Book-A-Study Room

Deliverable #6

November 12th, 2018

HAXORS

Eliza Gaudio (1554032)

Elie Sader (6128748)

Jonnie Klein Quezada (1640380)

Kevin Hirsh (1634710)

Kyle Nancoo (1441915)

Client organization:

Vanier College Library

Client name:

Haritos Kavallos

Table of Contents

Executive Overview

In this deliverable, you can find a design of our future database system for our prototype. We included diagrams to illustrate our thoughts, as well as in depth descriptions of the relative drawings. Moreover, we did some math to project how we see our database growing if it were to continue growing over the next 3-5 years, and how the data would change accordingly.

Throughout the narrative description section, we have block diagrams that demonstrate who are users (actors) are for this prototype, what tasks each of those actors perform, and which table they will represent in our created database system. This section has a description of what our prototype is supposed to do, and what we intend on creating for this project, and for our client.

In Appendix 1, we used MS Visio to produce an ER diagram that describes our prototype's database design. Throughout this appendix, and illustrated in the diagram, you'll see the multiplicity constraints between each different table, and the primary keys/foreign keys will be identified on the diagram itself, to show which attributes are primary, and which ones are foreign. The types of the attributes were also identified (i.e. if the attribute is varchar, int, etc.).

In Appendix 2, you can find information about each entity in the diagram, and an explanation as to why we chose to identify the attributes with their particular type. We have breakdowns of each table, with their respective entities and attributes, and below the tables are descriptions of what the data in them actually mean.

In Appendix 3, we are projected to calculate how our database will expand over the next 3-5 years, and what the size is anticipated to be. In this section, you can find the calculations we performed, in order to figure out how exactly our database is estimated to expand. We used a sample of approximately 7000 Vanier students to figure out the math.

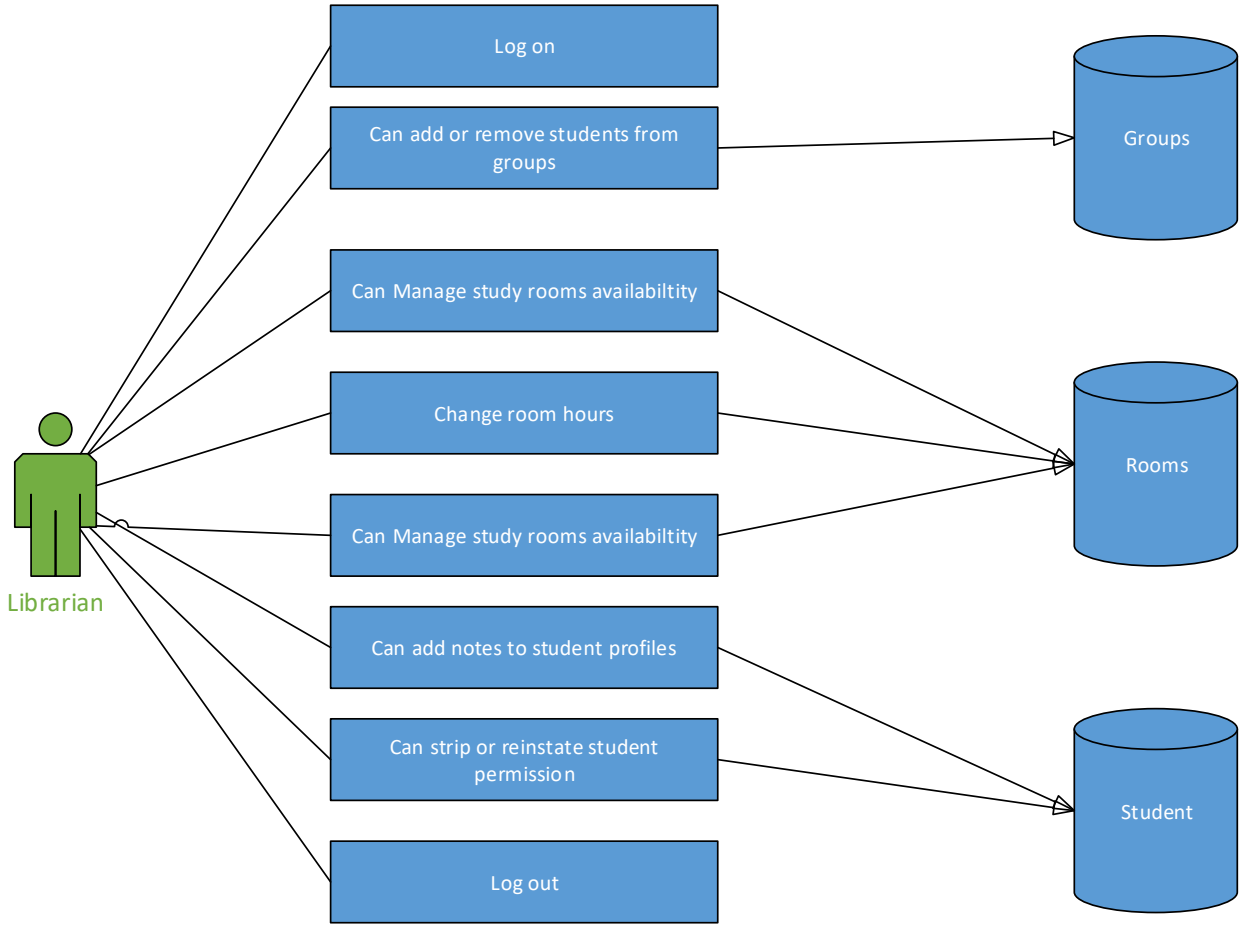
In Appendix 4, we described the access speed that'll be required to run our prototype, and how long we expect the runtime to be. With a proper functioning OS, our database system/prototype should be fairly straightforward and easy to use.

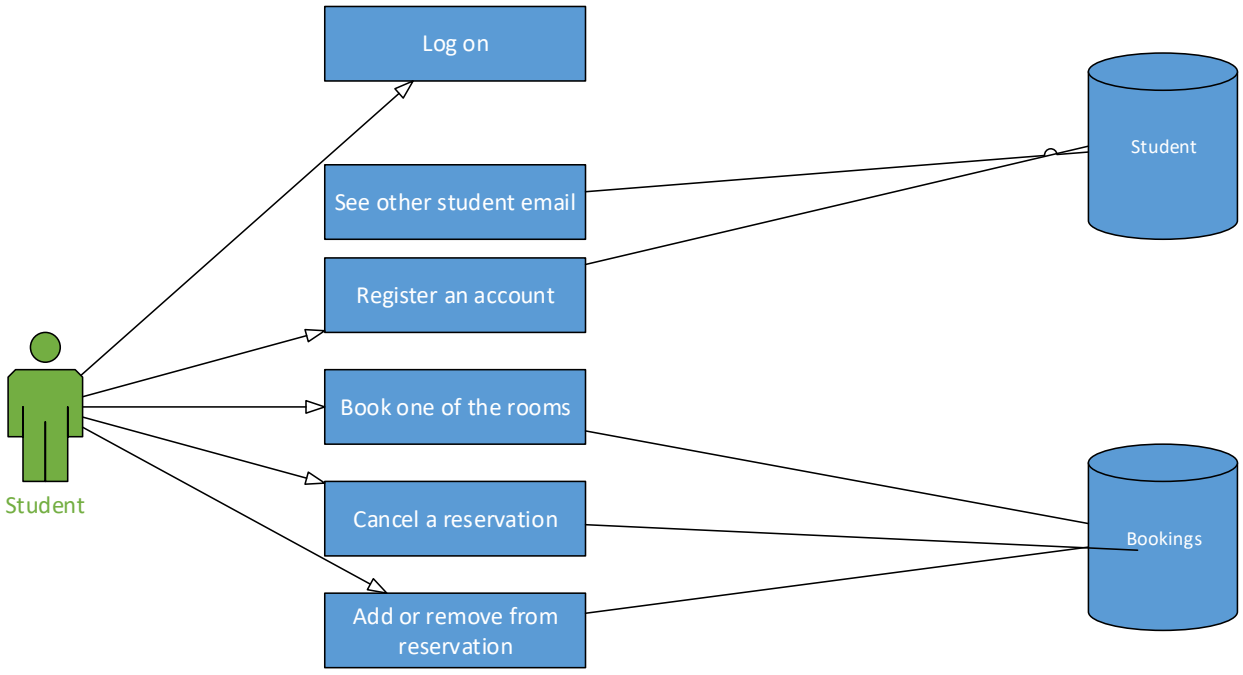
Narrative description of database design

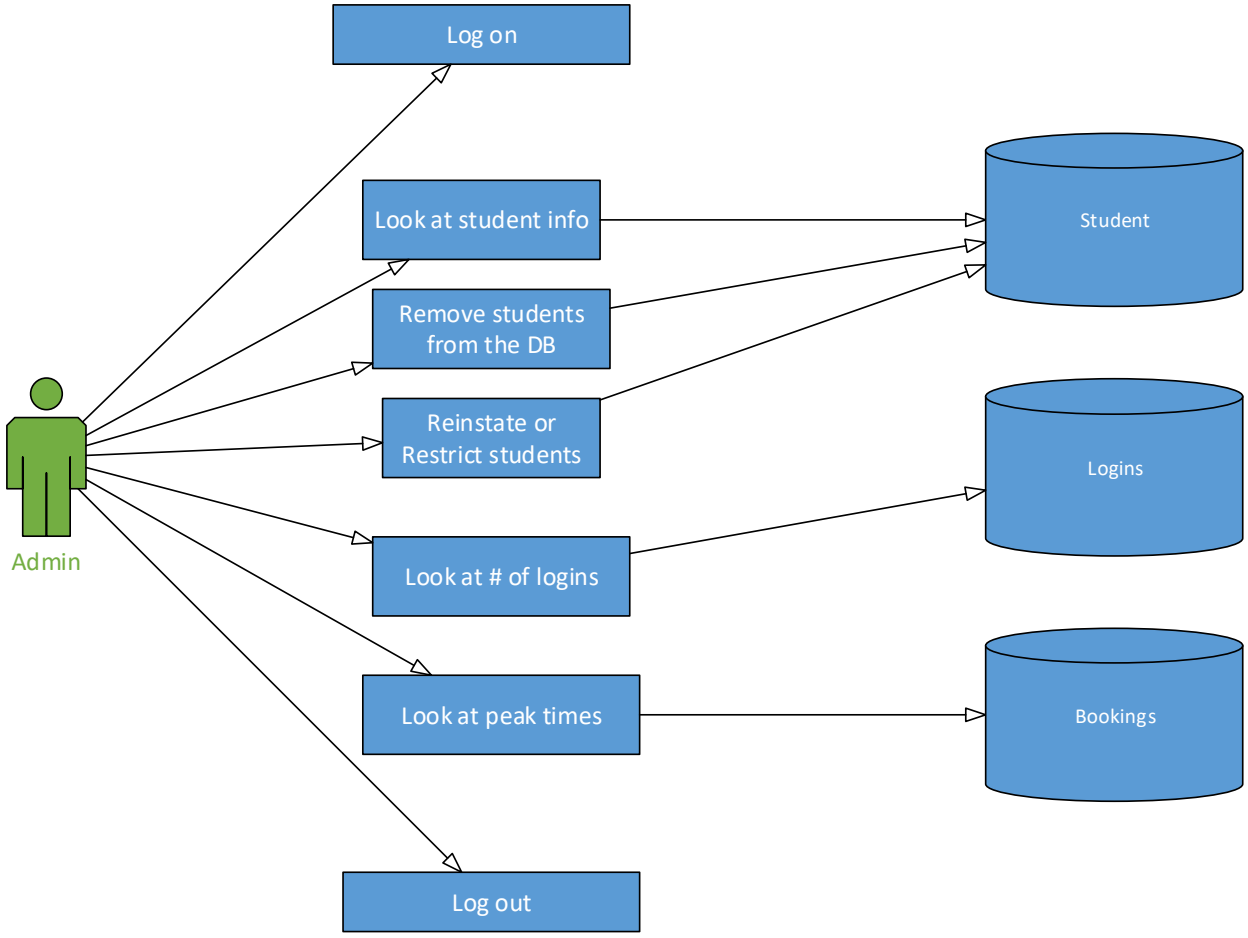
Our intention for this prototype is to generate a website that will allow students or teachers to book themselves a study room at the library. Our purpose is to give them a chance to book ahead of time, and plan their study sessions. Librarians will have more in depth access to the system, because they need to see student information, and they are able to modify information, if need be. The admin, which is Haritos, has the most access to the system, being able to view the analytics, and perform important actions (i.e. deleting a student from the database when they graduate). Our database design supports all of that.

We have 4 actors taking part in this system. They are the admin, students, teachers and librarians. Each actor has their respective task, differentiating the importance of each of them. For example, the admin is able to look at how many times a particular student logs in to book a room, and they can see what times are at peak for study room bookings. A regular student would not have access to that type of information, and therefore that would not be one of their tasks.

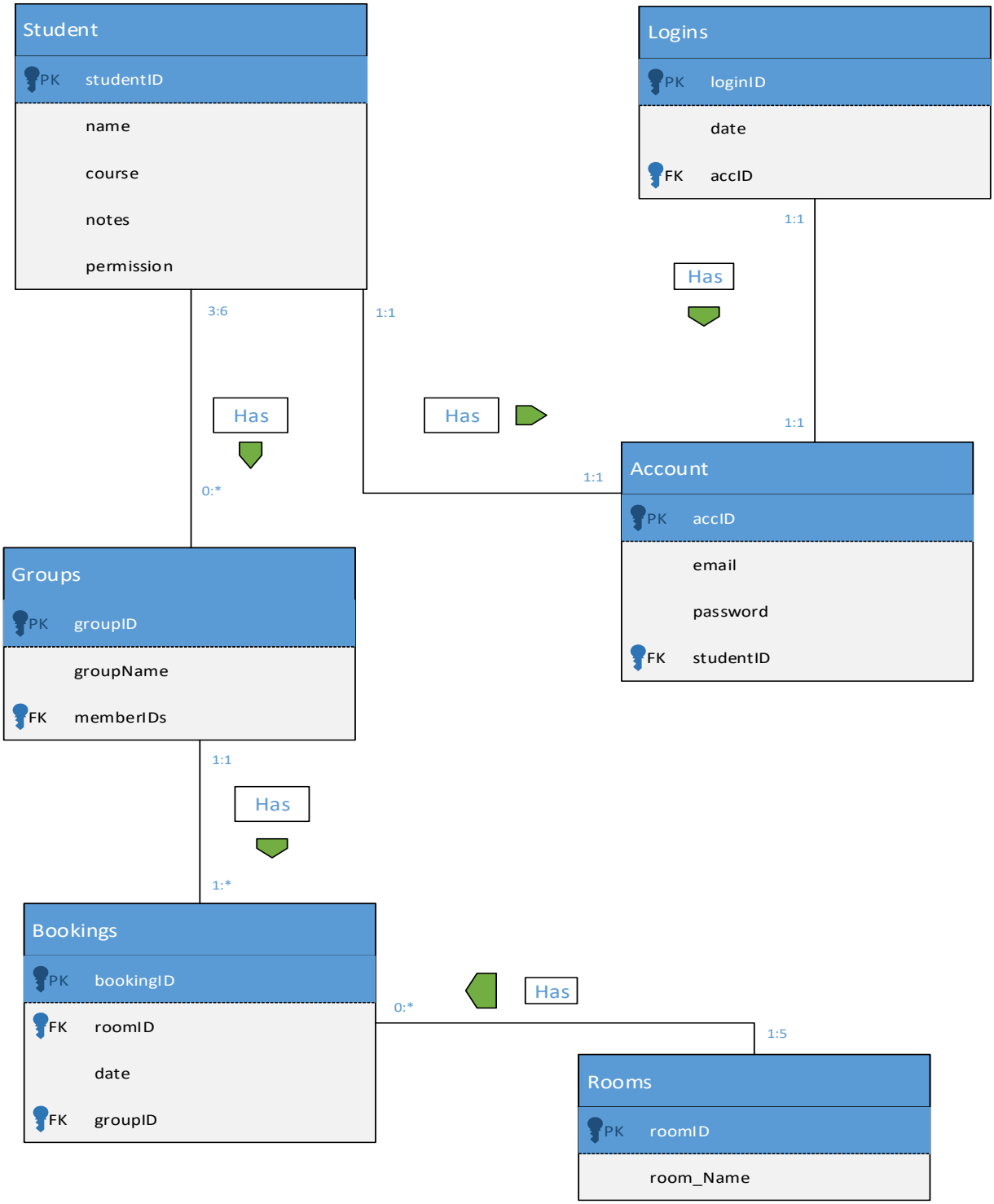
The block diagrams we created, found in the images below, show which task that actor performs, and it shows the relation to which database table it connect to.


Log on
Can add or remove students from groups
Groups
Can Manage study rooms availabiltity
Change room hours
Rooms
Can Manage study rooms availabiltity
Librarian
Can add notes to student profiles
Can strip or reinstate student permission
Student
Log out


Log on
Student
See other student email
Register an account
Book one of the rooms
Cancel a reservation
Add or remove from reservation
Bookings


Log on
Look at student info
Remove students from the DB
Reinstate or Restrict students
Look at # of logins
Look at peak times
Log out
Admin
Student
Logins
Bookings

Appendix 1


Student
PK studentID
name
course
notes
permission
Logins
PK loginID
date
FK accID
Account
PK accID
email
password
FK studentID
Groups
PK groupID
groupName
FK memberIDs
Bookings
PK bookingID
FK roomID
date
FK groupID
Rooms
PK roomID
room_Name
Has
Has
Has
Has
Has
3:6
1:1
1:1
1:1
1:1
0:*
0:*
1:1
1:*
1:5

Appendix 2

For the sake of our application, we have decided on the inception of a few database tables. The Student table, as its name suggests, will hold records of student information.

STUDENT

STUDENTID (VARCHAR(7))
NAME (VARCHAR(100))
COURSE (VARCHAR(30))
NOTES (VARCHAR(200))
PERMISSION (TINYINT)

We have decided to identify the studentId attribute as a varchar, as opposed to an int. We believe that, as a rule of thumb, attributes requiring arithmetic operations and indexing are the only ones that should be identified as a numeric type. As such, because we will not be performing any arithmetic operations on studentId, name, course, notes and permission, we have decided to stick with a string type, except permission has tinyint, since it would be 0 or 1, permitted or not permitted. Although a hundred characters seems daunting for a name, one shouldn't assume the likely length of a person's name as indicator. In Canada, names may be shorter than a citizen from another country, however, the mere possibility of a person's name exceeding our average proves to be enough of an evidence for us to settle with this number. For the password field, we have decided to encrypt all passwords before passing it on to the database. This means that, using www.npmjs.com/package/bcrypt, we are able to securely store plaintext passwords the user has entered, into a multitude of characters that represents the algorithm used, the algorithm cost, the salt, and the hashed password. Combined, they compose a majestic hash that, based on bcrypt's nature of being an adaptive function, it will remain resistant to brute-force search attacks, even with increasing computation power. We would like to provide an example on how passwords are going to be stored to our database, so to assure our clients that we are a group that firmly believes in the hideous nature and the security risks it poses in storing passwords as plaintext: $2b$10$zpdyt7.ZcUq7J8hjQqmIVeS6o6ejwt3KkoSRk4MdXvuI7fTk5Nusa. As mentioned earlier, the combination of the algorithm used, algorithm cost, salt

and hashed password composes this, derived all but from one of the most commonly used password in the world: “password”.

LOGINS

LOGINID (VARCHAR(7))
DATE (DATETIME)
ACCID

We have decided to set the LOGINID as foreign key, as opposed to simply stating the name of the logged on user. By doing so, we ensure a link between the logged student’s information from the Student table with the Logins table. As such, it will be simpler to query student information, should we ever need to, by joining these two tables. Thus, we reduce the need for duplicate field values on name, email, course, etc. The Date attribute will hold the day, year and month of the login, as well as the time. This will be crucial to us in order to be able to properly sort the logins by time, should we ever need to. There’s also ACCID, to see what type of account they have.

ROOMS

ROOMID (TINYINT)
ROOM_NAME (VARCHAR(30))

The roomId, of type tinyint, will be serving as the primary key for the Rooms table. We decided with identifying this attribute as a tinyint because we believe, as evidence suggests, there won’t be more than 128 rooms, which is the maximum value that a tinyint type variable can hold, in the very near future. In fact, there are only a total of 4 rooms to this date and using a bigger type would result in fragmentation, thus will turn into a waste of space. The room name will be of type string which will hold the very brief description of the room such as “music room”, “room one”, etc.

BOOKINGS

BOOKINGID (TINYINT)
ROOMID (TINYINT)
DATE (DATETIME)

GROUPID (TINYINT)

The Bookings table will hold records of each booking that has taken place. BookingID will have a number to indicate the reference number for the booking. The roomId attribute will serve as the foreign key referencing the Rooms table, the date will be of type datetime and members will be of type varchar of length 47. The members attribute will be a string containing all the studentId who are part of this reservation. As noted, each studentId is comprised of 7 characters and each room can hold a maximum of 6 students. As such, we will need a total of (7 x 6 = 42) + 5 characters serving as delimiters.

GROUPS

GROUPID (TINYINT)
GROUPNAME (VARCHAR(30))
MEMBERIDS (VARCHAR(35))

The Groups table will hold records of the different types of group bookings that students/teachers can create. The groupID attribute will be the primary key for this table, and it identifies as a tinyint, because members can't book very large groups anyway. Another attribute is the groupName, where the people creating a group have the option of giving their group a unique name to book with. This attribute identifies as varchar with a maximum of 30 characters so that the title of their group isn't excessively long. Our final attribute for this table is memberID, which is a foreign key, and identifies as varchar with a maximum of 35 characters, since there's a maximum of 5 members that could create a group, and 5 * 7 = 35.

ACCOUNT

ACCID (TINYINT)
EMAIL (VARCHAR(30))
PASSWORD (VARCHAR(30))
STUDENTID (VARCHAR(7))

For the Account table, we have the ACCID, which is the account ID of the person booking, and it's a tinyint because the number will not be that long. This table also has attributes for the email and password, because that's going

to be the user's login information. Email and password both have a maximum allotted slot of 30 characters, which should be more than enough. StudentID is the same as previous, the student will only have to enter their 7 number ID.

We have decided to implement the MVC architecture. That is, our application will be divided into two distinct categories: the frontend and the backend. The frontend will be responsible for the rendering of data, elements, etc. In essence, it serves as the Views of the application. The backend will serve as the Controller. It will handle all logical operations of the application and will be the primary, if not only, medium between the frontend and the database. The database will serve as the Model of the application, that is, it will serve as a data structure for the program and will dictate what data the application should contain. Should the state of this data changes, the model will then notify the view of its changes and will re-render accordingly.

Although we possess exactly 6 tables, we still know that there will be quite a workload for the database. We believe the Bookings table will be the table that will hold the most data. In fact, with the assumption that each room will have an average of 3 bookings, we can assume that daily, 3 * 4 = 12 entries will have to be added to the Bookings table. This will grow exponentially the more rooms and more bookings daily you have. As such, it is important for us to optimize our queries. We decided on the usage of Secondary Level Indexing. For the Bookings table, we will have an index based on roomId. We believe that indexing by roomId will prove to be efficient over-time as opposed to querying the Bookings table record by record instead of roomId by roomId. Thus, we will be able to provide results with more efficiency on a larger scale query.

Appendix 3

As discussed earlier, we decided to create a Students, Bookings, Logins, Groups, Accounts and Rooms table. We know that three out of those six (Students, Bookings and Logins) will carry most of the load. Before we begin, we must first determine how costly storing one record of a student is. Knowing that each character in a varchar variable occupy one byte, one student record will occupy (7 + 100 + 60 + 30 + 60) 257 bytes, rounded to 260 bytes.

For the sake of this calculation, let's assume Vanier College has over 7k students and that about half of those will be using the study rooms during their stay. For 3.5k records, the Students table will require (260 * 3500) 910,000 bytes, or 0.91 megabytes. Over the course of the next three years, one could assume that we would likely keep student records in this database, however, a better solution would be to move student records who have graduated or are no longer studying at Vanier to another database,

independent and distinct from the Vanier College Booking System database. This will further reduce the complexity of the application and will minimize the work load that should be performed by the database. It will also help keep the number of records as small as possible. With the assumption that Vanier College will continue to grow, we estimate about 8k students by 3 to 5 years. As such, the Students table will then hold about (260 * 4000) 1.04 megabytes.

Each record in the Logins table will hold approximately (7(VARCHAR) + 8 (DATETIME)) 15 bytes. We will assume that each user will have to login twice, once to look for the availability in advance, and once to book (although one could perform these operations in one sitting). We will also assume that each user will perform, on average, one booking a week, which will bring the login number to two logins a week. As such, over the course of a year, which is 30 school weeks, one student will perform, on average, 60 logins which will hold (60 * 15) 900 bytes. For a userbase of 3.5k students, that will be about 3.15 megabytes. If there is an increase in the number of students at Vanier, and assuming that number will be 8k, also assuming that about 50% will use the study rooms during their stay, the Students table will then hold (900 * 4000) 3.6 megabytes of information. As such, depending on whether the administrator of the system decides to move graduate students' information on a separate database or keep them in the same database, the storage will increase by 1.04 MB + 3.15MB a year, not accounting for the entries in the Bookings table yet.

Moving on, another crucial element in the database is the Bookings table. Each booking log will occupy one byte from the roomId, 8 bytes from the date and 47 bytes from the members, which will bring the total to 56 bytes per entry. With the assumption that each room will be booked by an average of three bookings daily, four rooms will hold a total of (4 * 3) 12 bookings daily. Thus, we know that daily, there will be 12 entries added to the Bookings table which will hold a storage space of about (56 * 12) 672 bytes. Over the course of a week, that number will be (672 * 5) 3360 bytes, or 3.36 megabytes. Over the course of a year, it will in turn increase to (3.36 * 30) 100.8 megabytes.

The Rooms table will indubitably remain as the cheapest between the four tables. Each roomId is represented as a tinyint, which means it will be represented in one byte and the description will hold 30 bytes. As such, for each room, 31 bytes of storage space will be used. With 4 rooms, 124 bytes of space will be taken. Depending on how many rooms are added over the 3 to 5 years, we are looking at a requirement of 0.000124MB ± 0.000031MB.

Appendix 4

As previously described, if we decide to keep the entries on this database for years to come, even after a student graduates, then we are looking at a cost of 1.04MB (Students) + 3.15MB (Logins) + 100.8MB (Bookings) + 0.000124MB = 104.99 ≈ 105 megabytes a year.

With an efficient OS, the prototype should run smoothly, without any major issues, and we don't expect our system to be too slow.

Works Cited

- No references used for this deliverable.