Building the Product Supported Database

Camron Khan

Introduction

In order to assist contact center representatives efficiently redirect calls, Motorola Solutions' North America Solution Support Center utilizes an unnormalized list of products it supports – including information regarding technical support contacts, repair service contacts, hours of operation, and additional comments, . This *Product Supported List (PSL)* is currently implemented in an Excel file with a "front end" tab where users can input search terms, which are utilized in a VLOOKUP function to search values located on a "back end" tab. The result of the VLOOKUP is returned to the user on the initial tab. It is important to note, this method only allows one value to be returned at a time.

Please follow the links below to view the Product Supported List prior to its overhaul and the Product Supported Database currently in construction:

Before

Product Supported List

After

Product Supported Database

Initial Schema

In its initial form, the PSL is one relation comprised of eleven attributes. The Excel row number (<Row#>) was included below as one of the attributes because it is implicitly utilized to individuate tuples.

productSupportedList (LookupValue, ProductName, SupportGroup, ContactInfo, Hours, Escalation, QuickNotes, Repair, ProductCategory, Comments)

Below is a description of each attribute:

- **ProductName:** Official product name
- **LookupValue:** Set of keys utilized by the VLOOKUP function
- **SupportGroup:** Name of the technologist group that supports the product
- **ContactInfo:** Technologist group's contact information
- **Hours:** Technologist group's hours of operation
- **Escalation:** Notes regarding technical support escalations (i.e., after hours)
- **QuickNotes:** Highlighted comments
- **Repair:** Facility name and contact information where the product is repaired
- **ProductCategory:** Product division (this attribute is defunct post-divestiture of the enterprise mobility business in 2014)
- **Comments:** Additional information

Normalization, Decomposition, & Standardization

Below are the steps taken to normalize and decompose the relations in the PSL:

1. ProductCategory was eliminated as an obsolete attribute due to the aforementioned change in organizational structure:

   productSupportedList (ProductName, LookupValue, SupportGroup, ContactInfo, Hours, Escalation, QuickNotes, Repair, Comments)

2. QuickNotes and Comments were identified as redundant attributes, and as a result, QuickNotes was eliminated:

   productSupportedList (ProductName, LookupValue, SupportGroup, ContactInfo, Hours, Escalation, Repair, Comments)

3. Servicer address information was required as a necessary attribute, which was not included in the original implementation of the PSL. As a result, Address was added:

   productSupportedList (ProductName, LookupValue, SupportGroup, SupportInternalContact, SupportExternalContact, Hours, Escalation, Repair, Address, Comments)

4. ContactInfo was identified as violating atomicity (1NF) as tuples may contain internal contact information, external contact information, or both. As a result, it was eliminated and replaced with the SupportInternalContact and SupportExternalContact attributes:

   productSupportedList (ProductName, LookupValue, SupportGroup, SupportInternalContact, SupportExternalContact, Hours, Escalation, Repair, Address, Comments)

5. Repair was also identified as violating atomicity (1NF) as tuples containing repair information includes both the service and their contact number. As a result, Repair was eliminated and replaced with the ServiceName, ServiceInternalContact, ServiceExternalContact attributes:

   productSupportedList (ProductName, LookupValue, SupportGroup, SupportInternalContact, SupportExternalContact, Hours, Escalation, ServiceName, ServiceInternalContact, ServiceExternalContact, Address, Comments)

6. A number of repetition, insert, update, and delete anomalies were identified due to the fact that the product functionally determined the product's search key and comments, the technologist group functionally determined the internal and external contact information, hours of operation, and escalations, and the servicer functionally determined the servicer contact information:

   ProductName → { LookupValue, Comments }
   SupportGroup → { SupportInternalContact, SupportExternalContact, Hours, Escalation }
   ServicerName → { ServiceInternalContact, ServiceExternalContact, Address }

   Due to this violation of 2NF, the *productSupportedList* relation was decomposed into the *product*, *technologist*, and *servicer* relations, and ProductName, SupportGroup, and ServiceName were identified as the primary keys, respectively:

   product(<u>ProductName</u>, LookupValue, Comments)
   technologist(<u>SupportGroup</u>, SupportInternalContact, SupportExternalContact, Hours, Escalation)
   servicer(<u>ServiceName</u>, ServiceInternalContact, ServiceExternalContact, Address)

7. It was determined that company information must be stored for each product since Motorola Solutions fields interactions regarding third party equipment. As a result, the CompanyName and CompanyNotes attributes were added to the *product* relation:

product(ProductName, LookupValue, Comments, CompanyName, CompanyNotes)

8. A transitive dependency, and thus a violation of 3NF, was identified among the ProductName, CompanyName, and CompanyNotes attributes in the *product* relation:

   ProductName → CompanyName → CompanyNotes

   As a result, the *product* relation was decomposed into the *product* and *company* relations:

   product(ProductName, LookupValue, Company, Comments)
   company(CompanyName, CompanyNotes)
9. In order to preempt the rare cases in which a product, technologist, servicer, or company name may change, an ID attribute was added to each relation and designated as its primary key:

   product(ID, ProductName, LookupValue, Comments)
   company(ID, CompanyName, CompanyNotes)
   technologist(ID, SupportGroup, SupportInternalContact, SupportExternalContact, Hours, Escalation)
   servicer(ID, ServiceName, ServiceInternalContact, ServiceExternalContact, Address)

10. To link the *product* and *company* relations, a CompanyID attribute was added to *product* that references *company*(ID):

    product(ID, ProductName, LookupValue, CompanyID, Comments)

11. In order to record relation-specific notes and avoid redundant storage of information, it was determined a Notes attribute should be added to all relations. To standardize attribute labels, *product*(Comments) and *company*(CompanyNotes) were renamed:

    product(ID, ProductName, LookupValue, CompanyID, Notes)
    company(ID, CompanyName, Notes)
    technologist(ID, SupportGroup, SupportInternalContact, SupportExternalContact, Hours, Escalation, Notes)
    servicer(ID, ServiceName, ServiceInternalContact, ServiceExternalContact, Address, Notes)

12. In order to track when updates were made to relations, a LastActivity attribute was added to each relation:

    product(ID, ProductName, LookupValue, CompanyID, Notes, LastActivity)
    company(ID, CompanyName, Notes, LastActivity)
    technologist(ID, SupportGroup, SupportInternalContact, SupportExternalContact, Hours, Escalation, Notes, LastActivity)
    servicer(ID, ServiceName, ServiceInternalContact, ServiceExternalContact, Address, Notes, LastActivity)

13. Each relation's attributes were reviewed and edited (where applicable) to enforce standardized naming conventions:

    product(id, name, search_term, company_id, notes, last_activity)
    company(id, name, notes, last_activity)
    technologist(id, name, internal_contact, external_contact, hours, escalation, notes, last_activity)
    servicer(id, name, internal_contact, external_contact, address, notes, last_activity)

14. Technologists support many products and certain products are supported by multiple technologists (e.g. if the product is used in conjunction with different systems). As a result, a *support_assignment* relation was created to model this many-to-many relationship:

    support_assignment(id, product_id, technologist_id, notes, last_activity)

15. Similarly, servicers repair many products and certain products may be repaired at multiple service facilities. As a result, a *repair_assignment* relation was created to model this many-to-many relationship:

    repair_assignment(id, product_id, servicer_id, notes, last_activity)

16. Finally, the database was renamed "Product Supported Database" (or, *psdb*) to indicate the change in structure.

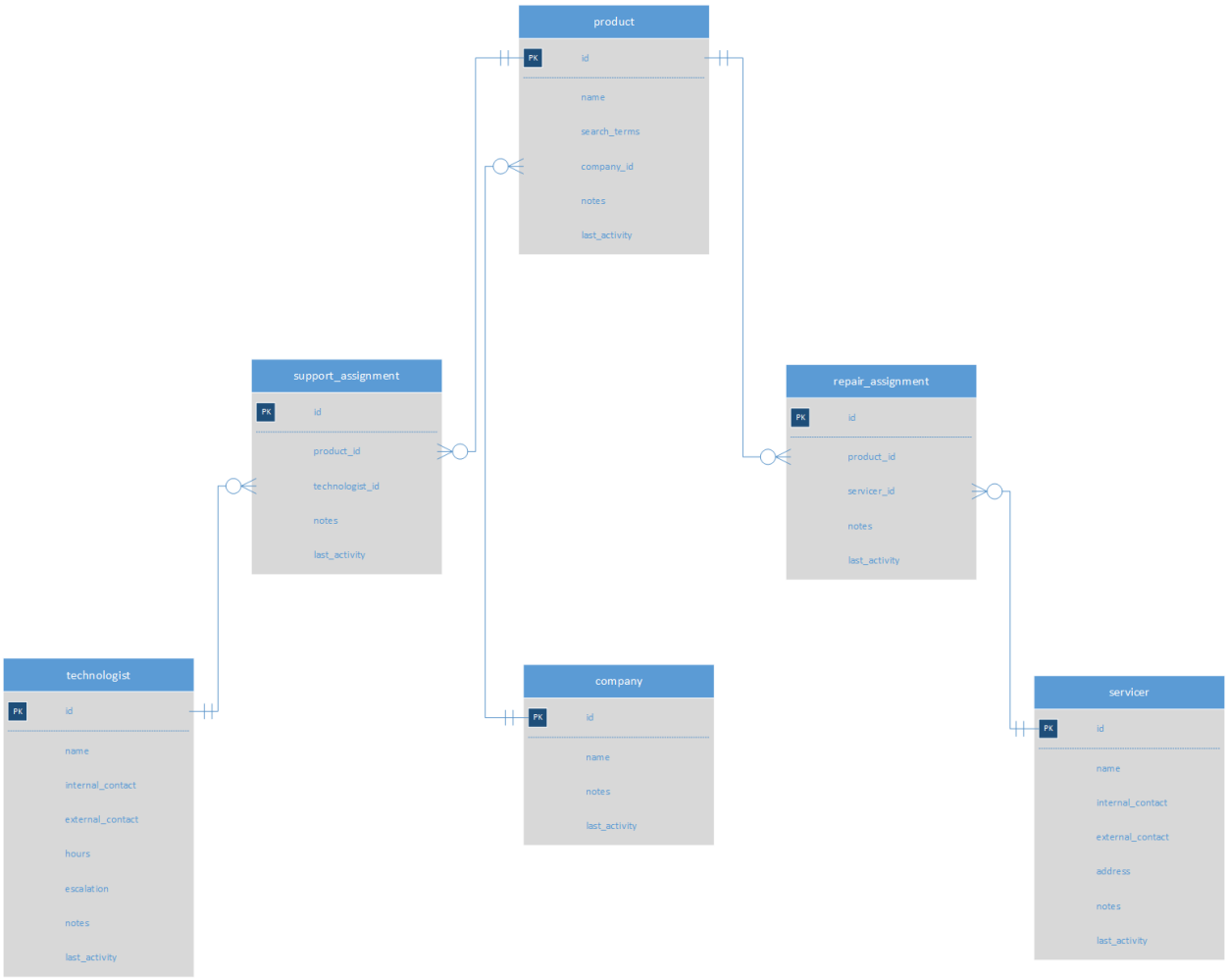
Revised Schema

Below is a revised schema of the relations based on the normalization, decomposition, and standardization performed above:

product(id, name, search_term, company_id, notes, last_activity)
company(id, name, notes, last_activity)
technologist(id, name, internal_contact, external_contact, hours, escalation, notes, last_activity)
servicer(id, name, Internal_contact, external_contact, address, notes, last_activity)
support_assignment(id, product_id, technologist_id, circumstance, notes, last_activity)
repair_assignment(id, product_id, servicer_id, circumstance, notes, last_activity)

Entity-Relationship (ER) Diagram

Relational Schema Diagram



The following relationships are modeled in the above relational schema diagram:

- Each product comes from exactly one company
- Each company can own zero or more products
- Each product can have zero or more support assignments
- Each support assignment is created for exactly one product
- Each support assignment is assigned to exactly one technologist
- Each technologist can be assigned zero or more support assignments
- Each product can have zero or more service assignments
- Each service assignment is created for exactly one product
- Each service assignment is assigned to exactly one servicer
- Each servicer can be assigned zero or more service assignments

Algebraic Statements

The application's search function allows users to search via product name or company name. Below are two algebraic expressions demonstrating each:

Search via product name

The following algebraic expression returns the product name and company name where the product's name matches the user's inputted search terms. More specifically, the expression results in a relation that projects the product name and company name based on a selection where the product's company ID must match the associated company's ID and an additional selection where the product's name must be "APX 7000" (in this particular instance):

$$\pi_{\text{product.name, company.name}}(\sigma_{\text{product.company = company.id}}(\sigma_{\text{product.search\_terms = "APX 7000"}}(\text{product x company})))$$

Search via company name

The following algebraic expression returns all products owned by Motorola Solutions. More specifically, the expression results in a relation that projects the product name and company name based on a selection where the product's company ID must match the associated company's ID and an additional selection where the company's name must be "Motorola Solutions":

$$\pi_{\text{product.name, company.name}}(\sigma_{\text{product.company = company.id}}(\sigma_{\text{company.name = "Motorola Solutions"}}(\text{product x company})))$$

SQL Statements

The following MySQL statement performs a comparison of the user's inputted search term(s) – denoted by the PHP variable, $userInput – against *product*(search_term) and returns a list of potential product matches along with a "quick view" of details. These mirror the functions described in the above section, "Algebraic Statements":

Search via product name

```
SELECT p.id AS p_id, p.name AS p_name, p.last_activity AS p_activity, c.name AS c_name
FROM product AS p, company AS c
WHERE p.company_id = c.id
AND (p.search_terms LIKE '%$userInput%')
ORDER BY p.name;
```

Search via company name

```
SELECT p.id AS p_id, p.name AS p_name, p.last_activity AS p_activity, c.name AS c_name
FROM product AS p, company AS c
WHERE p.company_id = c.id
AND (c.name LIKE '%$userInput%')
ORDER BY p.name;
```

The next MySQL statement returns all products if the user provides no search terms:

```
SELECT p.id AS p_id, p.name AS p_name, p.last_activity AS p_activity, c.name AS c_name
FROM product AS p, company AS c
WHERE p.company_id = c.id
ORDER BY p.name;
```

This MySQL statement queries the product, company, technologist, and servicer details given a product ID, which is retrieved by a user clicking on the desired product link:

```
SELECT p.name AS p_name, p.notes AS p_notes, c.name AS c_name, c.notes AS c_notes, t.name AS t_name,
        t.internal_contact AS t_iContact, t.external_contact AS t_eContact, t.hours AS t_hours, t.notes AS t_notes,
        supp.circumstance AS supp_circumstance, supp.notes AS supp_notes, s.name AS s_name,
        s.internal_contact AS s_iContact, s.external_contact AS s_eContact, s.address AS s_address, s.notes AS
        s_notes, serv.circumstance AS serv_circumstance, serv.notes AS serv_notes
FROM product AS p
JOIN company AS c ON p.company_id = c.id
JOIN support_assignment AS supp ON p.id = supp.product_id
JOIN technologist AS t ON supp.technologist_id = t.id
JOIN service_assignment AS serv ON p.id = serv.product_id
JOIN servicer AS s ON serv.servicer_id = s.id
WHERE p.id = $productID;
```

Search Terms

User input will be compared against a list of set of search terms – i.e., *product*(search_term) – to allow for variations of the official product name. Each element in *product*(search_term) is a "stripped down" version of the official product name; it is the product of removing all whitespace and non-alphanumeric characters and changing all remaining characters to lowercase. For example, if a user searches for "2031(B02) Base Station Controller," this string will be passed to a Javascript function that will convert it to its search term equivalent utilizing regular expressions. As a result, the search term "2031b02basestationcontroller" will be passed to PHP and ultimately MySQL for comparison against the elements in *product*(search_term). Below is the Javascript function:

```
function produceSearchTerm (input) {
        var temp = input.replace(/[^A-z0-9]/gi, '');        // Remove special characters
        temp = temp.replace(/[ ]/gi, '');                   //  Remove whitespace
         temp = temp.toLowerCase();                         //  Convert to lowercase
         return temp;                                       //  Return string
}
```

Creating Relations

Below are the MySQL commands to create the database's relations:

```
CREATE TABLE product (
        id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL UNIQUE,
        search_term VARCHAR(1023) NOT NULL,
        company_id INT UNSIGNED,
        notes TEXT,
        last_activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
        FOREIGN KEY(company_id) REFERENCES company(id)
) ENGINE=Aria;
```

```
CREATE TABLE company (
        id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL UNIQUE,
        notes TEXT,
        last_activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) ENGINE=Aria;

CREATE TABLE technologist (
        id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL UNIQUE,
        internal_contact VARCHAR(255),
        external_contact VARCHAR(255),
        hours VARCHAR(255),
        notes TEXT,
        last_activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) ENGINE=Aria;

CREATE TABLE servicer (
        id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL UNIQUE,
        internal_contact VARCHAR(255),
        external_contact VARCHAR(255),
        address VARCHAR(255),
        notes TEXT,
        last_activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) ENGINE=Aria;

CREATE TABLE support_assignment (
        id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        product_id INT UNSIGNED NOT NULL,
        technologist_id INT UNSIGNED NOT NULL,
        circumstance TEXT,
        notes TEXT,
        last_activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
        FOREIGN KEY(product_id) REFERENCES product(id),
        FOREIGN KEY(technologist_id) REFERENCES technologist(id)
) ENGINE=Aria;

CREATE TABLE service_assignment (
        id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        product_id INT UNSIGNED NOT NULL,
        servicer_id INT UNSIGNED NOT NULL,
        circumstance TEXT,
        notes TEXT,
        last_activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
        FOREIGN KEY(product_id) REFERENCES product(id),
        FOREIGN KEY(servicer_id) REFERENCES servicer(id)
) ENGINE=Aria;
```

User Interface

I utilized the Bootstrap framework to implement the user interface for three reasons. First, using a pre-defined framework allowed me to quickly produce a prototype and iteratively code and test during the development phase. Second, I required a proven framework that works across multiple devices, screen widths, and browsers since we have many a variety of agents access the tool in different ways. Third, Bootstrap's default color scheme resembles the Motorola Solutions brand color scheme, and as a result, I was able to quickly produce a tool that agents would feel comfortable using.