, content STRING, insurance NUMBER, dimensions DIMENSIONS, seller REF DEALER, buyer REF DEALER, shipment REF SHIPMENT*).

**ROUTE** (*id NUMBER, stamp DATE, position POSITION, event REF EVENT*).

The candidate attributes for optimization are:

- EVENT.id, indexed with B+tree due to the retrieval of EVENT.id and EVENT.kind;
- EVENT.kind, not indexed because retrieved through EVENT.id;
- ITEM.shippingCode, indexed with B+-tree due to the retrieval of ITEM.shipment;
- (ROUTE.id, ROUTE.stamp), indexed with B+-tree due to the retrieval of both in the alphanumerical order.

*Operation 3*: has a single optimizable selection. Since this is an interactive operation, we will be considering potential indexes. It interests the following tables:

**EVENT** (*id NUMBER, kind STRING, vehicle REF VEHICLE, shipment REF SHIPMENT, centre REF CENTRE*).

The candidate attributes for optimization are:

- MAX(EVENT.id), indexed with B+tree due to the retrieval of the maximum value for the attribute EVENT.id;

*Operation 4*: has two optimizable selections. Since this is an interactive operation, we will be considering potential indexes. It interests the following tables:

**EVENT** (*id NUMBER, kind STRING, vehicle REF VEHICLE, shipment REF SHIPMENT, centre REF CENTRE*).

**ROUTE** (*id NUMBER, stamp DATE, position POSITION, event REF EVENT*).

The candidate attributes for optimization are:

- MAX(ROUTE.id), indexed with B+tree due to the retrieval of the maximum value for the attribute ROUTE.id;
- MAX(EVENT.id), indexed with B+tree due to the retrieval of the maximum value for the attribute EVENT.id;

*Operation 5*: has five optimizable selections. Since this is a batch operation, we won't be considering potential indexes.

Oracle DBMS automatically creates B+tree indexes for attributes declared "UNIQUE" in the creation of the tables, hence all the listed attributes were already indexed before even considering the performance optimization step. Only one of them was added manually and it involved the attributes ROUTE.id and ROUTE.stamp.

Once the database was set up and the operations were fully optimized, it was useful to make the database active through the automatization of certain operations that normally the user should do in particular situations before the start or after the end certain state transactions.

This step is called ***Trigger Implementation*** since triggers are those tools that is needed for automatization. In particular the triggers implemented can be divided in two big subcategories:

- ***Control Triggers***: triggers that have to control the attributes inserted or updated satisfy their domain ranges.
- ***Deletion Triggers***: triggers that manage the recursive deletion/storing of the tables.

In the following section there will be an analysis of the triggers using the standard paradigm of Event-Condizion-Action (ECA).

*Control Triggers*: control triggers are identified by the keyword "*verify*" in their name and there are 4 of these kinds:

1) Trigger: *verifyDealerKind*

| | |
|---|---|
| EVENT | Insert or update a tuple in the DEALER table. |
| CONDITION | The attribute DEALER.kind is neither a 'seller' nor a 'buyer'. |
| ACTION | Raise the application error 20501 explaining the insertion/update problem to the user. |

2) Trigger: *verifyShipmentDates*

| | |
|---|---|
| EVENT | Insert or update a tuple in the SHIPMENT table. |
| CONDITION | The attribute SHIPMENT.withdrawalDate is higher than the attribute SHIPMENT.deliveryDate. |
| ACTION | Raise the application error 20502 explaining the insertion/update problem to the user |

3) Trigger: *verifyPastShipmentDates*

| | |
|---|---|
| EVENT | Insert or update a tuple in the PASTSHIPMENT table. |
| CONDITION | The attribute PASTSHIPMENT.withdrawalDate is higher than the attribute PASTSHIPMENT.deliveryDate. |
| ACTION | Raise the application error 20502 explaining the insertion/update problem to the user |

4) Trigger: *verifyLatLongRoute*

| | |
|---|---|
| EVENT | Insert or update a tuple in the ROUTE table. |
| CONDITION | The attribute ROUTE.latitude is not in range [-180, 180]. |
| ACTION | Raise the application error 20503 explaining the insertion/update problem to the user |

*Deletion Triggers*: deletion triggers are identified by the keyword "*deletion*" in their name and there are 6 of these kinds:

1) Trigger: *delShipmentStat*

| | |
|---|---|
| EVENT | Delete a tuple from the SHIPMENT table. |
| CONDITION | Any. Always applied. |
| ACTION | Delete the tuples in the tables EVENT and ITEM referenced to the shipment. |

2) Trigger: *delShipmentRow*

| | |
|---|---|
| EVENT | Delete a tuple in the PASTSHIPMENT table. |
| CONDITION | Any. Always applied. |
| ACTION | Inserts a new corresponding tuple in PASTSHIPMENT and updates the tuples in the table PASTITEM with the same shippingCode. |

3) Trigger: *delPastShipmentStat*

| | |
|---|---|
| EVENT | Delete a tuple in the PASTSHIPMENT table. |
| CONDITION | Any. Always applied. |
| ACTION | Delete the tuples in the table PASTITEM referenced to the past shipment. |

4) Trigger: *delCentreRow*

| | |
|---|---|
| EVENT | Delete a tuple in the CENTRE table. |
| CONDITION | Any. Always applied. |
| ACTION | Change the field in EVENT storing the centre to NULL. |

5) Trigger: *delVehicleRow*

| | |
|---|---|
| EVENT | Delete a tuple in the VEHICLE table. |
| CONDITION | Any. Always applied. |
| ACTION | Change the field in EVENT storing the centre to NULL. |

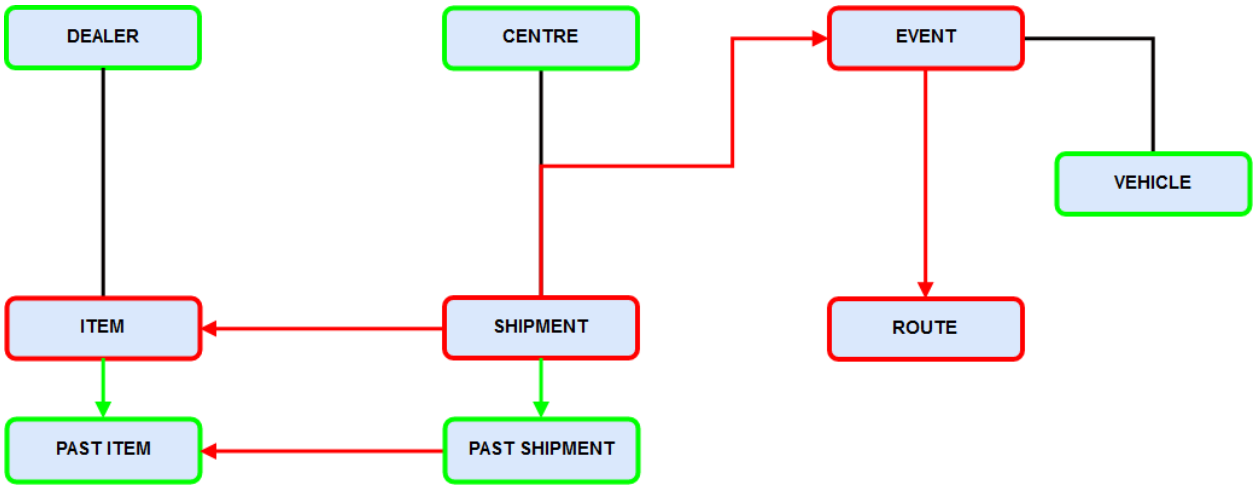
6) Trigger: *delItemRow*

| | |
|---|---|
| EVENT | Delete a tuple in the ITEM table. |
| CONDITION | Any. Always applied. |
| ACTION | Inserts the corresponding tuple in the table PASTITEM. |

7) Trigger: *delEventStat*

| | |
|---|---|
| EVENT | Delete a tuple in the EVENT table. |
| CONDITION | Any. Always applied. |
| ACTION | Delete the tuples in the table ROUTE referenced to the event. |

Deletion Triggers can be condensed in the ***Deletion Control Flow***:




The deletion control flow can be defined as a directed graph of consequent deletions that illustrates the iterative application of deletion triggers throughout a summary diagram of the entities. Entities in this diagram are distinguished according to the colour and specify into:

- *Green entities*: represent the table responsible of storing historical data. These cannot be removed in standard conditions. In case of user's mistakes, rollback operations need to be done by querying directly to the database by the administrator.
- *Red entities*: the entities that fire the deletion trigger cascade once these get removed.

Arrows in this diagram are distinguished according to the colour too and specify into:

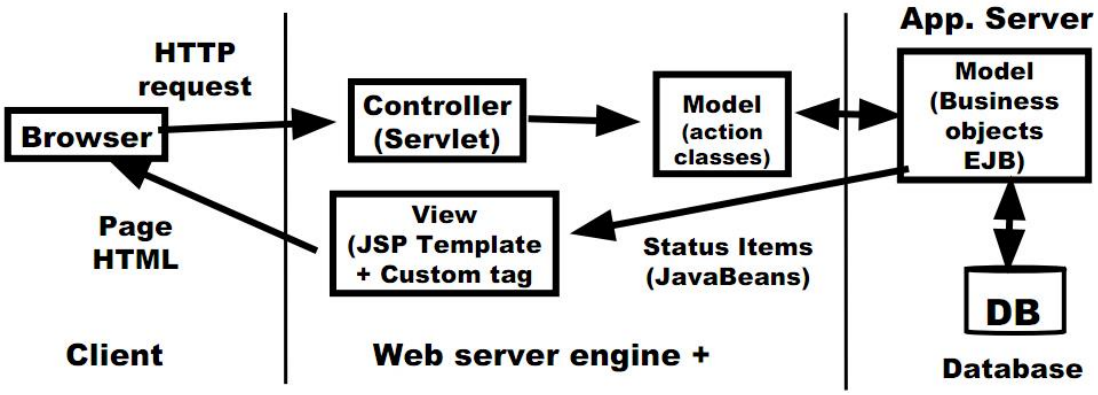
- *Green arrows*: represent the operation of creation for one or more tuples belonging to the destination table.
- *Red arrows*: represent the operation of deletion for one or more tuples belonging to the destination table.

# 4. Implementation of the Web Application

The next step in the project development plan is the implementation of a web application hosted on a specific web server to let the user execute the operations that requested and listed in the Specification Requirements.

The implemented web service is based on a ***MVC Architecture***, which is based on three different layers that communicate with each other and that encapsulate different kinds of logic:

- *Model*: represents the application's dynamic data structure from which data and business logic is directly managed. Hence, it includes the set of entities involved in the operations.
- *View*: represents the application's representation of information, that is translated in this specific application to servlets acquiring and sending messages with the jsp pages.
- *Controller*: converts inputs into commands defining the interactions between Model and View.




The implementation of the architecture also involved other ***J2EE Design Patterns*** useful to improve the structure and satisfying the Single Responsibility Principle elicited among the software development SOLID principles. The adopted patterns are:

- *Application Controller*: handles the operation obtained by the user's input through the servlet and creates the corresponding business object.
- *Business Object*: handles data and operations on data, split in two separate modules: entity handlers and operations handlers: the former implement the model while the latter call other object patterns to access the database and execute the correspondant queries. The result is a list of rows to be displayed by the servlet into the browser interface.
- *Transfer Object*: handles data that need to be transported from the application controller to the model and viceversa.
- *Data Access Object*: handles the submission of queries to the database according to the selected operation and returns the model updated with the information derived from the selection of result tuples.

The servlets contain the methods *get* and *post* that are recalled by the jsp pages. For each entity there is a jsp page and each operation is accessible througha navbar with dropdown menus.

The business objects have been modeled through the use of ***Enterprise JavaBeans*** (EJB), that is a Java API useful to encapsulate the business logic of an application. Since it is a Java object executed within EJB containers installed on the application server, it was necessary to use a server different from Apache Tomcat as soon as it does not include this particular enviroment. Instead of Tomcat, ***WildFly*** (the newer version of *JBoss*) was utilized to run the environment.

Two kinds of classes were implemented:

- Entity Bean classes, that encapsulate data.
- Java Bean classes, that encapsulate operations on data.

For Java Beans it was considered two interfaces for the business interface:

- Local Interface, to access to EJBs on the same host application.
- Remote Interface, to access to EJBs on a remote server.

These interfaces also let the application to live in a distributed context in which the application server on which is hosted the servlet does not know how the ejbs are made. This is possible thanks to a *Java Naming and Directory Interface* (*JNDI*) that is asked to instantiate the corresponding business object through the annotation "*@EJB*".

In order to further improve the abstraction, hence the separation of roles, the architecture was enriched with ***Java Persistence API*** (*JPA*). It is a framework that handles data persistence in the database through the use of annotations and is in tight correlation with ORM. *Object-Relational Mapping* consists in the integration of RDBMS systems with software systems that adopt the object-oriented paradigm, in the specific case Oracle Relational DBMS and a Web Application developed in Java.

The exploited JPA Framework is ***Hibernate*** and it allows to access to the database through an EntityManager which automatizes the process of connection to the database using the *persistence.xml* file. This file is composed of persistence unit tags that include the entity to be mapped and the correlated informations to access to the database:

- Driver: the Oracle JDBC driver.
- URL: the address complete of jdbc prefix, ip, port and the service name.
- User: the user enabled to access the database.
- Password: the correspondant password.
- Hibernate Dialect: the dialect used by hibernate to recognize the annotations.