CS316: Duke Laundry

Sally Al-Khamees, Karen Li, Sara Pak, Rhondu Smithwick

Table of Contents:

Project Description

We developed a Duke Laundry iPhone app that provides users with real-time information about the status of laundry machines on Duke's campus. Users can query for laundry machines based on their location on campus, dorm and laundry rooms as well as status (available, busy, out of order). Users may also view machines based on proximity to their own dorm.

Key Components:

- iPhone Application written in Objective C ("frontend")
- mySQL database stored on AWS EC2 ("backend"); Administratively managed through phpMyAdmin and mySQL Workbench
- PHP API that controls access to database
- Table data generation (Java)
- Simulation data generation (Java)
- Implementation of simulation (Python)

Problem Description and Motivation

Speed, ease of use, and accuracy.

- Students on campus waste time and effort running back and forth between their rooms and the laundry room simply to check if there are enough available laundry machines
    - Especially a big problem for students who must go outside to other dorm houses if their own dorm house doesn't have a laundry machine.
- Real time and convenient information on status of laundry machines. If the machine is broken, can inform many people at once in real time (and hopefully get it fixed faster).

- Interesting because this has never been implemented before.
- This is something that most residents on campus can relate to and use immediately.

Survey:

- Laundry Alert
    - https://www.laundryalert.com/cgi-bin/du9458/LMPage
    - Laundry alert on Central Campus, but is not very robust/ inclusive of all laundry machines across campuses
    - Does not appear to fully work
    - Implies/ Confirms our assumptions
    - We base our schema on this website in part
- Duke ePrint Application

Assumptions

We assume the following:

- Laundry status is tracked by the individual machines, which can identify themselves via a unique machine id (not by the DukeCard payment module -- which would make it impossible to know when the laundry machine is actually started since user could press "start" on the machine anytime after payment). This assumption is based on the fact that even if quarters are used to pay for the laundry machine (i.e. no contact at all with the DukeCard payment module), the laundry shows up as unavailable on the DukeCard payment module interface, which tells us that there is some sort of identification of the specific machine and network connection between the machine and overall laundry system.
- We assume duration and status of machines can be dynamically determined by the laundry system (e.g. a washing machine would be able to provide information with its status and duration when a student begins to use it)
- When a student swipes on the DukeCard payment module to use a particular laundry machine, we are able to figure out what laundry machine and student netid to tie together (e.g. student ab123 swipes to use washing machine A3 (unique id 20). We now know that ab123 is using machine 20).
- We assume that when Duke actually implements a laundry app, they will have prepopulated Students, MachineUse, and Machine tables, or at least the information to fill them. All these tables are relatively static (generally, students will have to be updated each year and MachineUse/Machine rarely). Therefore, we assume that an initial insertion of all this information will be sufficient.

<u>The Database:</u>

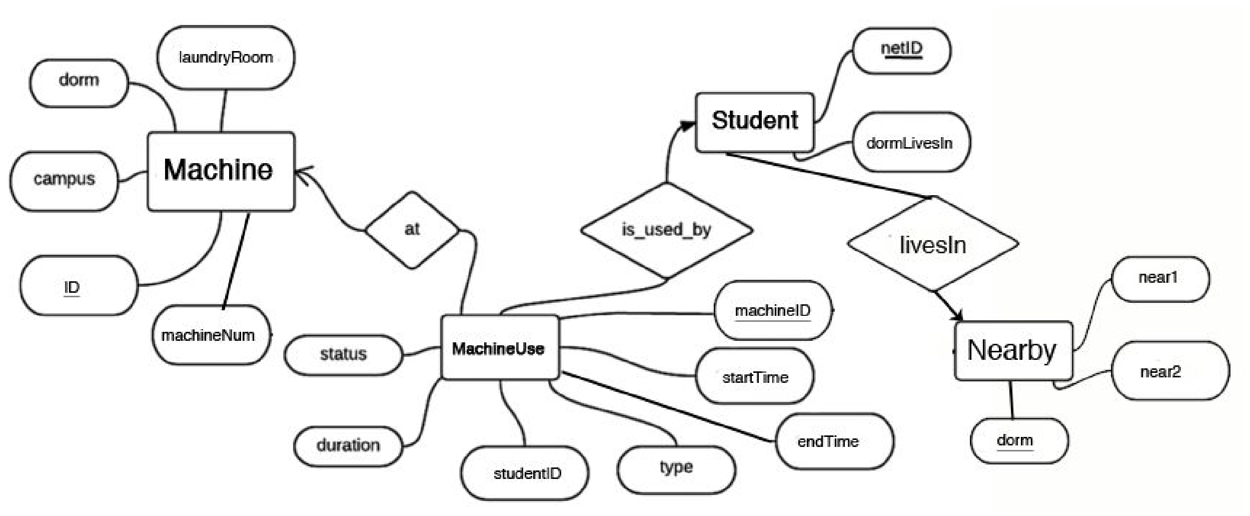


Figure 1: The Theoretical Design of the Database (E/R)

We describe here the database and the design choices behind it. For the actual insertion statements, please see the Creation.sql file.

1. Machine- Stores general information about each machine

- Attributes: <u>ID</u> (*string, primary key*), machineNum (*string*), campus (*string*), dormName (*string, foreign key to NearDorms(dorm)*), laundryRoom(*string*)

Table size: 1300 tuples.

(****Note**: this number is a slight overestimation of the actual number of machines on campus. We estimate there would actually be about 960 machines-- 96 houses on campus posted on the university housing site. Though for this simulation, we say that each house has 2 laundry rooms, each with 10 machines (4 washers and 6 dryers), this isn't the case in reality. Not every house has a laundry room. We overestimate so we can test the database performance on a heavier load to be a more robust platform.**)

<u>Attribute description:</u>

- ID: a unique identifier for each machine.
- machineNum: (A1-A4 for washers, B1-B6 for dryers). Modeled after machine numbers found on laundry machines on Duke campus.
- campus: the campus where a machine is (West, East, Central)
- dormName: references NearDorms(dorms)
- laundryRoom: which laundry room the machine is in.

2. MachineUse- Stores all machines and how they are being used

- Attributes: <u>machineID (*string, foreign key*)</u>, machineType (*string: washer/dryer*), status (*string*), startTime (*double*) , studentID (*string, foreign key to Student(netID)*), duration (*double*), endTime (*string*)

(****Note**: As discussed with our primary TA, Stephen, we considered having a separate table to keep track of MachinesInUse. The two tables would have been: Machine(<u>id</u>, status, machine_type, duration) and MachinesInUse(<u>id</u>, student_id, start_time, machine_type). Duration stays with machine since the duration for how long a machine runs is directly tied to the type of machine it is (washers = 29 minutes, dryers = 60 minutes). While this might save some "NULL" spaces compared to our current single MachineUse table, we realized having them consolidated into MachineUse is better in terms of query efficiencies. If we have both Machine and MachinesInUse, we would have to delete and insert rows into MachinesInUse whenever a machine begins to be used and update status in Machine. When the machine is completed, we would again have to update both tables. With MachineUse, we just need to update the status on the MachineUse table-- this turns out to be a lot more efficient and therefore, is why we have decided to stick with having a single MachineUse table.**)

Table size: 1300 tuples. (Note: again an overestimation per explanation for table 1)

Attribute description:

- machineID: Same as Machine(ID)
- machineType: washer/dryer
- status: machine status can be either available, busy, or out of order (ooo)
- studentID : netID of student using machine. When status is available or out of order, studentID is set to NULL.
- startTime: time when a machine begins to be in use. The startTime is defined as the number of seconds that the current time is past 12AM. The PHP API does this calculation.
- duration: The time of a cycle. This attribute depends on the machine type (dryer's cycle duration = 60 minutes / washer's cycle duration = 29 minutes). These values are modeled after cycle duration for laundry machines used on Duke campus. (Note: washers can take 25, 27, 29 minutes based on cycle options but for the purpose of simulations, we are sticking to the maximum duration of 29 minutes).
- endTime: a formatted time "hh:mm AM/PM" string that the App displays to the user. The PHP API generates this.

3. Student(netID (*string, unique*), dormLivesIn (*string*))

Table size: 6000 tuples. This number is modeled after Duke University's undergraduate current population of 6,626 students.

Attribute description:

- netID
- dormLivesIn: the dorm house where the student lives.

4. NearDorms(dorm (*string, unique*), near1 (*string*), near2 (*string*))

Table size: 65 tuples. This is based on the number of listed houses on Duke's housing website.

Attribute description:

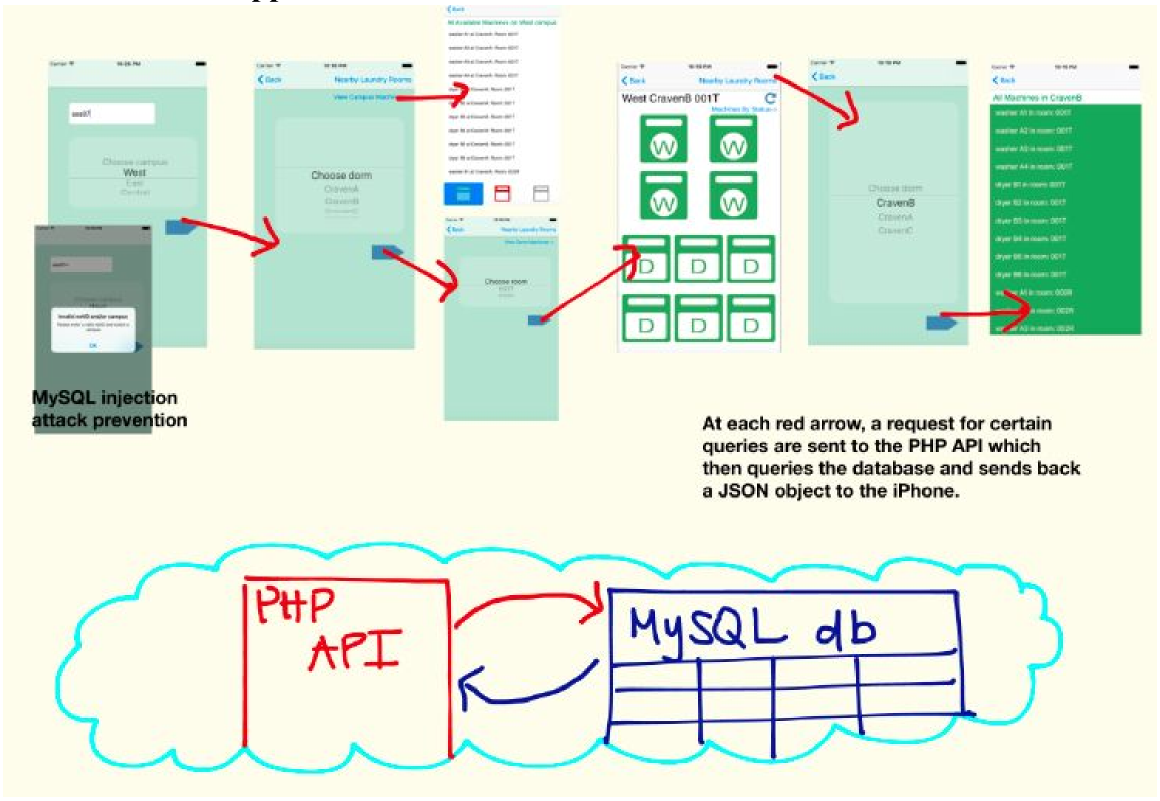
- dorm: a dorm house (linked to what students live in that dorm in Student table)
- near1: a dorm house nearest to the "dorm"
- near2: another dorm house near the "dorm"

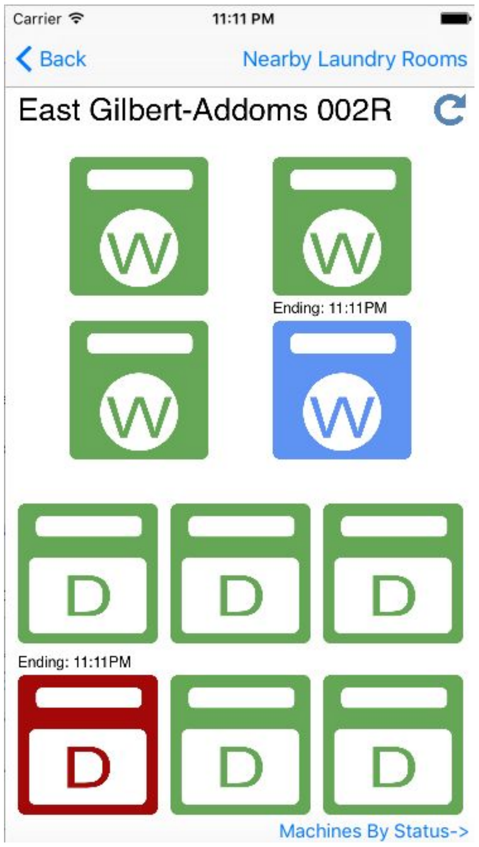
Database optimizations? There was no need for extra optimizations such as further indexing or transaction specifications for our tables given that the default primary key indexing was efficient and fast already. The greatest bottleneck throughout the platform is all the requests that are sent to the PHP API, but we resolve this in both the Python simulation runner and iPhone app through the use of threads and completion blocks.

System / Design choices

- iPhone app: based on an informal survey, students would rather check for statuses on their phones than a website (app will serve this purpose better than a web application).
  - Also allows for easier integration into Duke's current iPhone app
- AWS/EC2 for our database webserver and service to make the project more realistic (rather than having database locally)
  - mySQL database lives on EC2
    - Can robustly manipulate database using PHPMyAdmin or mySQL Workbench
    - Graders provided instructions on how to access
  - PHP API
    - Allows for communication between the database and the frontend / Python simulation
    - One of the files (service.php) is publicly accessible. It is a skeleton Web Service that does not include any real code that would provide information. This is for security reasons. Service.php requires servicehelp.php.
    - The other (servicehelp.php) is not publicly accessible and contains the real Web Service code, including a main function, a runQueries function, database connection info, and queries.
- Python simulation
  - A program that simulates many people using the laundry machines across campus
  - Please see the program (Simulator.py) for more documentation
  - Data for simulation created in Java

Screenshots and App Flow


Choose campus
West
Choose dorm
Choose room
West CravenB 001T
All Machines in CravenB
MySQL injection attack prevention
At each red arrow, a request for certain queries are sent to the PHP API which then queries the database and sends back a JSON object to the iPhone.
PHP API
MySQL db



Close up of screen with machine statuses in a specific laundry room. Green = available, Red = busy, Blue = User of the iPhone app is using the busy machine, Grey (not shown since it happens rarely to be realistic) = Out of Order. Any busy machine has an ending time displayed above the machine, which is when the machine finishes its cycle. Note that for the purpose of our simulation, we are using machines for cycles of seconds (6.25 seconds of simulation = 30 minutes in real life) so the ending time will generally be within the same minute during which the machine was started. The endTime will adapt correctly as is if real data were supplied by the Duke laundry machines.

New Approaches/ Algorithms: Our Simulation

Having the actual machine information and activity would have been ideal. But because that was not the case (Duke OIT doesn't have the API for sharing laundry machine information yet as Dr. Yang helped us find out), our team spent additional extensive effort generating the data and simulation to play the role of thousands of students accessing hundreds of laundry machines all across the campus as follows:

- PART I: Data generation
    - Java to create Machine (for every house on campus, 2 laundry rooms, each with 4 washers and 6 dryers)
    - Java to create MachineUse synced with Machine
    - Java to create student netIDs and distributing them in dorm houses across campus
- PART II: Simulation generation
    - GenerateNetids.java requires input of dorm data including the dorm name and its size
        - Generates netIDs of students within each dorm in text files that are later passed to generate simulation data of laundry rooms
    - MachineUsageSimulator.java class randomly generates when a student is using a machine
        - First, determines whether a student in a dorm is currently using a washer or a dryer
            - usingMachine()
        - Next, the simulator generates a time stamp for when the student is using the machine in intervals of 6.25 seconds (which represents 30 minutes of a 24 hour day

in our simulation). We could have had a random timeStamp, but we clumped it all together every 6.25 seconds so that we could show how our database works under a lot of strain. Furthermore, it would be hard to see changes in 1 second intervals. This allows the simulation to display washer/dryer machine users over 30 minute (real-time) intervals, an interval we chose that generally represents the frequency that machines may be used. An interval that is too short or too long would either have an insignificant number of updates, or a very large number of updates during our simulation demo.

        - Then the simulation randomly generates a machine id associated with a machine id in the Machine table, and subsequently determines whether the machine being used currently is a washer or a dryer based upon its id
        - Also includes the duration of each machine, depending on whether the machine is a washer or a dryer
            - generateMachineDuration()
    - The Simulation also simulates how machines switch to the out-of-order state, and the time at which this specific machine returns back to its available state.
        - generateMachineActive() method
        - Simulates a 5% chance that a machine will switch to out-of-order
        - Simulates that a machine returns to its available state 75 seconds later (which represents 6 hours in real life)
- PART III: Simulation implementation
    - Please see the Simulator.py source code and its documentation for details
    - First creates a mapping of timeStamps to instances of a Machine class. Then works over 5 minutes, using threading to access web service (since upwards of 90 urls must be accessed within a second)

<u>**Evaluation of the system**</u>:

- There are no other systems like this currently implemented within Duke.
- In terms of performance and efficiency, everything is working at a pretty fast and timely manner. The time between Python simulation queries to the PHP API, and PHP to mySQL queries and modifications to ultimately displaying changes on the frontend take less than a matter of seconds and are all done in a very streamlined fashion.
- Using multiple threads: sending requests to the PHP API in the Python simulation and frontend code on different threads to allow for many executions to take place efficiently.

<u>**Security**</u>

1. SQL injection attacks:
    a. Can occur when user is asked to type in their netID when they first open the laundry app.
        i. Sanity check implemented to ensure that no characters other than the 26 English alphabet characters and numbers 0-9 are used. Also ensure length of the netID < 7 (based off of Duke's netID system)
        ii. Prevents any use of SQL query-like input which generally needs the use of some comparator like = and may be lengthier than a standard netID
    b. Can occur if someone figures out the link to send a request to the PHP API

i. Keep the request to PHP API link safe and secure (Ex. in our scope, no leakage beyond the 4 members of our group and our project graders)
ii. Also, even with access to link, can't submit any queries that will result in major catastrophes like deleting the whole DB-- PHP API written to not support such deletions.
iii. PHP API code is not available to the public since the real code is stored in a non accessible location. The code available to the public is just a skeleton.
iv. The PHP user (the database user present in the PHP code) is only allowed to SELECT from tables and UPDATE MachineUse. The user with all permissions is kept secure and will only be shared with the project graders.
v. Lastly, the data that are returned and manipulated by the PHP API are ultimately not very sensitive data -- netIDs are public (through emails we send with our duke.edu emails), machine information stored in table are not very helpful for attackers, especially financially or politically

Open issues / directions for future work:

- Great to have this actually implemented and working with the real laundry machines when that API gets built out
- No longer implementing "sending email notifications when the machine a student is using is done" due to resource limitations-- we have created a whole table of fake student netIDs (which would be ideally attached to netID@duke.edu to know what email to send the notification to). But given that all these netIDs are fake, we can't send emails to all these "fake" emails. Furthermore, it may be better to simply send push notifications on the iOS device which is best implemented using tools provided by a paid developer kit (but we didn't feel the need to invest $100 for this sole purpose)
- No other real open issues besides the fact that everything is based on our homemade simulation