# Abstract

**Problem statement:**

The problem that we are trying to solve is to create a database for a fictional IT hardware rental company and satisfy its use cases.

**Motivation:**

Our motivation towards this project is to determine if our database design would meet the professional standards that real large scale company would require by applying the theoretical lectures learned in our Database Design class.

**Approach:**

The approach taken to creating this database was to meet the goals for phase 1 and phase 2 so that we could apply what we learned to the synthesis of the database. Many hours were spent translating the design into the actual database and learning how to implement triggers and procedures correctly. For the finalized part of this project, we increased our focus toward the Normalization process of verifying and in some cases redesigning the entities and relations through decomposition.

**Results:**

Our end result was a very sophisticated database that covers our use cases outlined in the section ‘Product Requirements.’

**Conclusions:**

This project has revealed much about designing a database. Careful forethought is required in the early stages to avoid pitfalls that affect the functions of the database when in use. Our initial design was flawed and would have led to inconsistencies when data was inserted, deleted or modified. Through iterations of improvement on the design and by following the Boyce Codd Normalization procedure, our final design produced a robust database schema which will not be vulnerable to anomalies.

# Introduction

Hi, we’re Dynamic Solutions Hardware Inventory company. We loan our equipment to various companies, and therefore need a database to make our every day-to-day transactions easier to handle. This database that we designed helps us keep track of our inventory- whether it is checked out to a customer or bought from a supplier- and to calculate late fees and damage fines in case we have a customer that doesn’t play by the rules. Much of the data which is inserted, updated or deleted is carried out via triggers. For example if someone owes an amount on their bill they are marked as having a delinquent account, and once they pay that amount and/or pay for repair services this delinquent account status will be taken off.

## Product requirements

The Actors for our Database are the Manager and Staff of the store.

The use cases related to our business plan are `Manager Adds Inventory`, `Staff Creates Transaction`, and `Manager assigns staff to Repair Services`.

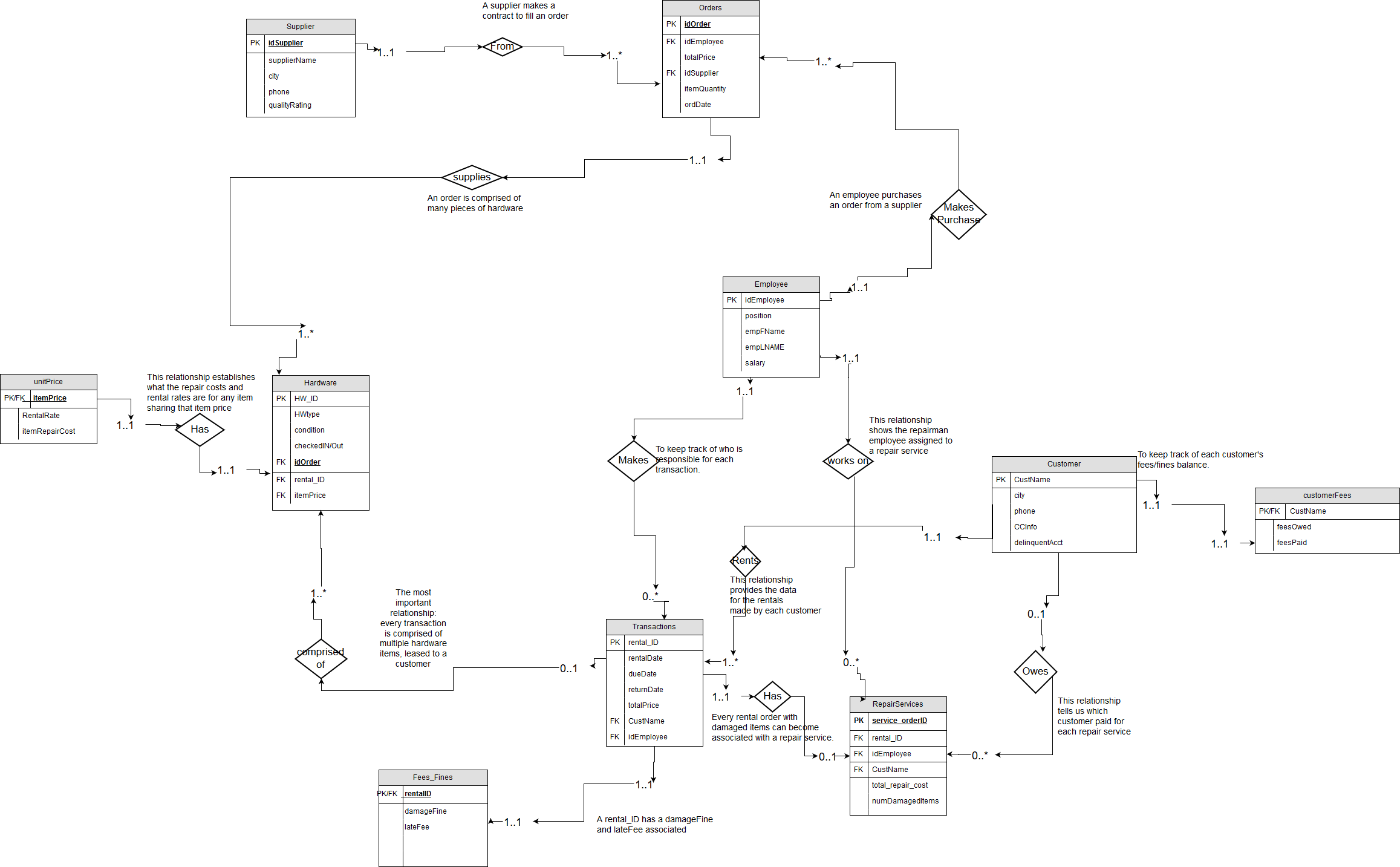
The Aggregate Operation use cases are:

1. Manager chooses “Find minimum salary and maximum salary”
2. Employee selects “Generate report of HW checked in”
3. Employee selects “Generate report of Customers with unpaid fees/fines”
4. Employee selects “Generate report of average price of transaction sale”
5. Manager selects “Generate report on lowest rated Supplier”
6. Manager selects “Generate report on average supply price”
7. Manager selects “Generate report on cumulative amount charged for repairs and total damaged items”
8. Manager selects “Generate report on highest priced item and average item rental rate”
9. Manager selects “Generate report on greatest amount owed”
10. Manager selects “Generate report on total damage fines and late fees”

The Joint Operation use cases are

1. Manager selects “Generate Employee-Transaction Report”
2. Any employee (Staff or Manager/Supervisor) will click on “Show Outstanding orders”
3. An employee selects “View hardware checked out”
4. Manager selects “Who bought supply?”
5. Any employee (Staff or Manager/Supervisor) selects “View items in supply order.”
6. Any employee (Staff or Manager/Supervisor) selects “Show repairman servicing order.”
7. Any employee (Staff or Manager/Supervisor) selects “Hardware in service order.”
8. `fees\_fines\_trigger` in Transactions table updates late fees in fees\_fines table
9. `customerFees trigger` in fees\_fines table updates delinquent account status in customer table
10. `hardware trigger` in hardware table:
    1. calculates price of rental orders in Transactions table
    2. calculates repaircost of service order in repairServices table
    3. calc price and total num for supply orders in Orders table
    4. calc damage fine of rental transactions in fees\_fines table

## ER Diagram



### Relational Model

Supplier(idSupplier, supplierName, city, phone, qualityRating)

Orders(idOrder, totalPrice, idSupplier, itemQuantity,ordDate, idEmployee)

Employee( idEmployee, position, empFNAME, empLNAME, salary)

Hardware(idHardware, type, condition, checkedIN\_OUT, rental\_ID, itemPrice)

unitPrice(itemPrice, RentalRate, itemRepairCost)

Customer(CustName, city,phone,CCInfo,delinquentAcct)

customerFees(CustName, feesOwed, feesPaid)

RepairServices(service\_orderID, rental\_ID, idEmployee, CustName, total\_repair\_cost, numDamagedItems)

Transactions(rental\_ID, rentalDate,dueDate,totalPrice, custName, idEmployee)

Fees\_Fines(lateFee, damageFine, rental\_ID)

## BCNF Verification

### Supplier Table

1NF: This table is in first normal form. Its attributes, idSupplier, supplierName, city, phone, and qualityRating occur only once in each row, i.e.: a supplier never has two or more id's, nor different names; it cannot exist in more than one city; there is only one phone number stored for each supplier and that supplier is only given one quality rating (1-5).

2NF: The primary key of this table is idSupplier. All non-key attributes in the Supplier relation are functionally dependent on its primary key. Because idSupplier acts as the primary key without any other attributes, there is no subset for the non-key attributes to be functionally dependent on. Thus this relation is in second normal form.

3NF: All the attributes in this table are functionally dependent on each other, except for city and qualityRating, which have no dependencies. Therefore all transitive dependencies are cancelled out by the fact that the primary determinant (A) will always be functionally dependent on the secondary determinant (B) in this relation. For example, supplierName would be transitively dependent on idSupplier via phone, except that idSupplier is functionally dependent on phone. Similarly, qualityRating would be transitively dependent on phone via idSupplier, however phone is also a dependent of idSupplier.

BCNF: In order to test whether this table meets the criteria for Boyce-Codd Normal Form, we need to define the determinants to be an attribute or group of attributes such that some other attribute is fully functionally dependent. If all the determinants are also candidate keys, the test is passed. The candidate keys of this relation include: idSupplier; supplierName; phone. There are no functional dependencies that are determined by city, qualityRating or the combination of the two. So the Supplier relation qualifies for BCNF.

### Customers Table

1NF: In the Customers Table each row has one CustName, one associated city, one phone, one credit card number (CCInfo), and one delinquentAcct value. The delinquentAcct value is the most subject to change for a customer, but only one value is kept at a time (true or false). Otherwise it would reflect the history of that customer’s delinquency status (whether they have paid all fines) rather than current status, which is the only thing we are interested in.

2NF: Phone and CCInfo are unique for every customer and therefore, fully functionally dependent on the primary key, CustName. There are multiple delinquentAccts set to true and multiple set to false, but each customer is has only one value at a time for this attribute, and therefore it is also fully functionally dependent on CustName only. The same argument can be made for city. Thus no attribute is dependent on a subset of the primary key.

3NF: City would be transitively dependent on phone via CustName, however phone depends on CustName, and so the transitive dependency is canceled out. Likewise delinquentAcct could have been transitively dependent on CCInfo via phone, but this was prevented by the fact that phone is dependent on CCInfo.

BCNF: The candidate keys of the Customers table are: CustName; phone; CCInfo. The non-key attributes, city, delinquentAcct, feesOwed or feesPaid are fully functionally dependent on each candidate key. However, these non-candidate key attributes are not determinants of any other attribute.

### Employee Table

1NF: Every person employed has one ID (idEmployee), one first name (empFName), one last name (empLNAME), one salary and one position. None of the rows in the Employee relation violate 1NF.

2NF: The primary key of the Employee table is idEmployee by itself. It does not have more than one subset of attributes, so the Employee table could not violate 2NF.

3NF: Except for the primary key, none of the attributes in this table are unique. There may be more than one employee with the same first name or last name, position or salary. This implies that individually they are not functionally dependent on each other. However the combination of empLNAME and empFName makes a candidate key. The non-key attributes such as position or salary would be functionally dependent on this candidate key via the primary key, but fortunately the candidate key is also dependent on the primary key.

BCNF: The Employee Table has two candidate keys: idEmployee (the primary key) and (empFName, empLNAME). The primary key is a dependent of this candidate key, which is allowed in BCNF. There are no other functional dependencies in this table which have a determinant that is not a candidate key.

### Orders Table

1NF: In the Order table the idOrder is the primary key. This table only has one value for each itemQuantity, ordDate, idEmployee, and idSupplier. Therefore this table satisfies the 1NF requirements.

2NF: itemQuantity and ordDate fully depend on the primary key idOrder, which are the only non-key columns and so this table satisfies the 2NF requirements.

3NF: This table satisfies the 3NF requirements since there aren’t any transitive dependencies. Individually, no non-primary key attributes are dependent on any other non-primary key attribute. For example, itemQuantity does not depend on ordDate, because there can be different quantity orders for that date. The same applies for ordDate not depending on itemQuantity. Therefore it is impossible for any attribute to be transitively dependent on the primary key.

BCNF: This table meets BCNF standards simply because there are no candidate keys except for the idOrder, which is the primary key. That is because, although uncommon, it is possible for there to be multiple orders with the same values for any combination of the non-primary key attributes. An order may have the same totalPrice and ordDate just by coincidence. Neither taken together nor separately do they make a candidate key. Similarly, the combination of idEmployee and idSupplier does not make a candidate key, because any employee may do business with the same supplier more than once. That employee may even purchase orders on the same ordDate, with the same itemQuantity, and the order may or may not have the same totalPrice. The only thing differentiating those two orders would be their idOrder(s).

### Hardware Table

1NF: Every intersection of a row and column has one value for idHardware, type, condition, itemPrice, checkedIN/Out, and foreign keys rental\_ID, and idOrder, so 1NF is satisfied.

2NF: The attributes type, condition, itemPrice, and checkedIn/Out all fully functionally depend on the primary key. The itemPrice would seem like the exception but it’s not derived from the type because we can have multiple laptops that are more expensive therefore the itemPrice is variable.

3NF: For this table we identified that in order for it to meet the requirements of 3NF we would need to take out the attributes itemRepairCost, and RentalDate, and make a new table with a copy of itemPrice. We would get these attributes using the foreign key itemPrice which would be the primary in that new table. This had to be done since RentalRate, itemPrice and the itemRepairCost are all dependent on each other and as a result they were transitively dependent on the primary key. Taking them out of the table resolves this issue. We also had to remove idEmployee, because it was functionally dependent on rental\_ID and vice versa. Because this relationship is already expressed in the Transactions table, we simply deleted idEmployee from the Hardware relation.

BCNF: This table meets BCNF form because the non-key attributes are not candidate keys. The only candidate being idHardware which is the primary key. The non-key attributes cannot influence each other since type, condition, checkedIN/Out can have the same values repeated in any arbitrary combination.

### RepairServices Table

1NF: service\_orderID is the primary key and every attribute only has only one value per row for total\_repair\_cost, numDamagedItems, and the foreign keys CustName, rental\_ID, and idEmployee.

2NF: The attributes tot\_repair\_cost and numDamageditems are calculated outside of this table, and so they are fully functionally dependent on the primary key service\_orderID. The same argument can be made for the foreign keys.

3NF: Besides service\_orderID, rental\_ID is a valid candidate key. This means there is potential for transitive dependencies, but they would be cancelled out because service\_orderID is fully functionally dependent on rental\_ID and vice versa.

BCNF: This table meets the requirements of BCNF since the only candidate keys are rental\_ID and service\_orderID (the primary key). The values of numDamagedItems can repeat without being associated with the same total\_repair\_costs or the same rental\_ID, and therefore these can’t be candidate keys. Similarly, CustName and idEmployee aren’t beholden to a fixed value for any of the other attributes.

### Transactions Table

1NF: Every intersection of a row and column has one value for and the primary key is rental\_ID, which satisfies 1NF. Each rental\_ID has only one value for rentalDate, dueDate, returnDate, totalPrice, CustName, and idEmployee.

2NF: In this table we identify that the attributes are dependent on the primary key therefore it meets 2NF requirements.

3NF: When performing 3NF we identified that we did have transitive dependencies which were that the lateFee was dependent on the following attributes: rentallDate, dueDate, and totalPrice. The attribute feesOwed also had the dependencies of damageFine and lateFee. The attribute feesPaid was dependent on feesOwed, which is another transitive dependency. Therefore in order to make this table into 3NF we had to go ahead and take those attributes out and reference that information using Foreign Keys.

BCNF: This table satisfies BCNF requirements since the only candidate key is the primary key . Since the values for the non-key attributes have a many-to-many relationship with other attributes, they don’t qualify as candidate keys, e.g., a customer and employee can be associated with multiple transactions so they are not considered candidate keys.

### ItemPrice Table

1NF: After revising the Hardware table, we made this table. Every intersection of a row and column has one value for itemRepairCost, rentalRate and itemPrice, which is the primary key here as well as the foreign key to the Hardware table.

2NF: This table is in 2NF since the attributes itemRepairCost and rentalRate are fully functionally dependent on the primary key. This is due to the fact that itemRepairCost is always 50% of itemPrice and rentalRate is always 25% of itemPrice.

3NF: Each attribute in this table is dependent on the other two attributes, so there can be no transitive dependencies.

BCNF: Following the above argument, all attributes in this table are candidate keys which doesn’t break any restrictions on this table not being in BCNF.

### fees\_fines Table

1NF: This table was created after revising the Transactions Table. The primary key rental\_ID is also a foreign key to the Transactions table and every intersection of column and row has one value for damageFine and lateFee.

2NF: The primary key of this table is rental\_ID by itself. It does not have more than one subset of attributes, so the fees\_finestable could not violate 2NF.

3NF: This table is in 3NF since damageFine and lateFee don’t depend on one another. Firstly, damageFine is calculated by a trigger in the hardware table, which joins the hardware and itemPrice tables to find the repairCosts for all items, then it selects the hardware items which belong to the rental\_ID in the fees\_finestable and finally, takes the condition number of each item and multiplies it by the repaircost plus an additional $50 per damaged item’s condition. The sum of all damaged items is now placed in damageFine in the fees\_finestable. Late fee is calculated by a trigger in the transactions table, which subtracts the dueDate from the returnDate of the associated rental\_ID and multiplies the result by totalPrice and 25%.

BCNF: Neither of the non-key attributes make a candidate key, and they are all dependent on the primary, which satisfies the requirements of BCNF.

### customerFees

1NF: After reviewing the Transactions table, we created a new table for feesOwed and feesPaid. The primary key is CustName, which is also a foreign key to the Customers table. Each CustName has one value for feesOwed and one for feesPaid.

2NF: The primary key has no subset that would be a determinant of feesOwed or feesPaid. These two non-key attributes are fully functionally dependent on CustName only.

3NF: You can not determine what the feesOwed are for any CustName via feesPaid, or vice versa. So these attributes are not transitively dependent on the primary key. The feesOwed are calculated by joining fees\_finestable with the Transactions table using rental\_ID. The damageFine and lateFees for every rental\_ID where CustName = customerFees.CustName is summed and stored in feesOwed. FeesPaid is variable because a customer voluntarily pays whatever amount they can up to the value of feesOwed.

BCNF:

The feesOwed and feesPaid are fully functionally dependent on the primary key, and since their combinations may not be unique, they are not candidate keys. This implies that the only candidate key is the primary key, making this table automatically in BCNF

## Aggregate Operations For Each Entity

### Employee

Data from the relation ‘Employee’ is queried to find the maximum or minimum salary.

1. Manager selects “Salary” from the “Data and Statistics” menu
2. Manager chooses “Find minimum salary and maximum salary”
   1. SELECT MAX(salary) AS TopSalary, MIN(salary) AS BottomSalary

FROM Employee;

1. Table is returned showing the the value of the **highest** salary and the value of the **lowest** salary.

### Hardware

Data from the relation ‘Hardware’ is queried to determine of “checked in” pieces of a given hardware type.

1. Employee selects “Inventory” from the “Data and Statistics” menu.
2. Employee selects “Generate report of HW checked in”
   1. Employee is prompted to “Enter the hardware type.”
   2. Input is stored in string hwtype
3. Table is returned showing total number of that type of hardware checked in.

SELECT COUNT(`type`) AS Checked\_IN

FROM Hardware

WHERE `type` = hwtype AND `checkedIN/Out`=TRUE;

### Customer

Data from the relation ‘Customer’ is queried to count the number of Customers who have delinquent accounts.

1. Employee selects “Customer Report” from the “Data and Statistics” menu.
2. Employee selects “Generate report of Customers with unpaid fees/fines”
3. Table is returned showing total number of Customers in bad standing.

SELECT COUNT (CustName) AS TotDelnqAccts

FROM Customer

WHERE delinquentAcct=TRUE;

### Transactions

Data from the relation ‘Transactions’ is queried to find the average transaction sale price.

1. Employee selects “Transaction Report” from the “Data and Statistics” menu.
2. Employee selects “Generate report of average price of transaction sale”
3. Table is returned showing average value of all transactions.

SELECT AVG(totalPrice) AS AvgSale

FROM Transactions;

### Supplier

Data from the relation ‘Supplier’ is queried to find the lowest quality rating of suppliers our company deals with.

1. Manager selects “Supply Management” from the “Data and Statistics” menu.
2. Manager selects “Generate report on lowest rated Supplier”
3. Table is returned showing the lowest 5 star rating.

SELECT supplierName, qualityRating AS worst\_supplier

FROM Supplier

WHERE qualityRating=(SELECT MIN(qualityRating)

FROM `Supplier`);

### Order

Data from the relation ‘Order’ is queried to find the average price of an order that our company buys.

1. Manager selects “Supply Management” from the “Data and Statistics” menu.
2. Manager selects “Generate report on average supply price”
3. Table is returned showing the average price of all supply orders that have been made.

SELECT AVG(totalPrice) AS avg\_supply\_price

FROM Order;

### RepairServices

Data from the relation ‘RepairServices’ is queried to calculate sum of all repair costs and number of damaged items.

1. Manager clicks on “Repair Services” on the left-side menu/toolbox.
2. Manager selects “Generate report on cumulative amount charged for repairs and total damaged items”
3. Table is returned showing the sum of all repair costs and items being repaired.

SELECT SUM(total\_repair\_cost) AS tot\_repair\_cost, SUM(numDamagedItems) AS total\_damaged,

FROM RepairServices;

### Unit\_Price

Data from the relation ‘Unit\_Price’ is queried to find the max itemPrice the company has paid for one unit and the average item rental rate.

1. Manager selects “Inventory Management” from the “Data and Statistics” menu.
2. Manager selects “Generate report on highest priced item and average item rental rate”
3. Table is returned showing the maximum item price when item was bought and average rental rate that our business charges.

SELECT MAX(itemPrice) AS most\_expensive, AVG(`rentalRate`) AS avg\_rental\_rate

FROM unitPrice;

### CustomerFees

Data from the relation ‘CustomerFees’ is queried to find what is the greatest amount of money owed to our company.

1. Manager selects “CustomerFees” from the “Finances” menu.
2. Manager selects “Generate report on greatest amount owed”
3. Table is returned showing the customer who owes the largest amount.

SELECT `CustName`, fees\_owed

FROM `customerfees`

WHERE fees\_owed=(SELECT MAX(`fees\_owed`)

FROM `customerfees`);

### Fees\_Fines

Data from the relation ‘Fees\_Fines’ is queried to find the total amount of damage fines that have been issued.

1. Manager selects “Finances” from the “Data and Statistics” menu.
2. Manager selects “Generate report on total damage fines and late fees”
3. Table is returned showing the total amount of damage fines that have been issued.

SELECT SUM(damageFine) AS total\_damageFines, SUM(lateFees) AS total\_ lateFees

FROM Fees\_Fines;

## Joint Operations For Each Relationship

### Employee/ Transactions

The ‘Employee’ and ‘Transactions’ entities are queried together to produce a list of all transactions, showing the employees who made the orders, sorted by rental date and sale price in the Manager’s view.

1. Manager selects “Generate Employee-Transaction Report” from a toolbox menu on the left of the screen.
2. Manager is prompted to “Enter a start date and end date.”
   1. Start date is stored in INT sDate, end date is stored in eDate
3. A list of all transactions is returned with employee names and IDs for each rental order in ascending order for rentalDate and then totalPrice.

SELECT empFNAME, empLNAME, idEmployee, rental\_ID, totalPrice

FROM Employee JOIN Transactions USING (idEmployee)

WHERE rentalDate BETWEEN sDate AND eDate

ORDER BY rentalDate, totalPrice;

### Customer/ Transactions

The ‘Customer’ and ‘Transactions’ relations are joined to produce a list of all orders by a customer to show the due dates for the orders which have not been resolved (upcoming dueDate or past dueDate but not returned and paid for)

1. Any employee (Staff or Manager/Supervisor) will click on “Show Outstanding orders” on the left-side menu/toolbox
2. Employee is prompted to “Enter a date”
3. Table is returned showing a list of all orders by a customer with due dates prior to the user input date. It also shows the rental dates and due dates for the orders, the total price of each order, the amount of days over-due, customer name and customer phone.

SELECT rental\_ID, totalPrice, rentalDate, dueDate,

`returnDate`-dueDate AS past\_Due, custName, phone

FROM Transactions JOIN Customer USING (custName)

WHERE dueDate < vDate

GROUP BY custName

ORDER BY rentalDate, rental\_ID;

### Hardware/ Transactions

The relationship between the ‘Transactions’ and ‘Hardware’ entities is used to make a list of all HW belonging to one transaction.

1. An employee selects “View hardware checked out” from toolbox
2. Employee is prompted to “Enter rental order ID”
   1. User input for ‘rental Order” stored in rentID
3. Returns a table showing a list of all hardware belonging to one transaction.

SELECT rental\_ID, rentalDate, dueDate,

`idHardware`, `type`, `checkedIN/Out`, `condition`

FROM Transactions JOIN Hardware Using (rental\_ID)

WHERE rental\_ID=rentID;

### Employee/Order/Supplier

The ‘Employee’ and ‘Order’ and ‘Supplier’ relations are joined to allow a manager to view which employees are responsible for buying the order hardware orders, what companies they contracted and how much they paid per order.

1. Manager selects “Supply Management” from Data and Statistics menu on left panel of application and chooses “Who bought supply?”
2. Manager is prompted to “Enter the last name of the employee”
   1. Employee last name is stored in VARCHAR vELastN,
3. A table is returned showing the orders made by one employee and the associated suppliers.

SELECT s.`idSupplier`, supplierName, e.`idEmployee`,

e.`empLNAME`, o.`idOrder`, `totalPrice`, ordDate

FROM `employee` e JOIN `Orders` o USING (idEmployee) JOIN `supplier` s USING(idSupplier)

ORDER BY totalPrice;

### Hardware/Order/Supplier

The relationship between ‘Supplier’, ‘Order’ and ‘Hardware’ is used to implement this use case: an employee views a table showing the hardware attached to a single supply order for a given order.

1. Any employee (Staff or Manager/Supervisor) will click on “Supply Management” on the left-side menu/toolbox and chooses “View items in supply order.”
2. Employee is prompted to “Enter supply order ID”
   1. User input stored in vSuppOrder
3. Table is returned showing order ID, hardware ID, item Prices, hardware types, order Dates and the supplier information for a specific order

SELECT `idOrder`, `idHardware`, itemPrice, `type`, `totalPrice`, ordDate, `idSupplier`, `supplierName`

FROM `Orders` JOIN `hardware` USING(`idOrder`) JOIN `supplier` USING(`idSupplier`)

WHERE `idOrder` = vSuppOrder;

### Employee/RepairServices/Customer

The relationship between ‘Employee’ and ‘RepairServices’ and ‘Customer’ is used to implement this use case: an employee views a table showing the Repairmen working on damaged or faulty equipment in a service order, along with the associated rental data and customers.

1. Any employee (Staff or Manager/Supervisor) will click on “Repair Services” on the left-side menu/toolbox and choose “Show repairman servicing order.”
2. Employee selects “Show repairmen working”
3. Table is returned showing employee ID, employee Last name, position, service\_orderID, total\_repair\_cost, number of Damaged Items, custName, customer phone

SELECT e.`idEmployee`, `empLNAME`, position, service\_orderID, total\_repair\_cost,

numDamagedItems, c.`CustName`, phone

FROM `employee` e JOIN `repairservices` r USING (idEmployee)

JOIN `customer` c USING(`CustName`)

ORDER BY service\_orderID;

### Transactions/RepairServices/Hardware

The relationship between **‘**Transactions’ ‘RepairServices’ and ‘Hardware’ is used to implement this use case: an employee views a table showing all the pieces of hardware belonging to a transaction that has been sent in for repairs.

1. Any employee (Staff or Manager/Supervisor) will click on “Repair Services” on the left-side menu/toolbox and chooses “Hardware in service order”
2. Employee is prompted to “Enter repair service ID”
   1. User input stored in vRepServ
3. Table is returned showing service\_orderID, total\_repair\_cost, number of Damaged Items, rental Order ID, damage Fine total for that transaction, and for each item: HW\_ID, HWtype, condition, item Repair Cost

SELECT service\_orderID, total\_repair\_cost, numDamagedItems,

t.rental\_ID, h.idHardware, `type`, `condition`

FROM `repairservices` r JOIN `transactions` t USING (rental\_ID)

JOIN `hardware` h USING(rental\_ID)

WHERE service\_orderID = vRepServ and `condition` > 0

ORDER BY idHardware;

### Transactions/Fees\_Fines

The relationship between **‘**Transactions’ and ‘Fees\_Fines’ is used to implement the use case: Where if a client returns an item late or if the hardware was returned damaged then we need this information to give specifics of a clients total billed amount.

CREATE TRIGGER `fees\_fines\_trigger` AFTER UPDATE ON `fees\_fines` FOR EACH ROW

BEGIN

SELECT `CustName`

INTO tempCust

FROM `transactions`

WHERE `rental\_ID`=new.`rental\_ID`;

SELECT SUM(`damageFine`), SUM(`lateFee`)

INTO vDmg, vLate

FROM `transactions` JOIN `fees\_fines` USING(rental\_ID)

WHERE CustName=tempCust;

UPDATE `customerfees`

SET `fees\_owed` = vDmg + vLate

WHERE CustName=tempCust;

### 

### Customer/customer\_Fees

The relationship between **‘**Customer’ and ‘customer\_Fees’ is used to implement the use case: A trigger in the customerFees table updates the deliquent account status of a Customer.

CREATE TRIGGER `customerFees trigger` AFTER UPDATE ON `customerfees` FOR EACH ROW

BEGIN

DECLARE owed decimal;

DECLARE paid decimal;

SELECT `fees\_owed`, `fees\_paid`

INTO owed, paid

FROM `customerfees` JOIN `customer` USING (`CustName`)

WHERE `CustName`=new.`CustName`;

IF owed > paid THEN

UPDATE `customer`

SET `delinquentAcct` = 1

WHERE `CustName`=new.`CustName`;

ELSE

UPDATE `customer`

SET `delinquentAcct` = 0

WHERE `CustName`=new.`CustName`;

END IF;

END

### unitPrice/Hardware

The relationship between **‘**unitPrice’ and ‘Hardware’ is used to implement the use case: In order to set the total price of a transaction a trigger is activated in the hardware table. A cursor is used to grab the rentalRate for each piece of hardware assigned to a rental\_ID, as well as the condition and itemRepairCost for that item. The cursor acts inside a loop where these values are summed up. Finally, the Transactions table is updated with the totalPrice for that rental\_ID, the fees\_fines table is updated with the total damageFine for that rental\_ID.

For brevity, this complex trigger is omitted except for the aforementioned cursor. But for the record I will mention that this trigger accomplishes

* 1. calc price of rental transactions and supply orders
  2. calc repaircost of service order in repairServices table
  3. calc price and total num for supply orders
  4. calc damage fine of rental transactions

/\*rentalRate and damage cursor\*/

DECLARE hwcursor cursor for

SELECT `condition`, itemRepairCost, rentalRate

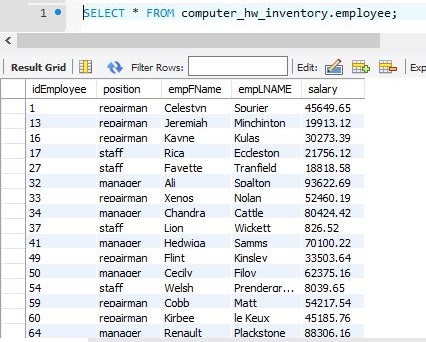
FROM Hardware JOIN `unit\_price` USING(itemPrice)

WHERE rental\_ID = new.rental\_ID;

## 

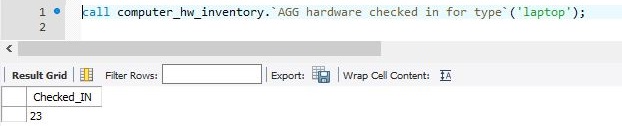
## Test Plan and Records

For each aggregate operation, the table will precede it. For the join operations we will simply show their output. The fees\_fines, customerFees and unit\_price relations are part of triggers, so in order to demonstrate them we can only look at the before-and-after pics to show their effects.

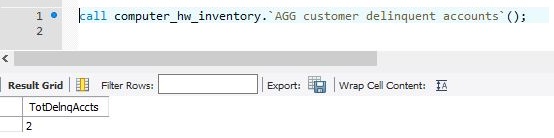


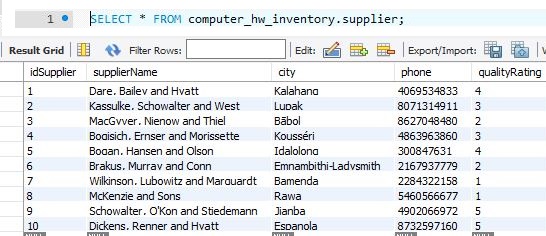


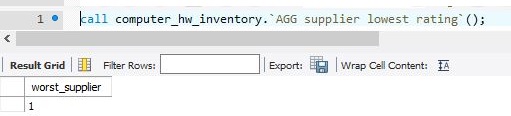


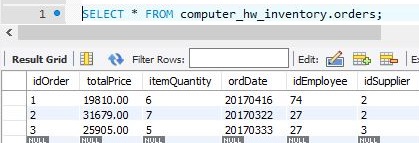


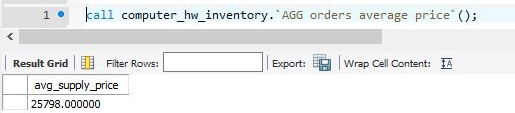


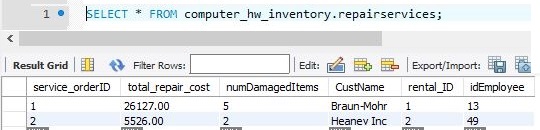


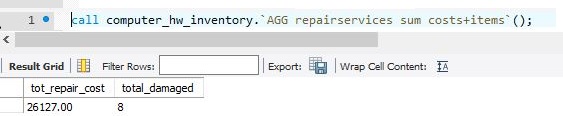


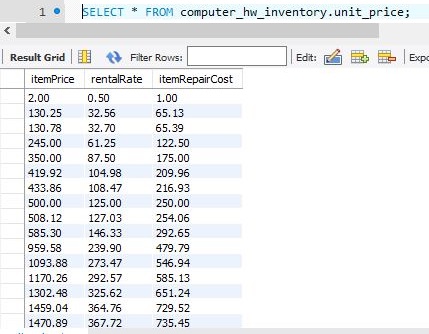


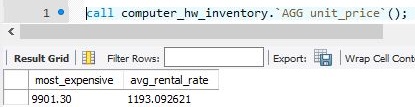


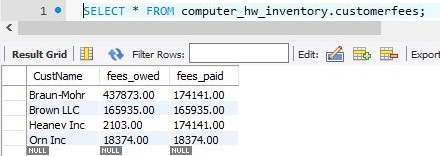


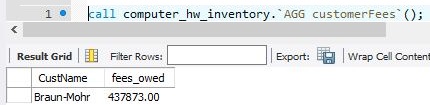


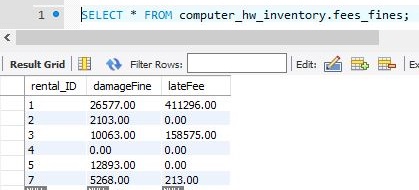


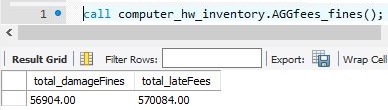






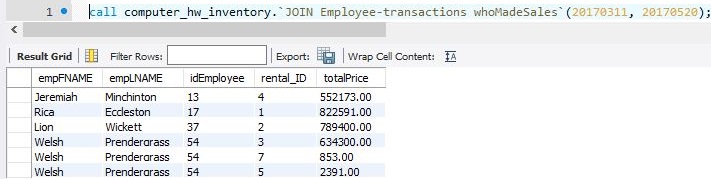


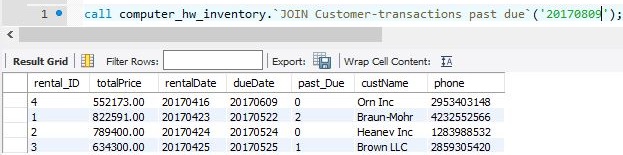


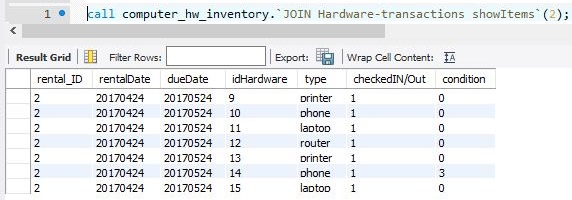


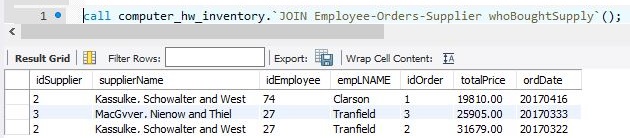
### 

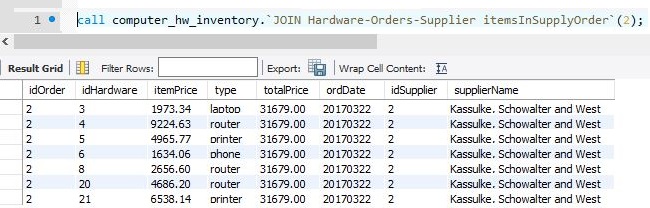
### Joint Operation Test Cases:

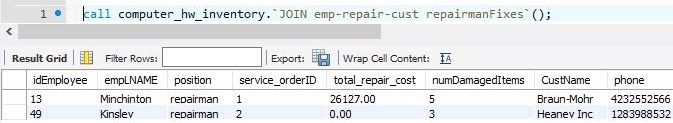


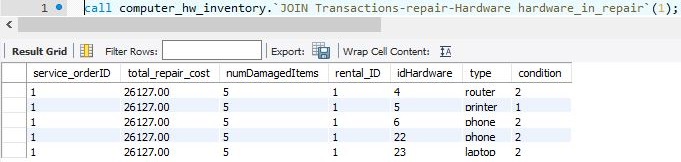












### Trigger Join Operation Test Cases

**JOIN Transactions+fees\_fines**

UPDATE `computer\_hw\_inventory`.`transactions` SET `returnDate`='20170610' WHERE `rental\_ID`='4';

This operation increases the date by one day for transaction rental\_ID=4. The effect is that the lateFee in fees\_fines table for rental\_ID=4 is updated from 0.00 to 138043.00. This calculation is based off (returnDate - dueDate)\*(0.25\*totalPrice) which for this rental\_ID is (20170610-20170609)\*(0.25\*552173.00)=138043.25 rounded down is 138043.00.

**JOIN Customer/customer\_Fees**

The previous test case caused the fees\_owed in the customerFees table for the customer Orn Inc. to increase from 18374.00 to 156417.00 which is equal to 138043.00+18374.00. This caused the Customer table to update the delinquent account for Orn Inc. from 0 to 1 because they owed more than they paid.

**JOIN unitPrice/Hardware**

To demonstrate this use case we will add hardware items to a rental order by updating the Hardware table. The total price of rental\_ID = 2 is 789400.00. By adding idHardware='29', type='printer', condition='0', itemPrice='1800.68', , checkedIN/Out=0 rental\_ID=2

to the rental\_ID = 2, the price increased to 834400.00.

totalPrice is calculated as

rentalPeriod = rentEnd - rentStart;

rentSubtot\*rentalPeriod

## 

## Conclusion

Databases and database management systems are the foundation of the world’s information networks. Our modern society would not be able to function without them. Clearly our database lacks the complexity of a real hardware rental system, but the major difference here is the scope. We have found it difficult to build use cases around the design of the database because some of the relations lacked complexity, or we did not have a business model that was sophisticated enough to make use of that data. However, after performing normalization we discovered it was easier to implement some of our use cases. For example, separating the fees\_fines table from the Transactions table allowed us to update and use this data better.

## References:

1. <https://www.tutorialspoint.com/sql/>
2. <https://stackoverflow.com/questions/>
3. <https://dev.mysql.com/doc/>
4. Connolly, T. & Begg, C. (2015). *Database systems: A practical approach to design, implementation, and management*. Upper Saddle River, New Jersery: Pearson.

.