CT6013 – Advanced Database Systems

Assignment 1

Dane Nutting

S1211936

Word count: 1729 (tables, figures and figure labels not included).

Table of Contents

MongoDB 2

Non-Relational data 3

Relational data 4

Comparison of Data Structures 6

Dimensional database 8

Analysis 10

Report limitations 10

References 11

Appendix One 12

MongoDB code improvements 12

Appendix Two 14

Further MongoDB code improvements 14

Appendix Three 16

SQL code to answer Decision maker’s questions 16

# MongoDB

This section of the report will explain the choices made when building and querying a MongoDB database that stores student marks. This data will be stored in two different structures, non relational and relational. Tests will be run to record the time it takes each data structure to carry out certain tasks, times will then be analysed and conclusions drawn about each structure’s suitability. First a fully normalised, relational database was designed to help understand what a database for storing student marks looks like:

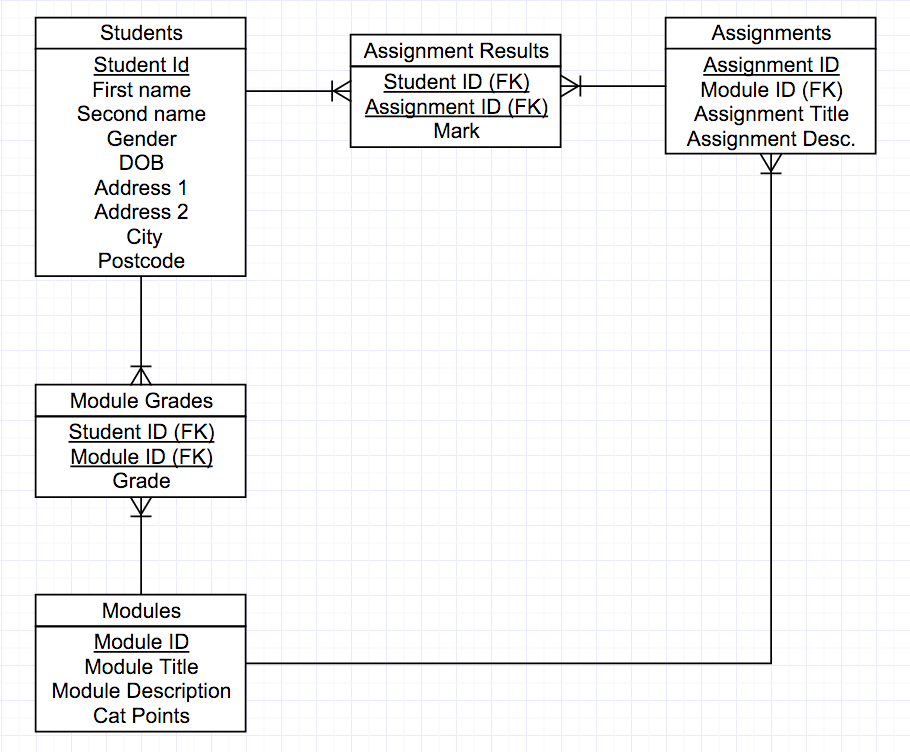


Fig. 1 A fully normalised, relational database for storing student marks.

The important factors that this design shows are:

* The many to many relationships from students to modules and students to assignments necessitates the need for two tables, Assignment Results and Module Grades to house the foreign keys and a measurement field.
* The relationship between Students and module grades could potentially be ignored and a module mark calculated at the time a report is run. However the inclusion of the Module Grades table allows for faster querying and gives a better clarity of data.

## Non-Relational data

This entity relationship diagram made structuring the non-relational data easier because it shows the hierarchy of the data, leading to a straight forward transition to a non relational data structure, as shown below:



Fig. 2 A section of JavaScript showing the structure of student data in Mongo DB.

Student information is the top level of the structure acting as the overall parent. Module information is a level down and this data is stored in an array because there are multiple instances of module data, the MongoDB equivalent of a one to many relationship. The assignment data is a level down again but is structured in the same way as the module data with the module data is the immediate parent. With this data structure understood the next task was to insert non relational test data, initially the insert statement was coded as it appears in Fig. 2 but the module and assignment data later became handled by functions for a number of reasons:

* Inserting multiple assignments per module and multiple modules per student is highly repetitive so was ideal to be written as two functions.
* Doing so saved 100 lines of code for inserting data without an index and saved over 300 lines of code when inserting data with an index creating much more efficient code.
* It allowed for the introduction of further for loops to control how many assignments were inserted per module and how many modules were inserted per student, giving a much greater amount of control over how much data was being inserted. This helped overcome issues where insert files were taking too long to run, because only a variable value needed to change to decrease the amount of data being inserted.

## Relational data

In order to compare data structures identical data was inserted in a relational format, that followed the entity relationship model shown in Fig. 1. Initially the inserts into all 5 tables were handled in separate for loops with no connection between what was being inserted in one table relative to another. For instance, when inserting data into the Assignment Marks table a function was used to find a random module I.D. and that I.D. was inserted into the module I.D. foreign key field. This resulted in some modules having large amounts of assignments associated with them and some modules having very few assignments associated with them. This produced unrealistic test data and was considered unsuitable, so the code to insert assignment data changed from Fig. 3 to Fig. 4.

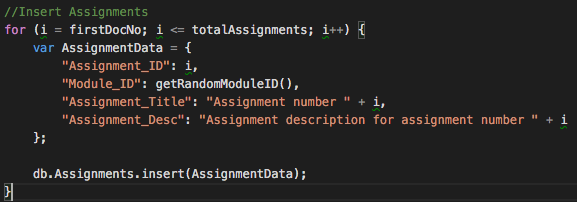


Fig. 3 Original assignment data insert code.

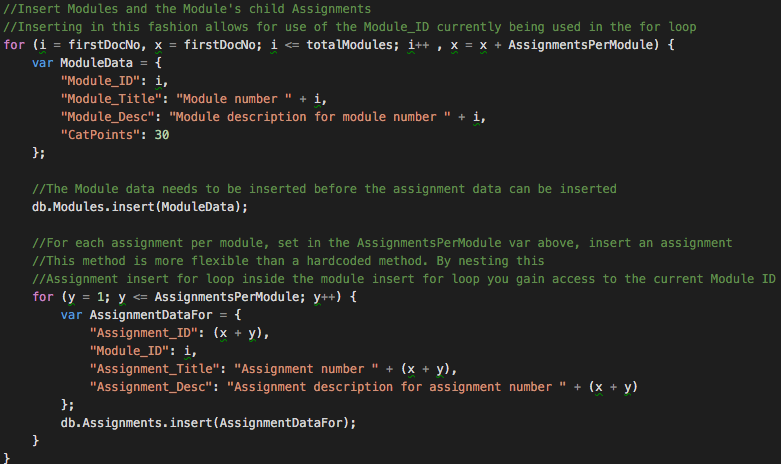


Fig. 4 Final assignment data insert code.

Fig. 4 shows this obstacle was overcome by moving the assignment data insert code into the for loop that inserts the Module data. Here for every module that is inserted a nested for loop also runs to insert a predefined number of assignments into the assignment table. Crucially, the assignment insert code has access to the module I.D. that has just been inserted, allowing for much greater control and the creation of realistic data.

To ensure the quality of the test data being inserted it was very important that the Grade field in the Module Grades table was an accurate representation of what a student achieved in that module’s child assignments. To understand how this was coded please see appendix one. Unfortunately the code in appendix one produced some unforeseen problems, when the relational scripts were run they were correctly inserting four million rows of data into the Module Grades table. Updating each of those rows with the calculated module mark takes a relatively long time, 26ms compared to a inserting a row, which only takes 1ms. This resulted in the relational data insert scripts taking far too long to run, therefore the decision was made to modify the code in appendix one to the code in appendix two.

In summary this code change inserts a new Module Grade table, doesn’t update the original Module Grade table and only uses the original table as a temporary table to understand which students are doing which modules. Despite inserting four million rows of extra data into a table a lot of time is saved over updating each of the original rows, up to 24ms per row which, in isolation might not seem like much but when millions of rows are considered it soon adds up. Further detail on the scale of this time saving can be found in Fig. 5.

Original format data insert:

Number of rows of data = 4,000,000

Time to insert 1 column of data = 1ms

Time to insert all three columns = 3 x 1ms

Time to update module grades = 26 ms

Total time to store a module grade for a student for a module = (3 + 26) = 29

Total time to store all module grades = 29 x 4,000,000 = 116 x106 ms

New format data insert:

Number of rows of data = 4,000,000

Time to insert 2 columns into temporary table = 2 ms

Time to insert 3 columns into Module Grades table = 3 ms

Total time to store a module grade for a student for a module = (2 + 3 ) = 5

Total time to store all module grades = 5x 4,000,000 = 20 x106 ms

Percentage difference between methods:

100 x (116 x106 /20 x106 ) = 580%

Therefore the new method of inserting data is just under six times quicker and can save up to 24ms per row of data.

Fig. 5 Calculations proving increased efficiency despite having an extra table

## Comparison of Data Structures

One million students worth of data was inserted into both non-relational and relational data structures, this was done twice per data structure, once without an index and once with an index. Care was taken to ensure exactly the same data was being stored in both structures, this is crucial because if one of the find statements has to return less data it is going to take less time, skewing the results. The time taken to carry out set tasks per structure and per insert script were then recorded, the results are shown in Fig. 6. Note the insert of relational data with no index timed out with a million students worth of data so timings for this are not available, however all four insert scripts were run to insert 50,000 students worth of data and these results can be seen in Fig. 7 proving the code in the script is fit for purpose.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Non-Relational no index (ms)** | **Non-Relational with index (ms)** | **Relational no index (ms)** | **Relational with index (ms)** |
| **Insert at beginning** | N/A | 2 | N/A | 5 |
| **Insert at middle** | N/A | 1 | N/A | 5 |
| **Insert at end** | 1 | 1 | Full insert failed | 4 |
| **Update at beginning** | 2 | 1 | Full insert failed | 1 |
| **Update at middle** | 859 | 1 | Full insert failed | 1 |
| **Update at end** | 1095 | 1 | Full insert failed | 1 |
| **Find at beginning** | 455 | 2 | Full insert failed | 5 |
| **Find at middle** | 378 | 4 | Full insert failed | 6 |
| **Find at end** | 377 | 3 | Full insert failed | 5 |
| **Delete at beginning** | 387 | 1 | Full insert failed | 1 |
| **Delete at middle** | 388 | 1 | Full insert failed | 1 |
| **Delete at end** | 387 | 1 | Full insert failed | 1 |

Fig. 6 Table of results for a 1 million students worth of data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Non-Relational no index (ms)** | **Non-Relational with index (ms)** | **Relational no index (ms)** | **Relational with index (ms)** |
| **Insert at beginning** | N/A | 1 | N/A | 5 |
| **Insert at middle** | N/A | 1 | N/A | 4 |
| **Insert at end** | 1 | 1 | 54 | 4 |
| **Update at beginning** | 2 | 11 | 0 | 1 |
| **Update at middle** | 46 | 1 | 12 | 1 |
| **Update at end** | 59 | 1 | 18 | 1 |
| **Find at beginning** | 1 | 1 | 1035 | 5 |
| **Find at middle** | 9 | 1 | 996 | 5 |
| **Find at end** | 22 | 1 | 992 | 6 |
| **Delete at beginning** | 18 | 1 | 180 | 1 |
| **Delete at middle** | 18 | 2 | 180 | 1 |
| **Delete at end** | 19 | 1 | 184 | 1 |

Fig. 7 Table of results for 50,000 students worth of data.

These tests were repeated to confirm the credibility of the results and these results are accurate. The results show that a non-relational data structure with an index performs the best in almost all tests, however the relational data structure with an index isn’t far behind. Based on these facts alone ensuring that any large volume of data is indexed is more important than choice of data structure. However if using an index is not an option then a non-relational data structure is considerably more efficient, particularly for read operations.

The system that stores student marks will be read intensive because a University will have thousands of students reading their marks and much fewer tutors writing grades into the system. A denormalised data structure is much more efficient for read intensive systems because there is no need to create time expensive joins to other tables (Msdn.microsoft.com, 2015). Denormalised databases are more scalable because they can be scaled horizontally by essentially spreading the database over multiple computers, whereas traditional database can only be scale vertically by increasing the capacity of a single computer (Ganesh Chandra, 2015). Therefore the non-relational data structure will be the most suitable and future proof solution to use for storing student marks.

# Dimensional database

This section of the report will explain the choices made when building and querying a dimensional database in Oracle Apex that stores student admissions data. Dimensional databases contain data that has been taken from a business’ operational data and processed into a format that helps make business decisions. The first step taken was to identify who might be making business decisions based on student admission data, these people are:

* A Principle,
* Heads of particular schools,
* Finance department.

The second step was to identify what questions these decision makers might want answered, these are:

|  |  |
| --- | --- |
| Number | Question |
| 1. | How many students on a course? |
| 2. | How many students on a course for a specific enrollment year? |
| 3. | How many students passed a course in a specific enrollment year? |
| 4. | What percentage of students passed a course in a specific enrollment year? |
| 5. | Of those students who passed students passed a course in a specific enrollment year how many are female? |
| 6. | Of those students who passed students passed a course in a specific enrollment year how many are Male? |
| 7. | What percentage of students passed a course in a specific enrollment year were female? |
| 8. | What percentage of students passed a course in a specific enrollment year were male? |
| 9. | How many students from all courses passed in a given year |
| 10. | What percentage of students from all courses passed for a given year? |
| 11. | What percentage of students from all courses passed for all years? |
| 12. | How many students come from each country? |
| 13. | How many students on a course from a country |
| 14. | How many students per campus? |
| 15. | How many Female students per campus? |

Fig. 8 Questions the decision makers would want answered.

From this dimension and fact tables of the database can be identified to produce the following database design:

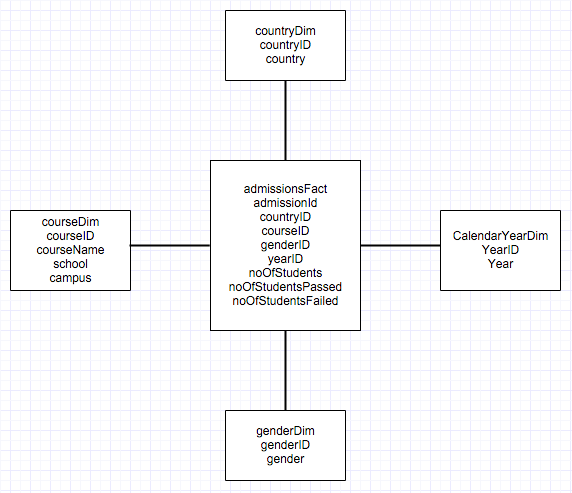


Fig. 9 Dimensional database model.

Important things to note from this design are:

* There is no student dimension, this is because decision-making data is an overview and an individual student’s information is too much detail (Kimball Group, 2009).
* As part of processing the data from the operational data three measures have been created in the admissionsFact table, noOfStudents, noOfStudentsPassed and noOfStudentsFailed.
* The courseDim table has course information but also includes school and campus columns, this is because there are many to one relationships linking the 3 objects, it is therefore appropriate to include all three columns in one dimension table (Kimball Group, 2009). The lowest level of granularity, course, has it’s I.D. added to the fact table so data can be queried on rising levels of granularity if needed.
* Months and days were not added to the CalendarYearDim table because none of the decision maker questions went into any more detail than a year, therefore year is the lowest level of granularity needed (Kimball Group, 2009).

This table structure was created in Oracle Apex and suitable test data inserted so that queries to answer the decision maker questions could be written, these can be found in appendix three. It is worth noting that queries 12 – 15 make use of the varying levels of granularity within the courseDim table by aggregating the results by country and campus and that query 12 contains a right outer join.

# Analysis

All three structures of database discussed in this report have their advantages:

* Fully normalised relational databases remove redundancy, are therefore very memory efficient and are excellent in write intensive systems, like those found in operational environments (MongoDB, 2015).
* Denormalised non-relational databases sacrifice memory efficiency to increase the speed at which data can be read; because data is stored in one table costly table joins do not have to be formed (Ganesh Chandra, 2015). This combined with horizontal scalability means that these types of databases are easier to scale than relational databases, making them excellent for web environments (MongoDB, 2015).
* Dimensional databases store data that has been transformed from an operational environment into a structure that is perfect for analytical environments. This structure uses somewhat denormalised data and once again cuts down on time expensive joins between tables to speed up querying (Kimball Group, 2009).
* For read performance it is very important to include an index in both normalised and denormalised data.

# Report limitations

Some limitations to this report include:

|  |  |
| --- | --- |
| Number | Limitation |
| 1. | All databases designed are quite simple, adding more complexity would help highlight the areas in which the different database structures exceled. |
| 2. | More test data, having a greater variety of test data would produce more realistic results. |
| 3. | Consulting with university decision makers would have produced more realistic and complex questions to answer, giving a greater understanding of what a dimensional database would have to provide. |

# References

Ganesh Chandra, D 2015, 'BASE analysis of NoSQL database', Future Generation Computer Systems, 52, Special Section: Cloud Computing: Security, Privacy and Practice, pp. 13-21, ScienceDirect.

Kimball Group,. 'The 10 Essential Rules Of Dimensional Modeling - Kimball Group'. N.p., 2009. Web. 12 Dec. 2015.

MongoDB,. "Mongodb And Mysql Compared". N.p., 2015. Web. 12 Dec. 2015.

Mongodb.com,. "Nosql Vs Relational Databases | Mongodb". N.p., 2014. Web. 12 Dec. 2015.

Msdn.microsoft.com,. 'Lesson 3: Optimizing The Database Design By Denormalizing'. N.p., 2015. Web. 12 Dec. 2015.

# Appendix One

## MongoDB code improvements

To calculate a module mark based on assignment marks the following strategy was developed where the Module Grades table was created using the code below:

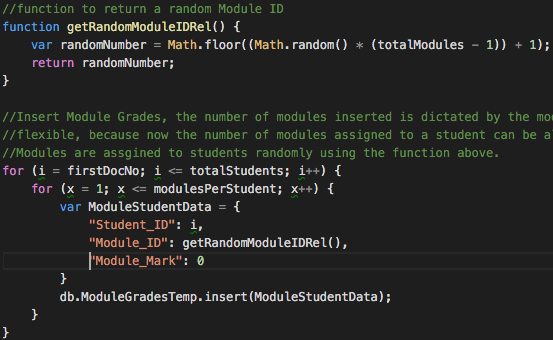


Fig. 10 Module Grades data insert code.

Once again this is a nested for loop statement, this is necessary to insert a predefined number of module results for one student and this is used to establish which modules a student is taking. Notice that the module mark is set to 0 at this point, this is so the column is created in the table and 0 is assigned as the module mark because the actual module mark has not yet been calculated. Now the relationship between students and modules has been established it can also be established which assignments that student will have to complete, so now the Assignment Results table can be created and have data inserted into it. The mark each student receives for an assignment is calculated randomly and an extra column was added to the table to contain a dynamic percentage value. This means that the parent module’s mark can be calculated by adding together each child assignment’s mark after it has been adjusted to represent the percentage that it is worth. An update statement was subsequently run on the Module Grades table to update the 0 value in the module mark column with this calculated mark. Due to the repetitive nature of this work it was wrapped in a function, please see Fig. 12 for this code.

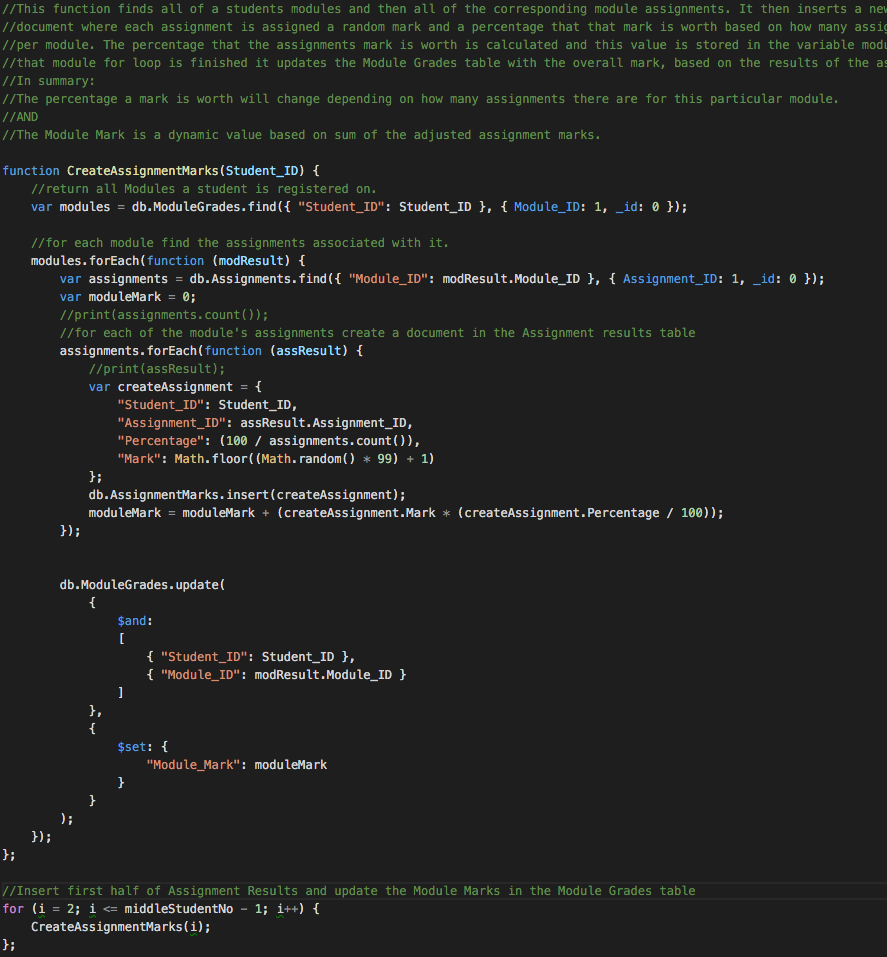
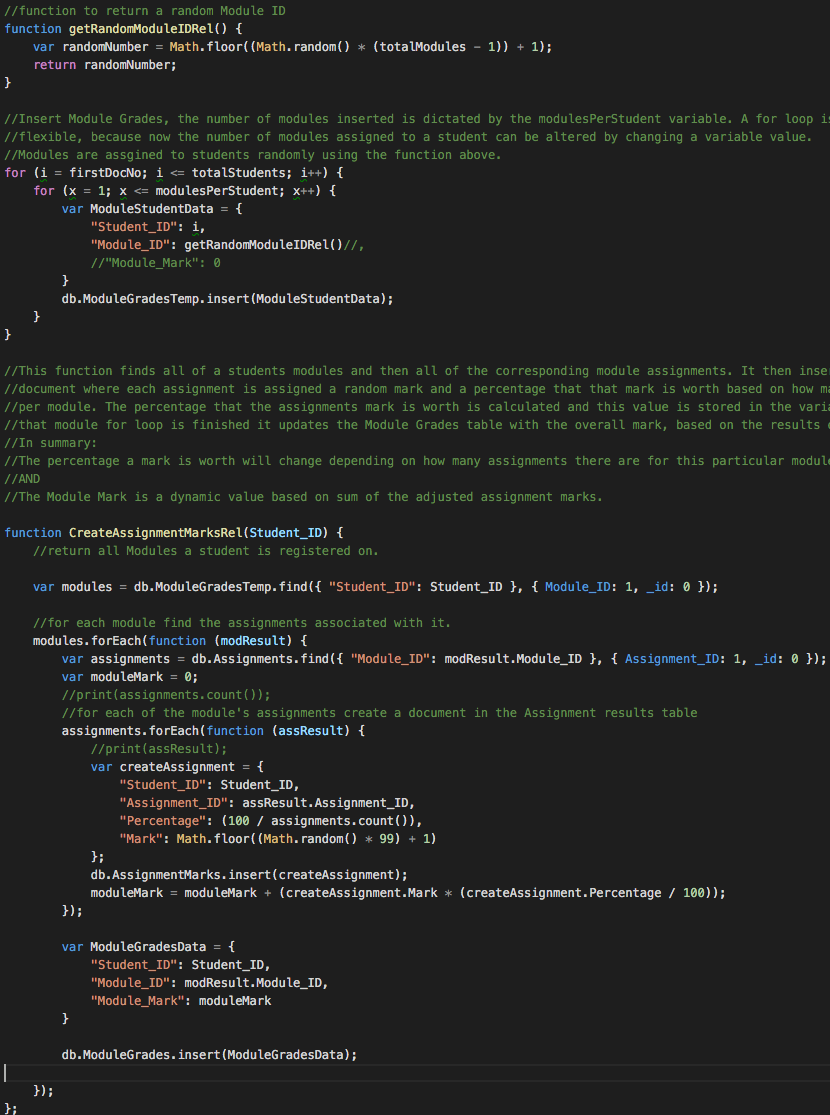
cb

Fig. 11 JavaScript function to insert assignment results data and calculate a module mark.

# Appendix Two

## Further MongoDB code improvements

No changes were made to the Assignment Results insert code in Fig. 12, all changes related to the Module Grades procedure. First the original Module Grades insert code was edited to not include the module mark column, meaning the table was only being used to establish the relationship between a student and a module and subsequently the relationship between a student and an assignment. This table was also renamed and should be considered temporary. The second change was to create a whole new table to hold the Module Grades data at the point in the code where the update statement used to be. This removes the update statement from the code altogether and results in a time saving of 24ms per row.

Fig. 12 Edited JavaScript for inserting assignment results data and calculating a module mark.

# Appendix Three

## SQL code to answer Decision maker’s questions

|  |  |
| --- | --- |
| Question  Number | SQL |
| 1. | select sum(noOfStudents) as "Number of Students"  from yr3admissionsfact fact  where fact.courseID = 3; |
| 2. | select sum(noOfStudents) as "Number of Students"  from yr3admissionsfact fact  where fact.courseID = 3  and fact.dateID = 14; |
| 3. | select sum(noOfStudentsPassed) as "Number of Students Who Passed"  from yr3admissionsfact fact  where fact.courseID = 3  and fact.dateID = 14; |
| 4. | select round((sum(noOfStudentsPassed) / (sum(noOfStudents))) \* 100) || '%' as "% of Students Who Passed"  from yr3admissionsfact fact  where fact.courseID = 3  and fact.dateID = 14; |
| 5. | select sum(noOfStudentsPassed) as "Number of Students Who Passed"  from yr3admissionsfact fact  where fact.courseID = 3  and fact.dateID = 14  and fact.genderID = 1; |
| 6. | select sum(noOfStudentsPassed) as "Number of Students Who Passed"  from yr3admissionsfact fact  where fact.courseID = 3  and fact.dateID = 14  and fact.genderID = 2; |
| 7. | select round((sum(noOfStudentsPassed) / (sum(noOfStudents))) \* 100) || '%' as "% of Female Students Passed"  from yr3admissionsfact fact  where fact.courseID = 3  and fact.dateID = 14  and fact.genderID = 1; |
| 8. | select round((sum(noOfStudentsPassed) / (sum(noOfStudents))) \* 100) || '%' as "% of Male Students Who Passed"  from yr3admissionsfact fact  where fact.courseID = 3  and fact.dateID = 14  and fact.genderID = 2; |
| 9. | select sum(noOfStudentsPassed) as "Number of Students Who Passed"  from yr3admissionsfact fact  where fact.dateID = 14; |
| 10. | select round((sum(noOfStudentsPassed) / (sum(noOfStudents))) \* 100) || '%' as "% of Students Who Passed"  from yr3admissionsfact fact  where fact.dateID = 14; |
| 11. | select round((sum(noOfStudentsPassed) / (sum(noOfStudents))) \* 100) || '%' as "% of Students Who Passed",  dim.calendarYear as "Calendar Year"  from yr3admissionsfact fact  inner join yr3CalendarYearDim dim on dim.dateid = fact.dateid  group by dim.calendaryear  order by dim.calendaryear; |
| 12. | select sum(noOfStudents) as "Number of Students",  dim.countryName as "Country"  from yr3admissionsfact fact  right outer join yr3countrydim dim on dim.countryId = fact.countryID  group by dim.countryName  order by dim.countryName; |
| 13. | select sum(noOfStudents) as "Number of Students"  from yr3admissionsfact fact  where fact.countryID = 5  and fact.courseID = 1; |
| 14. | select sum(noOfStudents) as "Number of Students",  dim.campus as "Campus"  from yr3admissionsfact fact  inner join yr3CourseDim dim on dim.courseID = fact.courseID  group by dim.campus  order by dim.campus; |
| 15. | select sum(noOfStudents) as "Number of Students",  dim.campus as "Campus"  from yr3admissionsfact fact  inner join yr3CourseDim dim on dim.courseID = fact.courseID  where fact.genderID = 1  group by dim.campus  order by dim.campus; |