**CSC3170 Group Project Report**

**Product Sales and Repair Database for a UAV Company**

Yihan Li 118010154

Zeyu Li 118010158

Yuxuan Liu 118010200

Yutao Qiu 118010249



**Abstract**

As science and technology advancing rapidly, unmanned aerial vehicle (UAV) has been popularized all over the world. It has been widely utilized in various fields such as military or meteorology and nowadays, it has even come into the market for personal usage like photography. The DJI company is one of the leading companies in this industry in China. Considering its future prospect, we believe that the database of the DJI for UAV is worth designing. We hope to learn about how to design a desirable database efficiently through this project.

Our group have figured out that the need for database mainly focus on the relationship between products, including drones and accessories, and relevant parts: sales and repair as well as staff and store. Based on these we first made several assumptions, and then we created the ER diagram as well as the relational schema for the database we designed according to the normalization analysis. Then we inserted in the data and uncovered some useful facts via query test. We even created a GUI to help us retrieve the data conveniently. Finally, we completely further exploration on indexing, which is related to the performance of the query, and data mining.

**Database Design**

**Assumptions**

Here are several assumptions concerning the organization’s operation:

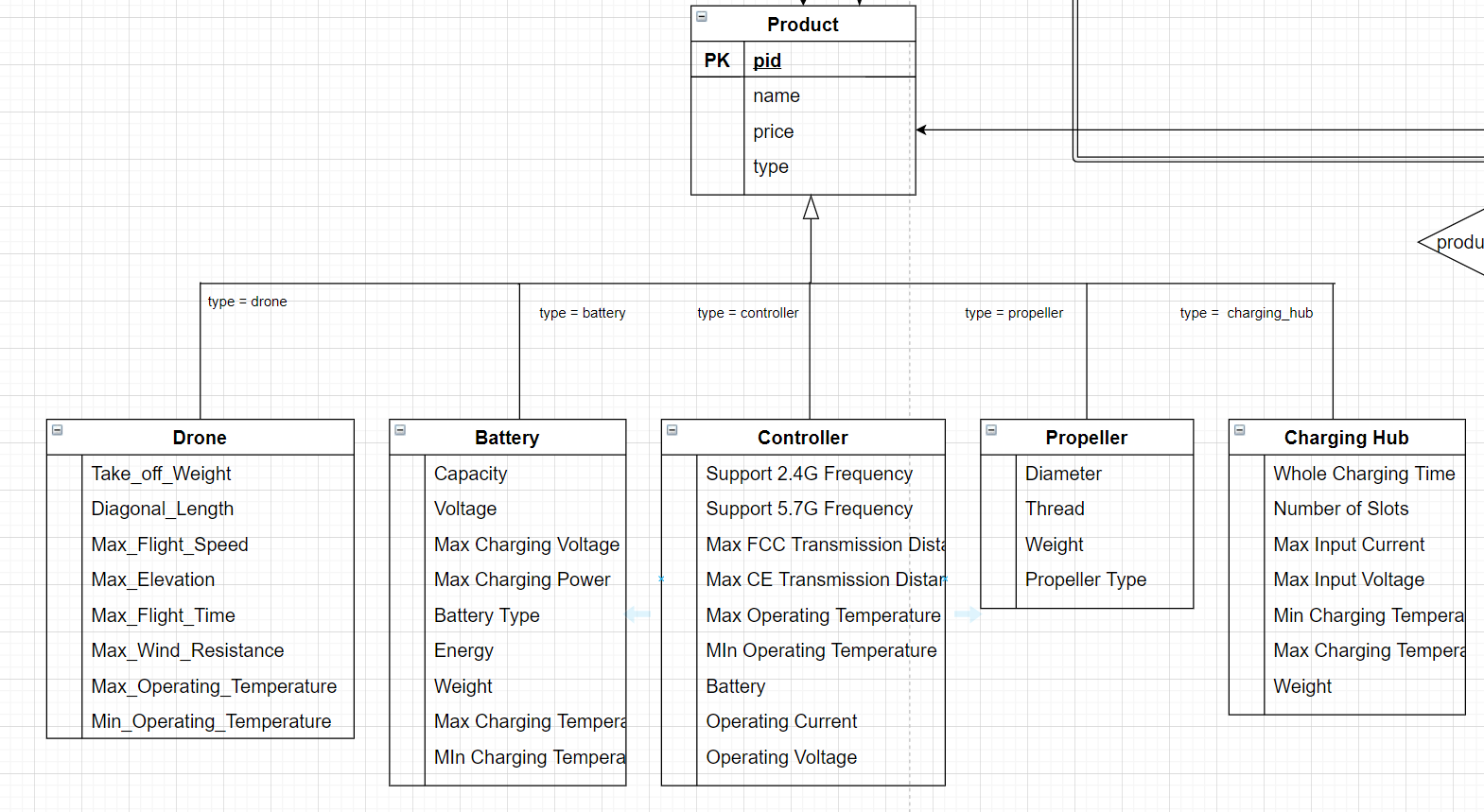
1. Products only include drones and other four kinds of components: batteries, controllers, propellers and charging hubs.
2. Each customer has only one address and one telephone to receive the goods.
3. Each time a customer can buy multiple drones and components in a sale order through only one sales staff.
4. Each time a customer can send a drone to repair in a store in a repair order through only one repair staff.
5. In each repair order, only one component will be replaced or repaired.
6. The repair sections are only located within the stores while some stores may not provide repairing service.
7. One repair order can only guarantee the repairing of one product.
8. There are only three positions for employees: sales staff, repair staff and factory staff.
9. One employee can only provide service in a fixed post.
10. Each factory has only one address and telephone number.

**E-R Diagram Design**

We primarily divided the entities into three groups: products, sales & repair, and staff & stores.

**(1) Products**

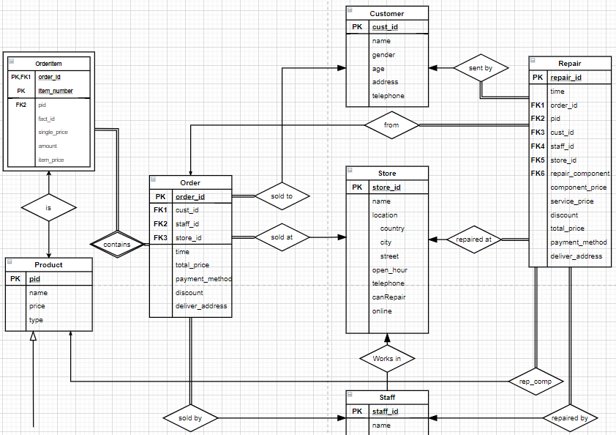
Each product has a unique product id (*pid*), with its *name*, *price* and *type* as generalized attributes. Five specializations - *Drone*, *Battery*, *Charging Hub*, *Controller*, and *Propeller* - inherit from product, where attribute *product.type* is the determinant. The specialization is disjoint and complete. The structure of the inheritance is shown below:



*Figure 1: Products: Product and its specializations*

**(2) Sales & repair**

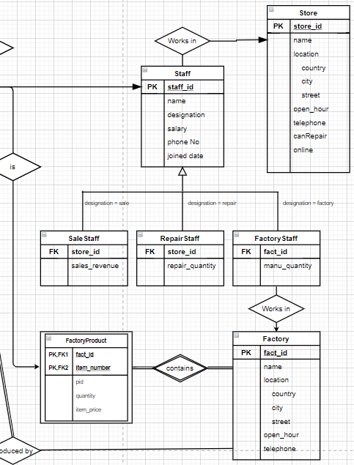
We consider the entity *order* as the center of the relationships. An order is composed by the information including customers, sales staffs, products and so on. Entity *order* contains foreign key: *cust\_id*, *staff\_id*, and *store\_id*, and these foreign keys connect *order* with entity *customer*, *store*, and *staff*. Also, the order records are required for repair orders, since the company need to confirm the truth that the products are sold by them. Additionally, the information of customers, stores, and staffs are added considering the whole delivery process. Besides, to specify the particular product item in an order, we designed an entity *OrderItem*, whose records are determined by a composite key referencing from *order* and *product*. Similar to *repair* entity, it contains the information about repair order. It needs to record who sent it (customer), who repaired it (staff), where it was repaired (store), the repaired product information (such as order, product, and repair component), and other related information like price.



*Figure 2: Entities and relationships of sales & repair*

**(3) Staff & stores**

In the staff part, we suppose that each staff only works for one store or factory. We divided the staffs into three types: *sale staff*, *repair staff*, and *factory staff*. Sale staff and repair staff work in stores. Sale staff take charge of selling products, and repair staff repair the products. Factory staff manufacture products in factories. We also record the products that each factory produces in order to derive the source of bad products.



*Figure 3: Entities of staff & stores*

As to the complete E-R diagram, please go to appendix A.

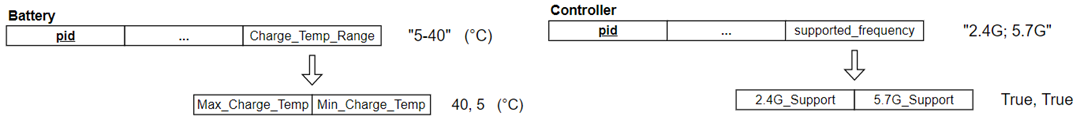
**Database Implementation**

In this section we will introduce how to convert our E-R diagram into relational schemas. The process includes normalization and specialization design.

**(1) Normalization**

**(a) 1NF: Atomic Domine**

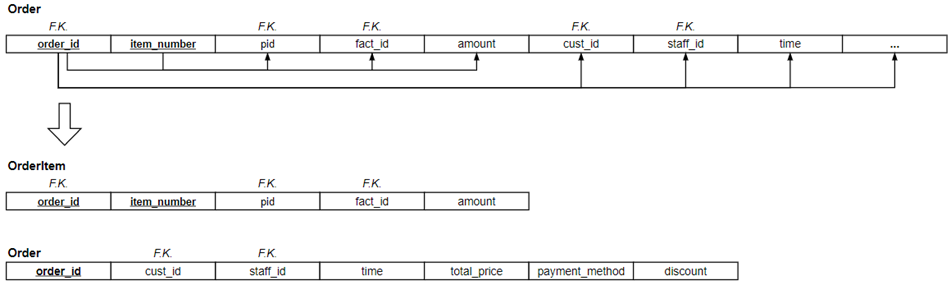
In some attributes, the recorded data is composite. We decompose them into several atomic attributes to satisfy the 1NF condition. For example, we decompose the *Charge Temperature Range* into *Max Temperature* and *Min Temperature*; and decompose the *Supported Frequency* (may contain 2.4G or 5.7G or both) into Boolean attributes *Support 2.4G* and *Support 5.7G*.



*Figure 4: 1NF process*

**(b) 2NF: Fully Functional Dependency**

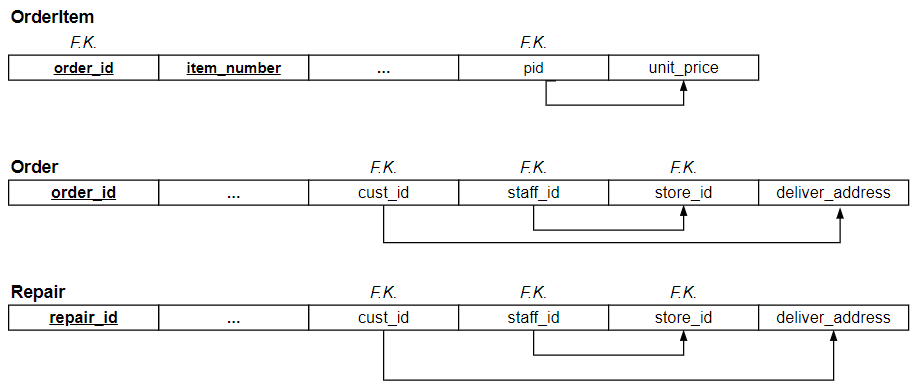
Take the Order entity as an example. The initial design didn’t satisfy the fully functional dependency because the attributes cust\_id, staff\_id, time and some others in fact are dependent on order\_id only. Therefore, we decompose it into Order and OrderItem entities, which satisfies the fully functional dependency and be in 2NF.



*Figure 5: 2NF process*

**(c) 3NF: No Transitive Dependency on PK**

Here are some examples of initial entity design, our goal is to eliminate the transitive dependency on the prime key. In the *OrderItem* schema, we can simply drop *unit\_price* to eliminate transitive dependency,because we can get this data from the *Product* schema. Similar process to the *store\_id* and *deliver\_address* in *Order* and *Repair* entities, which can be accessed through *Customer* and *Staff* entities. After this process, all the entities are in 3NF.

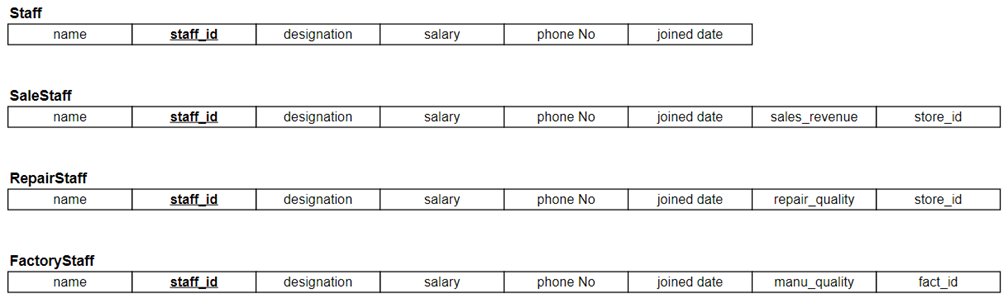


*Figure 6: 3NF process*

Until now, all the schemas are in good normalization form.

**(2) Specialization Design**

In our initial ERD relation design, both *Product* and *Staff* have specialization structure. In the relational schema design, for example, in the specialization staff schemas, they inherited all the attributes from the Staff schema, and each of them has their local attributes. We use this design because all the specializations are disjoint so that there is no redundancy, and meanwhile it simplifies the query process. Same treatment for *product* specialization schema.



*Figure 7: Specialization design on Staff schema*

As to the complete relational schema, please go to appendix B.

**Test and Analysis**

**(1) Data Insertion**

There are only limited pubic data available in our database, therefore we have to combine part of the real data and some randomly generated data.

**(a) Real data for table "product"**

We found plenty of specifications of drones and other products on DJI website and retrieved that part of data as the foundation of our database. We totally inserted 20 records of drone specifications and several rows for other types of products respectively.

**(b) Random data for other tables**

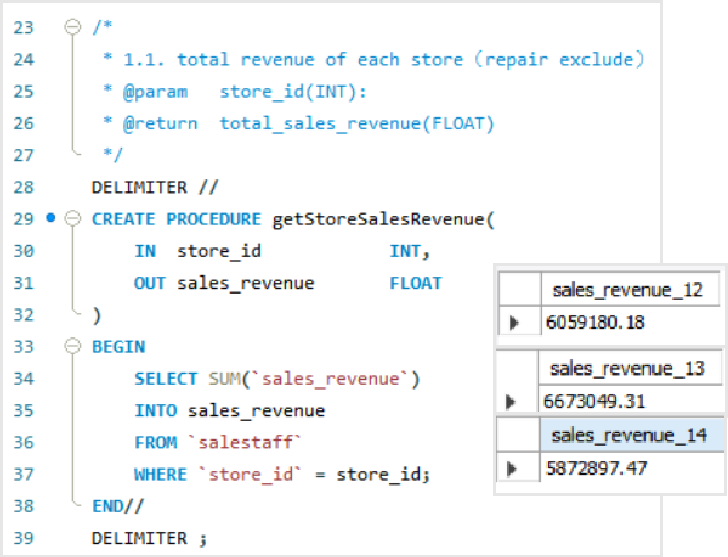
Based on the real data of products, we extended random data for other tables. We first determined the domains of each attribute, and then generated data and insert into database by python scripts. To accelerate the insertion process, we turn off the "autocommit" mode in MySQL. Related python codes are attached in the appendix.

**(2) Query Test**

We designed several query examples to test and validated our database. These query tasks are very likely to appear in practice.

**(a) Query sales revenues of each store**

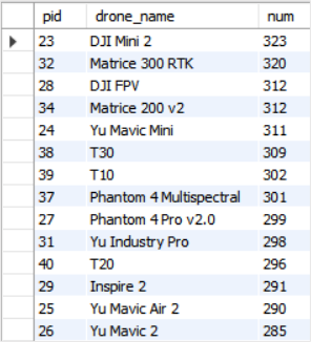
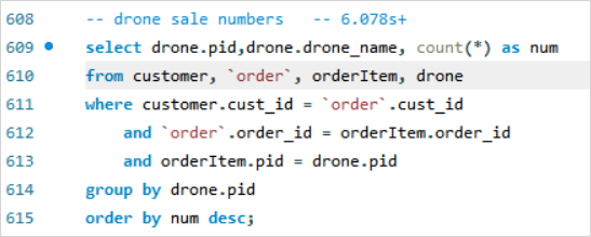
Sales revenue is a vital measurement of development of a company. Thus, managers of the company may want to inspect the sales of all the stores. We put this query task into the form of procedure, which helps managers to focus on a particular store directly. The snapshot of codes and query results is shown below:



*Figure 8: Query sales revenues of each store*

**(b) Query drones with the largest sales**

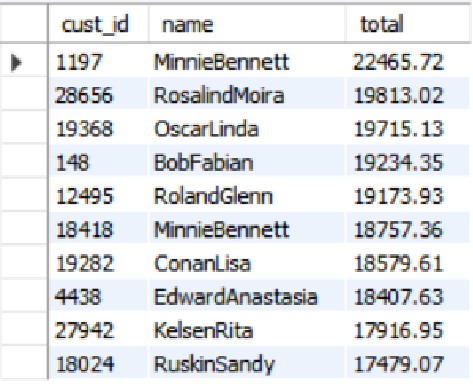
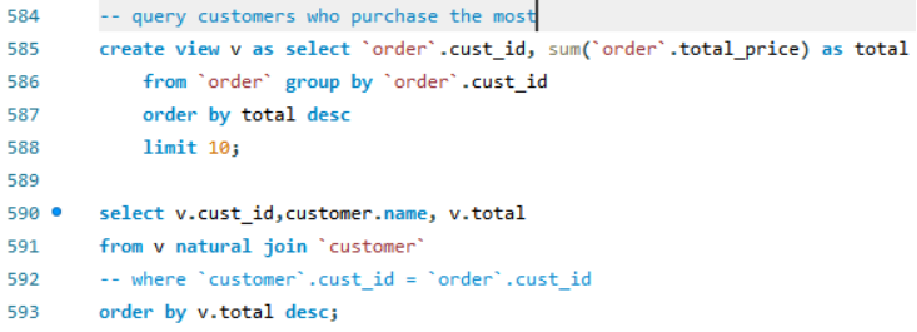
To understand market and customers’ preferences is a determining method of improving sales revenue. The company needs to assess the sales of each of their products, as this query task work. The following snapshots contain the codes and ten most popular drones:



*Figure 9: Query drones with the largest sales*

**(c) Query customers who purchase the most**

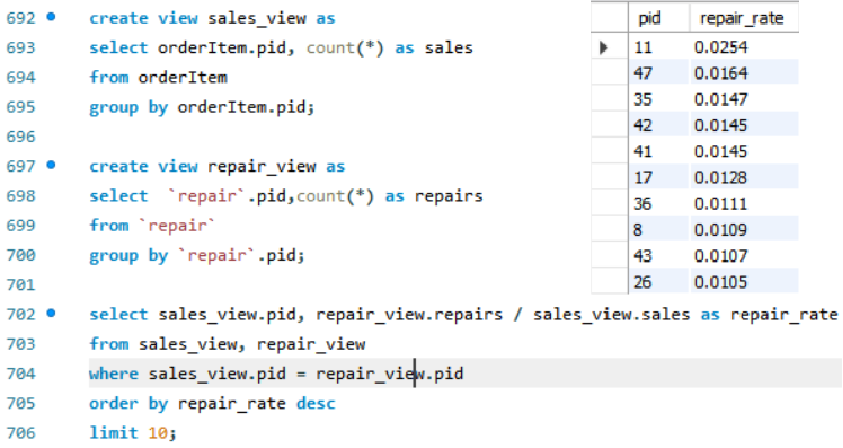
Sometimes a group of loyal and generous customers can contribute the majority of sales. By tracking them, the company may earn more profits rather than balance the requirement of all the customers. Related codes and corresponding query results are listed below:



*Figure 10: Query customers who purchase the most*

**(d) Query the products with the highest repair rate**

Other than pursuing more profit, the company must guarantee the quality of their products as well. Hence, managers want to check the repair rate of their products. Those products with high repair rates may need to be redesigned. The snapshot of codes and query results is shown below:

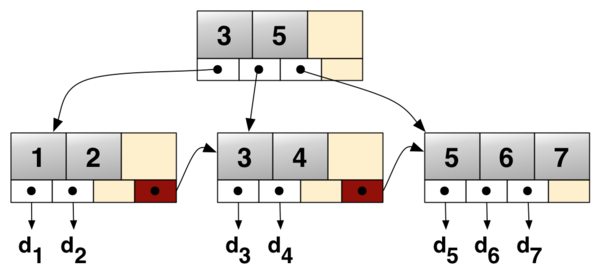


*Figure 11: Query the products with the highest repair rate*

**(3) Indexing**

Since our group is designing a database for a real company which has a large scale with tons of data involved, indexing is supposed to be introduced to speed up the queries and make the database more efficient. As previously taught by the professor, there are basically two types of indices that are commonly used: tree structure index and hash index. Due to the different data structure used, these two indices are different in multiple aspects, such as efficiency, stability and etc. However, before we move on to the details of each type of index, it has to be made clear that the InnoDB engine we chose in MySQL does not support user defined hash index and therefore all the discussions below are only for theoretical purposes.

**(a) Tree Structure Index (B+ Tree in InnoDB)**



*Figure 12: B+ Tree Diagram*

Unlike B tree which stores real records in the interval nodes, the records in B+ tree only exists among the leaf nodes, and this expands the number of nodes in a layer since the pointer occupies far less space than a real record. In this way, the number of layers is minimized to the largest extent and so is the time complexity.

**Time Complexity of B+ Tree Structure: O(log N)** [N: number of records in the tree]

Suppose the order of the tree is m, which means there are m child pointers in an internal node. Assume that the total number of layers is l, then

ml = N

l= logmN

However, one of the drawbacks of B+ tree structure is that it cannot decide which node to go next in O(1) time, instead it needs to iterate through the nodes sequentially to decide which interval the target node belongs to, which results in an average of (m/2) time.

Therefore, the final time complexity is

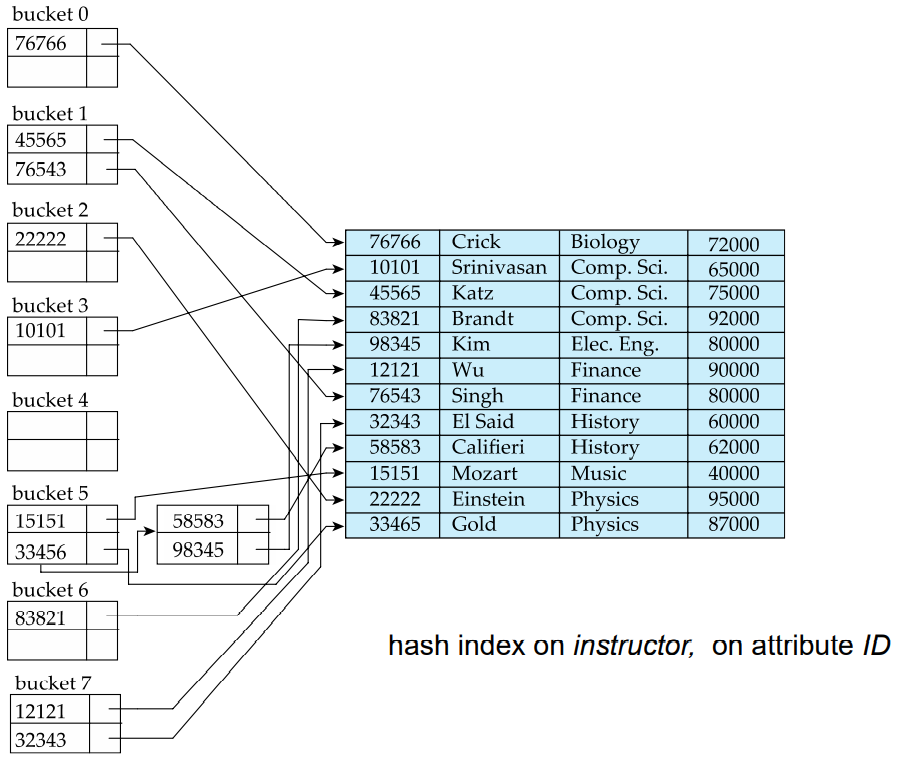
logmN \* (m/2) = log N [as N approaches +∞]

**Advantages and Disadvantages of Tree Index**

The best thing about the tree structure index is that it does not have obvious disadvantages, which makes it a good choice for nearly all the situations. Unless the extreme case of linear tree structure occurs, it is very stable for searching as well as the other operations. However, the efficiency is not as good as hash index, so we always consider it as a plan B and use it only when hash index is unavailable.

**(b) Hash Index**

Hash index, with the help of hashing functions, is a way to establish a one-way map between search keys (input) to the hash value (output), which makes it extremely fast in exact lookup. The hash table data structure looks like the following graph:



*Figure 13: Hash Index in Database*

**Time Complexity of Hash Index: between O(1) and O(log N)**  [N: number of records]

There may be situations when the same hash value (or called bucket here in MYSQL) is generated for different search keys just like the graph shows. As a result, even we use the hash function to get the bucket address using O(1) time complexity, extra time for searching among the bucket is still needed. However, that process can be then optimized with the help of other data structure, like B tree, so the overall time complexity of hash index is still less than that of tree index. In fact, hash index is way more efficient than tree index, which is the reason why we always prefer it.

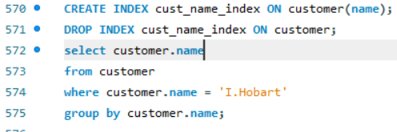
**Advantages and Disadvantages of Hash Index**

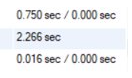
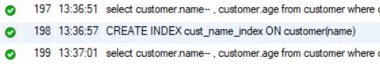
Although the speed is fast, hash index is very unstable and can be a bad choice for certain situations, such as range searching and composite search keys. When dealing with attributes involving these two operations, we have to abandon hash index and use tree index instead.

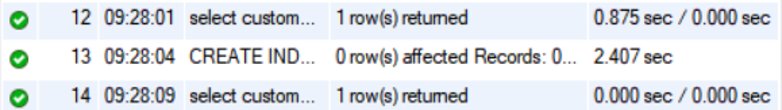
Range searching is not supported by hash index because the hash function is most likely to be not order-preserving, which means the hash value generated will not be in the same order as they used to be as search keys. Take the hash table in the graph above for instance, the hash algorithm is the (sum of each bit in id % 8). It maps 76766 to bucket 0 and 10101 to bucket 3, which reverses the order. In fact, as the MYSQL official document says, hash index can only be used in equal search (= or <=>) but cannot speed up any ORDER BY or range operation.

Composite Search Key is also not supported by hash index since a new hash algorithm is needed if we want to composite more than one attributes as search key. The reason is that the hash function must be carefully designed to suit the data type and avoid as many collisions as possible. In this way, the hash function of the combination of all the search keys cannot be used by part of the search keys unless a new hash index is created.

**(c) Efficiency with Indexing**







*Figure 14: Runtime before and after creating index*

It is clearly proved that creating index has increased the efficiency of queries by 40 to 50 times or even more.

**Conclusion & Self-Evaluation**

In this project, our group has designed a database for DJI company in relation to the retail and manufacture of Unmanned Aerial Vehicles (UAV). First of all, we did basic research about what the company is doing and the its requirements of the database, coming up with a large number of assumptions which are necessary and reasonable. Based on these assumptions, we then designed the table structure as well as attributes and relationships, and finally drew the ER diagram and relational schema diagram. During this process, many techniques we learned from the lectures were applied. For example, we designed ER diagram and performed normalization following the principle mentioned in lectures. In the programming part, we not only practiced the basic queries, but applied advanced skills including triggers, views, and procedures as well.

After the first version of our database was constructed, we did not forget to further optimize it using normalization theory, finding all the dependencies existed in the database and trying to split the tables to achieve a higher level of normalization without harming the original structure. To make this database more realistic, we retrieved a lot of realistic data relating to products and stores from the official website of DJI and also wrote some scripts to generate other data randomly. We came up with some effective sample SQL queries for the company as well. Finally, we even took the database performance into consideration and introduced the indexing and hashing technics to largely speed up the queries. We weighed the pros and cons of different indexing methods, which could guarantee better user experience.

This project is a great opportunity for us to have a hand-on experience to apply the knowledge we learned from the lectures to simulate the whole process of building a database for a real company. It enables us to have a deeper understanding of not only the basic knowledge about database but the skills to design one as well. Besides, our teamwork skills were also improved at the same time. Every group member has taken an active part in the project from the beginning to the end, contributing a lot from different aspects. During the process, we communicate closely with each other, and work hard together to overcome difficulties. Each group member not only is very responsible for their own work, but also help with others actively. We all did our best to try to develop our project better.

**Contribution**

Li Zeyu (118010158) --- ERD Design, Normalization

Liu Yuxuan (118010200) --- Database Design, Indexing, GUI

Li Yihan (118010154) --- Data Insertion and Query Code, Generating Random Data

Qiu Yutao (118010249) --- Database Design, Data Retrieval

**References**

1. DJI Product and Store Data

Retrieved from:[DJI - Official Website](https://www.dji.com/sg)

1. B+ Tree Data Structure

Retrieved from: <https://en.wikipedia.org/wiki/B%2B_tree>

1. Comparison of B-Tree and Hash indexes

Retrieved from: <https://dev.mysql.com/doc/refman/5.6/en/index-btree-hash.html>

1. InnoDB Adaptive Hash Index

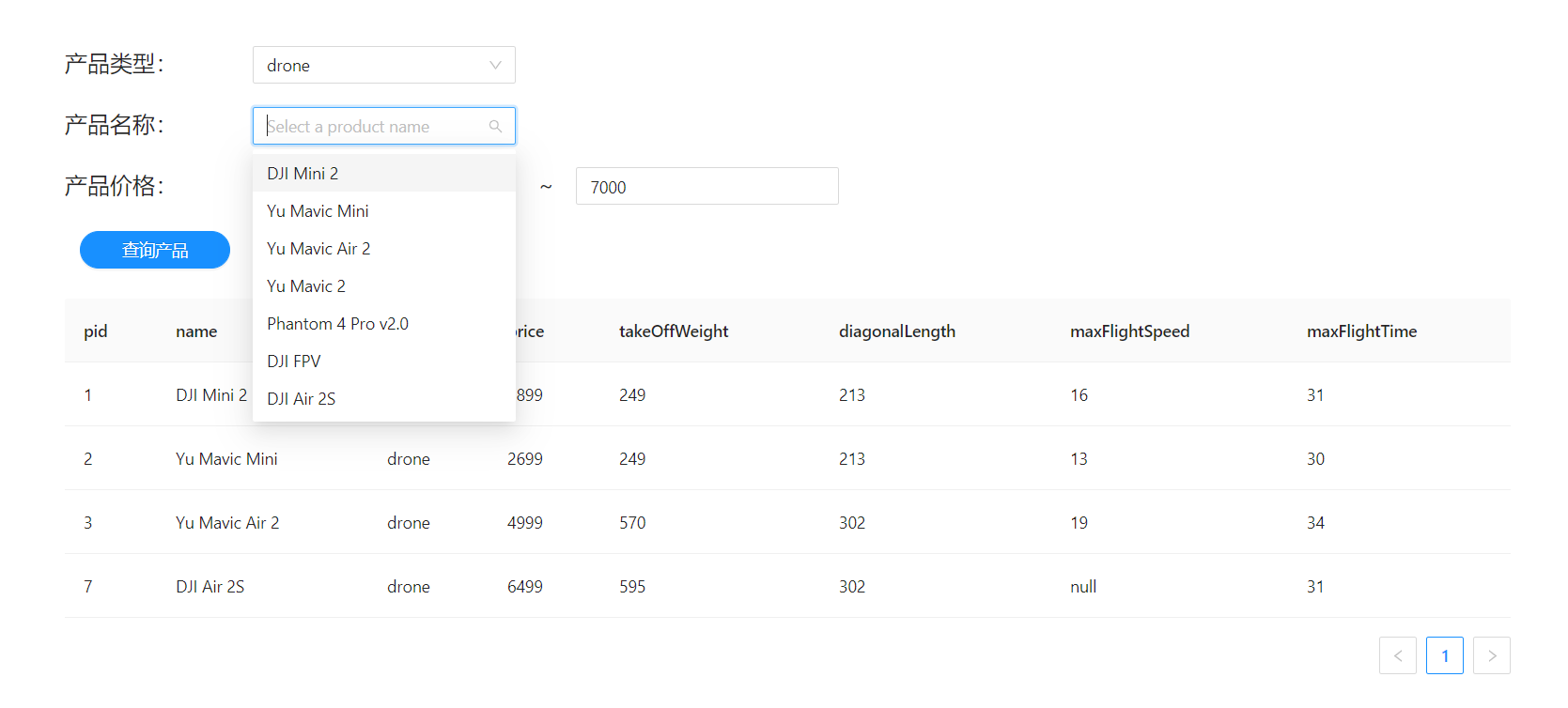
Retrieved from: <https://dev.mysql.com/doc/refman/5.7/en/innodb-adaptive-hash.html>

1. Database Concepts (7th Edition) by Silberschatz, Korth and Sudarshan

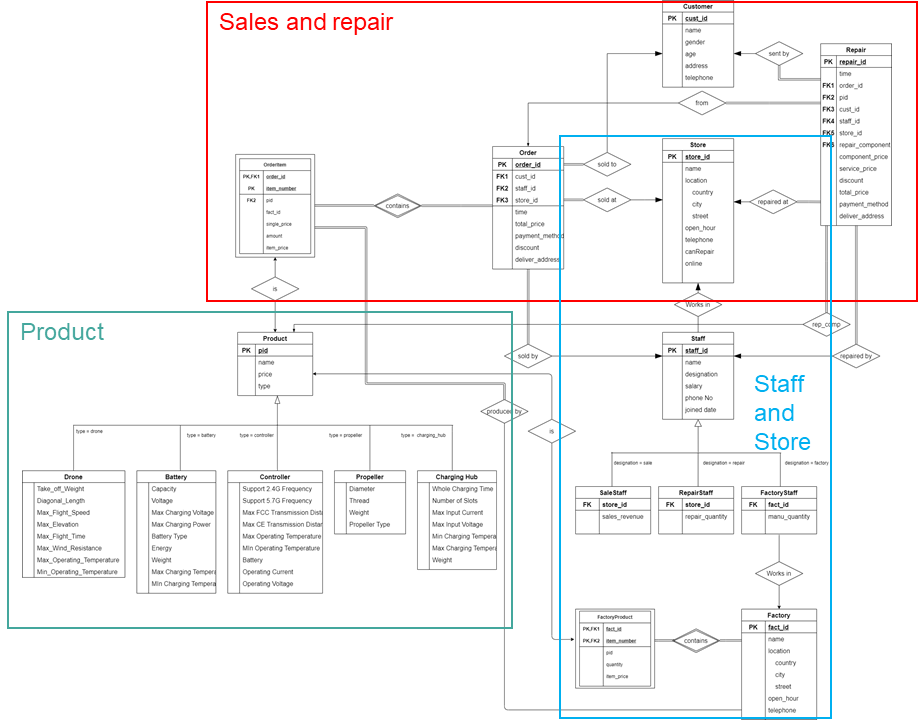
**Appendices**

**Appendix A: GUI**

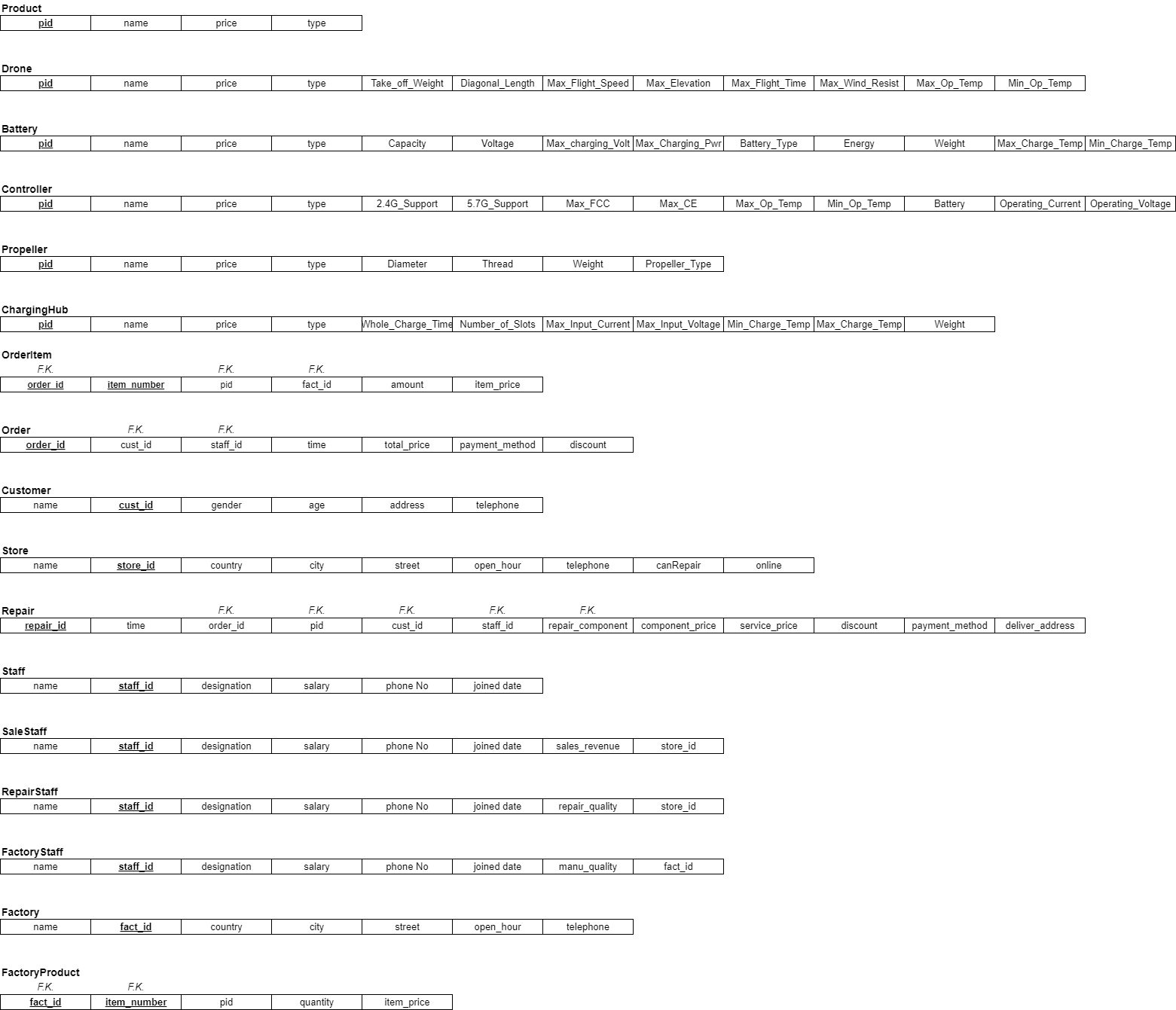




Here, for example, the user can specify the criteria of the target product they want, including which type the product belongs to, the exact product name, the price range. After getting the information, we will search by SQL queries and return the user with the products they want.

**Appendix B: Complete E-R Diagram**

**Appendix C: Complete Relational Schema**



**Appendix D: Data Insertion Code**

See random\_data.py

**Appendix E: MySQL Code**

See sql file.