University of Cape Town

CSC5007Z

Databases for Data Scientists

Databases Group Assignment 2

Author:
T Woodley, J Harris

Student Number:
WDLTIF001, HRRJES004

May 10, 2019

Contents

1 Question 1

In this question we create a relation from the database designed in Figure 1. We then use python to connect to it and change/ update it.

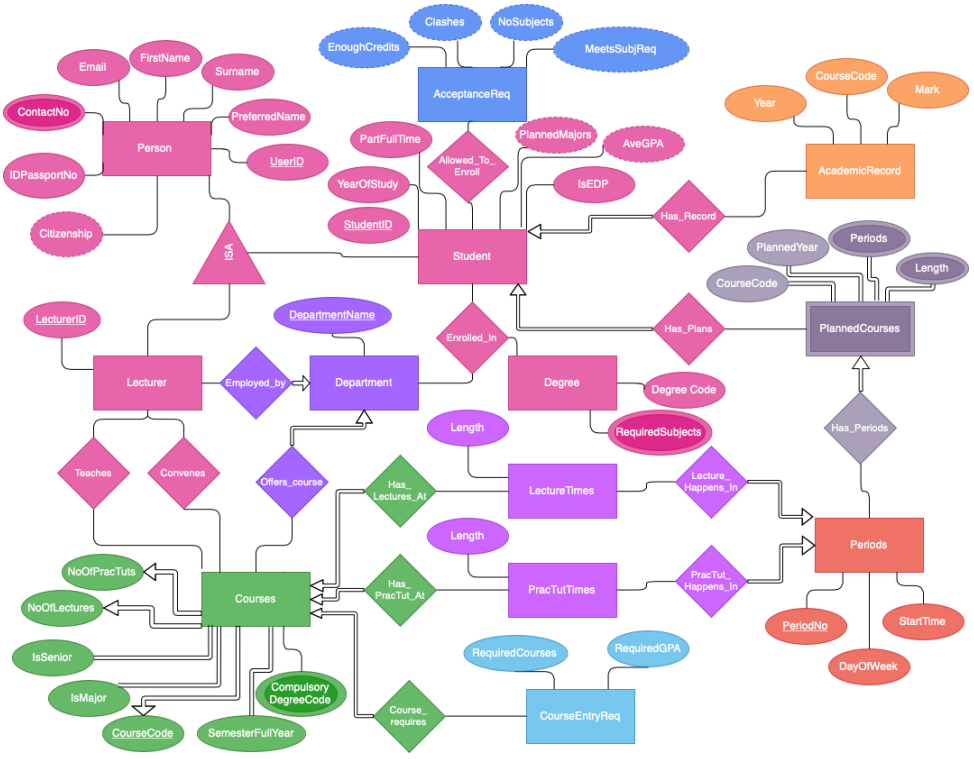

Figure 1: Relational database design for UCT admissions app.

1.1 Creating MySQL relation

The code snippet below shows how the table was created in mySQL

```
CREATE SCHEMA ‘Assignment2’ ;

CREATE TABLE ‘Assignment2’.‘Courses’ (
  ‘CourseCode’ INT NOT NULL,
  ‘NoOfPracTuts’ INT NULL,
  ‘NoOfLectures’ INT NULL,
  ‘IsSenior’ VARCHAR(45) NULL,
  ‘IsMajor’ VARCHAR(45) NULL,
  ‘SemesterFullYear’ VARCHAR(45) NULL,
```

```
  'Credits' VARCHAR(45) NULL,
  PRIMARY KEY ('CourseCode'));
```

1.2 Program to implement mySQL queries - Using Python

Python was then used to connect to the database and alter the database, the three alterations done as indicated by *sqlone*, *sqltwo* and *sqlthree* were:

1. Inserting a tuple into the table
2. Updating the *IsMajor* Column for the inserted row
3. Returning the updated row

```
import mysql.connector
db = mysql.connector.connect(user='root', password='Password'
   ,host='127.0.0.1',database='Assignment2',port = 3306)

cursor = db.cursor()

sqlone ="INSERT INTO Courses (CourseCode, NoOfPracTuts,
   NoOfLectures, IsSenior, IsMajor, SemesterFullYear, Credits)
    VALUES ('1236','2','2','Y','Y', 'S','26')"
sqltwo = 'UPDATE Courses SET IsMajor = "N" WHERE CourseCode =
    1236'
sqlthree = 'SELECT * FROM Courses WHERE CourseCode = 1236'

try:
        cursor.execute(sqlone)
except Exception as e:
        db.rollback()
        print(e)

db.commit()

try:
        cursor.execute(sqltwo)
except Exception as e:
        db.rollback()
        print(e)

db.commit()
```

```
try:
        cursor.execute(sqlthree)
        updated_line=cursor.fetchall()
except Exception as e:
        db.rollback()
        print(e)


print(updated_line)
```

2 Design in different NoSQL databases

NoSQL data bases are designs with the applications in mind. In this context, an online registration system is planned for UCT BSc students. They will be asked for their student number and which majors they plan to have, as well as which courses they plan to take this year (and also in future years of their BSc if applicable). The main applications of the system will be to:

1. Show the student's results for all courses they have already taken in the past (if any).
2. Point out any problems with the student's plan in terms of
   (a) timetable clashes
   (b) missing pre-requisites
   (c) too many courses
   (d) insufficient credits to graduate
   (e) lacking a compulsory course
3. Give them an indication of the academic performance of students in similar positions in past years —in this way they can hopefully increase the likelihood of achieving their academic goal.

2.1 Key-Value Store

Information about specific courses can be found by searching for a key that begins with `Courses` (Figure 2) followed by the `CourseCode` . General information was stored separately so that it can be accessed faster for queries such as: "how many credits does this course have?" or "who is the lecturer for this specific course?".

The key that refers to the schedule is in a json format with `NoOfTuts`, `NoOfLectures` and `FulltimeSemester`. These were grouped together as a user would want this information at the same time. A json format was used for ease of searching within the returned structure.

The key that refers to pre-requisites contains an array of course codes which are required for the specific course, this will allow the application to access what the pre-requisites needed are for specific courses a student has registered for quickly and efficiently.

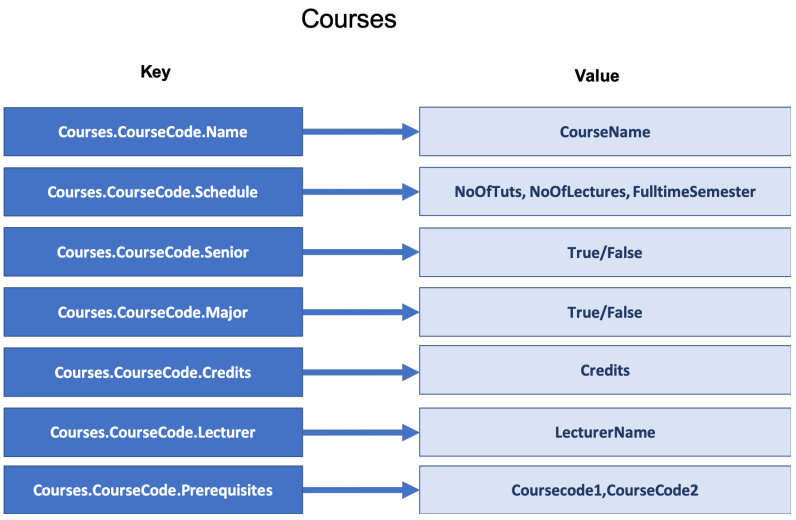

Figure 2: `Courses` components of the key-value store DB for UCT admissions app.

Information about students can be found by searching for a key that begins with `Students` followed by the student's `StudentID` (Figure 3). Information needed to be accessed quickly to help work out if requirements are met such as degree, year of study and if the student is on the EDP are stored as single values.

The key that refers to the `Name` is in a json format with Name, Surname and preferred name. These were grouped together as a user would typically want to access this information at the same time. A json format was used for ease of searching within the returned structure. Contact information was also grouped together; since this type of database allows for different types of inputs into the value column, multiple emails or contact numbers can be added.

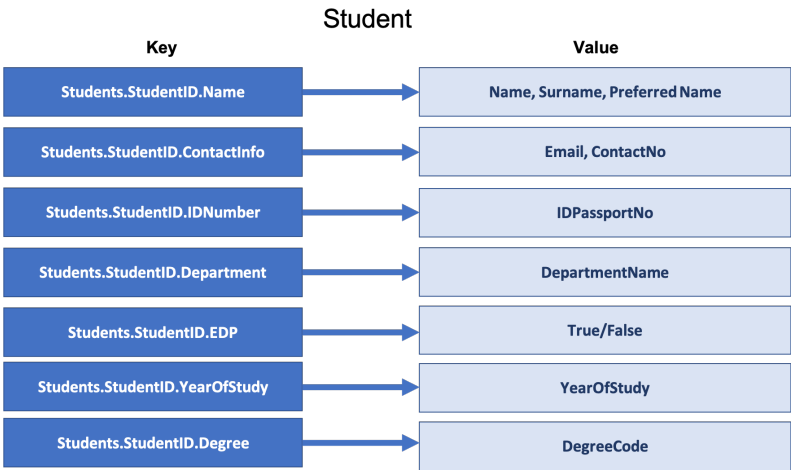

Figure 3: `Student` components of the key-value store DB for UCT admissions app.

Academic record stores the mark that a student got for a specific course in a specific year (Figure 4). A student's academic record can be accessed by searching for all primary keys that begin with `AcademicRecord` and contain the given `StudentID`.

If you search within the same collection of keys for keys that contain a `CourseCode`, you can see all the results achieved by all students who have taken that course in the past.

The planned courses a student takes can be found by searching for primary keys that start with `PlannedCourses` and contain the specified `StudentID` (Figure 4). This can also be searched for by year. The courses are then ordered by a number at the end of the primary key.

These keys will allow for the application to make queries to acquire information needed to work out if a student has registered for enough/too many courses. With the combined use of course information stored in `Courses.CourseCode` keys the application can calculate if the student has enough credits to graduate and if they are missing any compulsory courses.

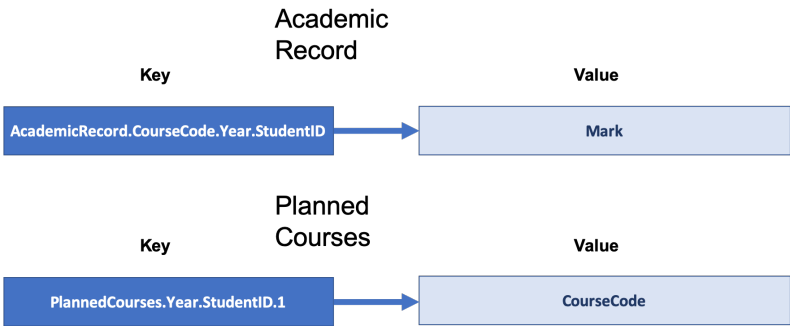

Figure 4: `AcademicRecord` and `PlannedCourses` components of the key-value store DB for UCT admissions app.

Each `CourseCode` will have a number of keys referring to different periods in which the course's lectures occur (Figure 5). The same format was used for tutorials and practicals (Figure 5).

This will allow the application to retrieve information needed to work out if a student will have any timetable clashes between subjects, so that they can resolve the conflict in their schedule.

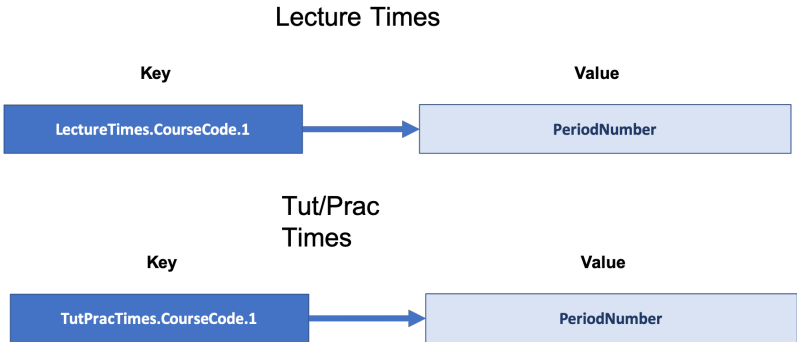

Figure 5: `LecctureTimes` and `TutPracTimes` components of the key-value store DB for UCT admissions app.

2.2 Document stores

The document store DB shown in Figure 6 is designed to allow for easy access to relevant data based on the main purposes of the UCT admissions app. All data pertaining to the student is stored in a document in the `Student` collection . All courses that a student has taken are nested within the student's document, together with the year they completed the course and the mark they achieved. This allows for their academic record to be readily generated.

Determining whether there are any problems with the student's plan can be done by considering the `PlannedCourses` array in the `Students` document. Here, the `Periods` and `Year` fields can be used to determine if there are any timetable clashes. The `Credits` can be summed across all courses in both the `AcademicRecord` and `PlannedCourses` (in the `Student` document) to determine if the student's subject choices will have a cumulative total of credits that makes them eligible for graduation at the end of their studies. The number of 1's in the `IsMajor` and `IsSenior` fields can be summed to ensure that the student's subject choices meet the degree requirements for the number of Senior and Major courses required for graduation. It can be determined whether the prerequisite courses (`Prereq`) for a student's `PlannedCourses` has been satisfied by the courses they have already completed, as shown in their `AcademicRecord`.

The historic record of how students performed in a given course can be determined by considering the relevant document in the `Course` collection, which contains a record of every student who has taken the course, the year they took it, and the mark they achieved.

The information in each document was grouped so that the data required to address each of the above concerns would be available within a single document.

Lastly, information about the lecturers and the sections they teach in which courses, and which courses they convene in stored in a document in the `Lecturer` collection. This is so that a student can contact a course convener in the event that they have a specific plea to make with respect to prerequisite courses (e.g. in the case of transferring credits from another university), for example.

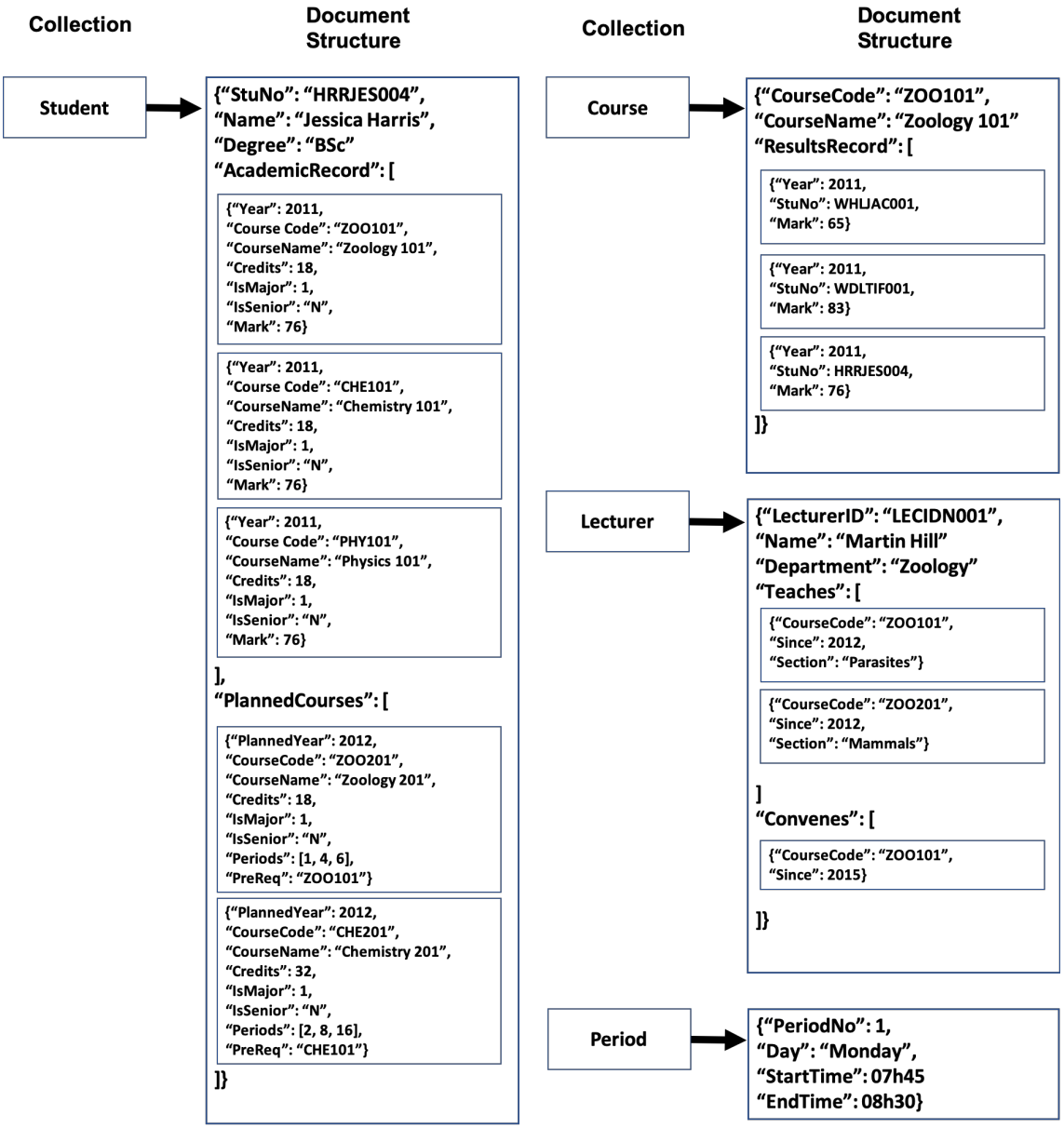

Figure 6: Document Store DB design for UCT admissions app.

2.2.1 Alternative Approach

Although having large amounts of data stored in each document makes grouping related information easier, it does create limitations when one wants to perform more specific query. For example, the previous design stores all the results of students

who took the course under `ResultsRecord` in documents in the `Courses` collection. This does not allow for a specific mark for a given student to be returned, as all the students and their marks will be returned. The larger return will then have to be sorted through to find the specific student's mark.

The software developers could prefer to do more specific indexing when pulling from the database, resulting in more requests being sent to the database. Alternatively, they might rather want to do specific indexing in the application's code and perform fewer large requests from the database.

If the preference is for more specific queries the following design can be implemented (Figure 7). This is the design that was used for questions 4 and 5. In this case, a separate document is created for each element that was previously stored in arrays.

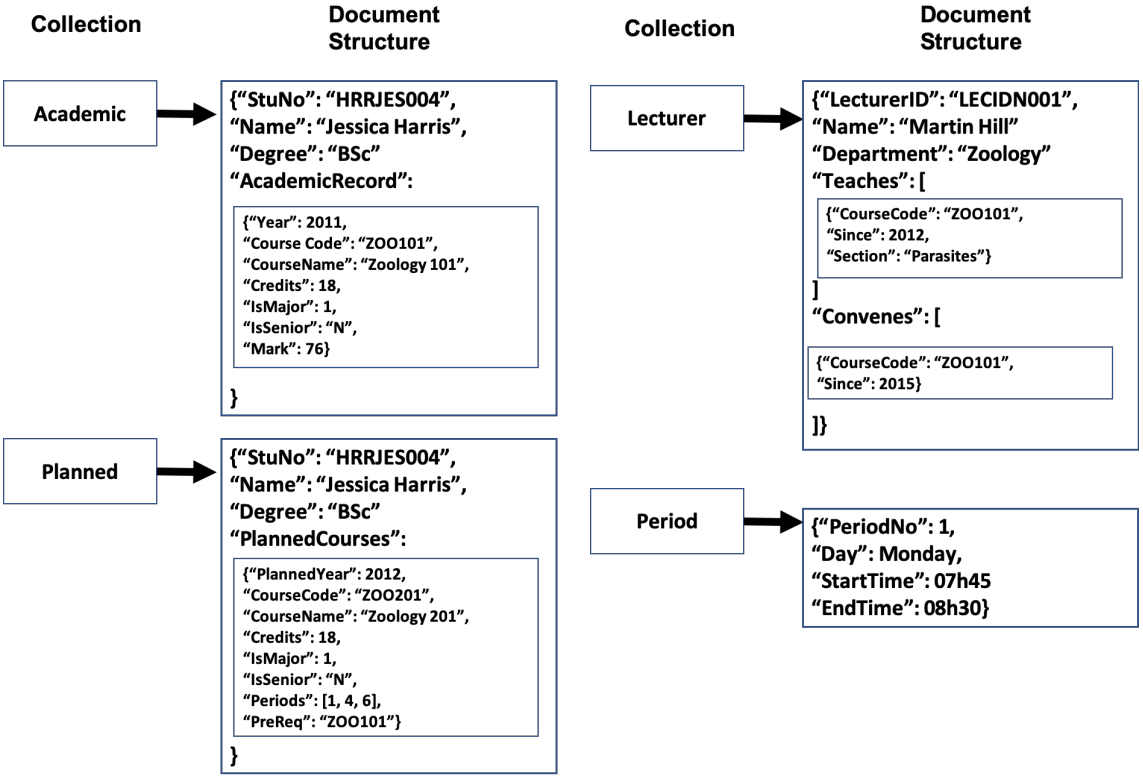

Figure 7: Document Store DB alternative design for UCT admissions app.

2.3 Graph data stores

The document store DB shown in Figure 8 contains nodes representing students, lecturers, departments, courses, and periods. These each have an ID code and several characteristics that are relevant for each type of entity. The directed edges between them indicate the relationships that they have with one another. These relationships have their own ID code, label, and characteristics.

One can determine a student's academic record by considering the edges that are labeled as `HasTaken`. This type of relationship is accompanied by the year in which the course was taken (allowing for multiple relationships to exist in the case of failure and re-attempt), as well as the mark that was achieved by that student in that year.

A student's timetable can be put together by considering all of the periods that their selected courses `HappensIn`. If a given period occurs more than once in a given semester, it indicates a timetable clash. Each course is represented by a node labeled by the `CourseName`. Characteristics of the course, including its `Credit` value and Boolean values describing whether it `IsMajor` or `IsSenior`, can be used to determine if a student's subject choices meet the requirements for the degree. Each course is linked to its prerequisite courses by an edge with ID and label `HasPrerequisite`.

To find the marks of each student who has ever completed a given course, one must trace back from the course of interest to which students that course `WasTakenBy`. This involves tracing a large number of edges.

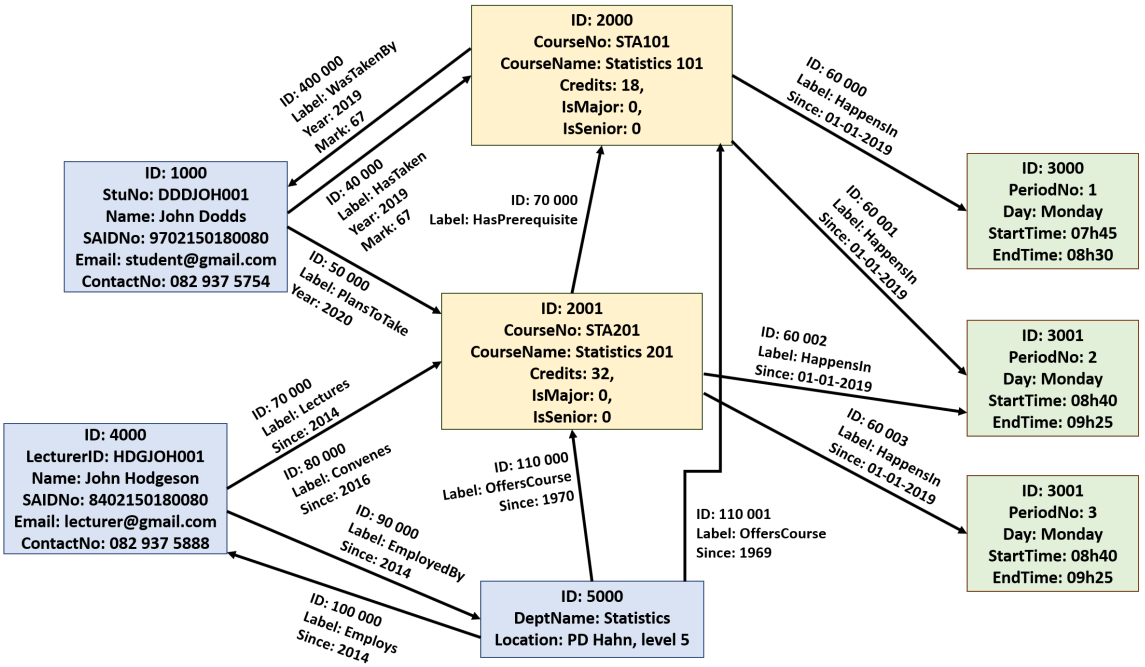

Figure 8: Graph Store DB design for UCT admissions app.

2.4 Column-family stores

The column-family store DB shown in Figure 9 shows multiple columnkeys that map to a column family containing related information that is usually accessed together.

Background information for each `Student` entity is grouped by `StudentNo` into `_Name`, `_PersonalData` and `_ContactData`. The `StudentNo_AcademicRecord` columnkey can be used to access all of the courses a student has taken in the past, together with the year they took it and the mark they obtained. The `StudentNo_PlannedCourses` columnkey can be used to access all of the courses a student intends to take throughout the remainder of their studies, together with the year they plan to do so. The other columns that are returned by this columnkey are `LecturePeriod`, `PracTutPeriods`, which allows for easy identification of possible clashes, and `Prereq`, which indicates which prerequisite course is required for the planned course.

To determine whether a student's academic plan contains enough credits to qualify for graduation at the end of their studies, one can consider the column family labeled `StudentNo_CheckCourseCredits`. To simplify the query of the total credits, there is replication of several entries from `StudentNo_AcademicRecord` and `StudentNo_PlannedCourses`. The objective was to place both credits that have al-

ready been completed and credits that are planned to be taken in the same column to rapidly answer the above question.

To easily access the historic record of student performance in a given course, we have the `CourseCode_StudentPerformance` columnkey that returns the student number of every student who has ever enrolled in the course, the year they did so, and the mark they achieved.

Evidently, the column-families were chosen so as to simplify the queries needed to access the information required by the UCT admissions app.

StudentNo_Name

FirstName	Surname
John	Doe

StudentNo_PersonalData

SAIDNo	Citizenship
9211110180075	SA

StudentNo_ContactData

Email	ContactNo
st@email.com	082 643 8484

StudentNo _CheckCourseCredits

Year	CourseCode	Credits
2011	CHE101	18
2011	MAT1C1	18
2011	CEL101	18
2011	PHY101	18
2012	CHE201	20
2012	PHY2	36
2013	CHE3	36

StudentNo_AcademicRecord

CourseCode	Year	Mark
CHE101	2013	81
MAT1C1	2013	60
CEL101	2013	72
PHY101	2013	71

StudentNo_PlannedCourses

CourseCode	Year	LecturePeriod	PracTutPeriod	PreReq
CHE201	2014	3, 4, 10	15, 16	CHE101
PHY2	2014	5, 6, 14	20, 21	PHY101
CHE3	2015	15, 16	25, 26	CHE201

Periods

PeriodNo	DayOfWeek	StartTime	EndTime
1	Monday	07h45	08h30
2	Monday	08h40	09h25
3	Monday	09h35	10h20

LecturerID_ConveningData

Name	CourseCode	Email
Prof. Hastings	PHY101	hasting@uct.ac.za

CourseCode_StudentPerformance

StudentNo	Year	Mark
HRRJES004	2013	81
WDLTIF001	2013	60
WHLJAC001	2013	72
ARCGUY007	2013	71

Figure 9: Column-family Store DB design for UCT admissions app.

3 Different NoSQL Advantages and Disadvantages

3.1 Key-value stores

Advantages of using key-value stores for the UCT admissions app:

- Key-value stores have the best performance in terms of lookup speed.
- As they do not store any placeholders for optional values, as in most relational databases, key-value databases can handle incomplete data and often require less storage memory.
- As data is partitioned and replicated across a cluster to get scalability and availability, key value stores are highly effective at scaling applications that deal with high-velocity, non-transactional data [1].

Disadvantages of using key-value stores for the UCT admissions app:

- Queries can only be done via the primary key, limiting the number of attributes queries can be performed on.
- Because there is no schemas in the database, it's more difficult to gain an overview of the real-world entities and their attributes.

3.2 Document stores

Advantages of using document stores for the UCT admissions app:

- They are able to handle incomplete data, which may become useful if additional unstructured information is added to the database at a later stage.
- Document databases are essentially key-value stores where the value is a document; they extend key-value databases in that the aggregates have named fields. These allow us to fetch the parts of entities that are relevant for a given query, and make field-based indexing and querying possible.

Disadvantages of using document stores for the UCT admissions app:

- Although they have named fields, these cannot be used to improve physical storage on nodes and their disks.
- Foreign key constraints are not enforced; the programmer has to make sure that the IDs in relationships are meaningful i.e. represent the right kind of

entity.

- No standard query syntax which would make it harder to integrate into the application.
- Slow query performance which could impact the application's speed.

3.3 Graph data stores

Advantages of using graph stores for the UCT admissions app:

- They are able to identify the shortest path between entities, which could help identify students who have taken similar courses. This could help us refine the database to work out expected performance over a given course set.
- They are able to convey connectedness.

Disadvantages of using graph stores for the UCT admissions app:

- Graph databases are not a good choice for performance analytics as they do not do aggregations particularly well [2]. This weakness of graph store DBs will impact the UCT registration app in terms of its ability to aggregate past students' performance to give an overview of how students have coped with a given course in the past. To find the marks of each student who has ever completed a given course, one must trace back from the course of interest to which students that course `WasTakenBy`. This involves tracing an ever-increasingly large number of edges.
- They are hard to use on a cluster. This may impact the performance of the UCT admissions app, particularly when the volume of usersis particularly high at the beginning of the academic year.

3.4 Column-family stores

Advantages of using column-family stores for the UCT admissions app:

- Column-oriented storage is very efficient for processing columns because the column values being fetched will be stored together on the disk, thereby minimizing disk access. This makes column stores good for analytics purposes, which will be an advantage when extracting information about the performance of past students in a given course.

- They have fast lookup times and good distribution. This will enhance the user experience of the UCT admissions app, as responses will be quick and many users will be able to access the data at the same time, which is an essential consideration especially at the beginning of the academic year, when high volumes of users are expected.

Disadvantages of using column-family stores for the UCT admissions app:

- They have a limited query syntax.
- They have very low-level application programming interface (API) which could lead to having to write more complex code.

4 MongoDB database creation and queries

4.1 Inserting data into database

Creating a collection to hold information about student's academic records with information about the courses they have taken.

4.1.1 Statement Run

```
db.Academic.insertMany([{Studno:"HRRJES004",
                         Name:"Jessica Harris",
                         Courseinfo:{Year:2011,
                         Coursecode:"ZOO101", Coursename:"Zoology 101",
                         Mark:76,
                         Credits:24,
                         Major:true,
                         Senior:true}},
                         {Studno:"HRRJES004",
                         Name:"Jessica Harris",
                         Courseinfo:{Year:2011,
                         Coursecode:"CHM101",
                         Coursename:"Chemisty 101",
                         Mark:66,
                         Credits:12,
                         Major:false,
                         Senior:false}},
                         {Studno:"HRRJES004",
                         Name:"Jessica Harris",
                         Courseinfo:{Year:2011,
                         Coursecode:"MTH101",
                         Coursename:"MATHS 101",
                         Mark:86,
                         Credits:30,
                         Major:true,
                         Senior:false}},
                         {Studno:"WDLTIF001",
                         Name:"Tiffany Woodley",
                         Courseinfo:{Year:2011,
                         Coursecode:"ZOO101", Coursename:"Zoology 101",
```

```
        Mark:62,
        Credits:24,
        Major:true,
        Senior:true}},
        {Studno:"WDLTIF001",
        Name:"Tiffany Woodley",
        Courseinfo:{Year:2011,
        Coursecode:"CHM101",
        Coursename:"Chemisty 101",
        Mark:77,
        Credits:12,
        Major:false,
        Senior:false}},
        {Studno:"WDLTIF001",
        Name:"Tiffany Woodley",
        Courseinfo:{Year:2011,
        Coursecode:"MTH101", Coursename:"MATHS 101",
        Mark:68,
        Credits:30,
        Major:true,
        Senior:false}},
        {Studno:"DOEJON003",
        Name:"John Doe",
        Courseinfo:{Year:2011,
        Coursecode:"ZOO101",
        Coursename:"Zoology 101",
        Mark:65,
        Credits:24,
        Major:true,
        Senior:true}},
    {Studno:"DOEJON003",
        Name:"John Doe",
        Courseinfo:{Year:2011,
        Coursecode:"CHM101", Coursename:"Chemisty 101",
        Mark:70,
        Credits:12,
        Major:false,
        Senior:false}},
        {Studno:"DOEJON003",
        Name:"John Doe",
```

```
                    Courseinfo:{Year:2011,
                    Coursecode:"MTH101",
                    Coursename:"MATHS 101",
                Mark:80,
                    Credits:30,
                    Major:true,
                    Senior:false}}])
```

4.1.2 Output

```
{
"acknowledged" : true,
"insertedIds" : [
ObjectId("5cd14044799433c5d081c60d"),
ObjectId("5cd14044799433c5d081c60e"),
ObjectId("5cd14044799433c5d081c60f"),
ObjectId("5cd14044799433c5d081c610"),
ObjectId("5cd14044799433c5d081c611"),
ObjectId("5cd14044799433c5d081c612"),
ObjectId("5cd14044799433c5d081c613"),
ObjectId("5cd14044799433c5d081c614"),
ObjectId("5cd14044799433c5d081c615")
]
}
```

4.1.3 Check

The previous insert statement can be checked by finding the records inserted for each student and then the return statement compared to what was wanted to be inserted into the database. If the database isn't too big the entire database could be returned and checked.

The statements which can be used to check each student's academic history can be seen below.

```
db.Academic.find({"Studno": "HRRJES004"}).pretty()
db.Academic.find({"Studno": "WDLTIF001"}).pretty()
db.Academic.find({"Studno": "DOEJON003"}).pretty()
```

4.2 Deleting information from database

Turns out that John Doe didn't actually complete MATHS 101 and so this needs to be deleted from the database.

4.2.1 Statement Run

```
db.Academic.deleteOne({"Studno":"DOEJON003","Courseinfo.Coursecode":"MTH101"})
```

4.2.2 Output

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

4.2.3 Check

Check all of John Doe's Subjects to make sure there is no MATHS 101 in the records.

```
db.Academic.find({"Studno":"DOEJON003"}).pretty()
```

4.3 Updating the database

John Doe entered his student number incorrectly, his studentno is actually DOEJON002 and so needs to be updated across the database

4.3.1 Statement Run

```
db.Academic.updateMany({"Studno": "DOEJON003"}, {$set: {"Studno": "DOEJON002"}})
```

4.3.2 Output

```
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
```

4.3.3 Check

Returns all entries for John Doe and from this was able to check the his student number had been changed for all entries

```
db.Academic.find({"Name":"John Doe"}).pretty()
```

4.4 Simple query in database

This query is to check which students took the subject Zoology 101

4.4.1 Statement Run

```
db.Academic.find({"Courseinfo.Coursecode": "ZOO101"}).pretty()
```

4.4.2 Output


```
{
        "_id" : ObjectId("5cd14044799433c5d081c60d"),
        "Studno" : "HRRJES004",
        "Name" : "Jessica Harris",
        "Courseinfo" : {
                "Year" : 2011,
                "Coursecode" : "ZOO101",
                "Coursename" : "Zoology 101",
                "Mark" : 76,
                "Credits" : 24,
                "Major" : true,
                "Senior" : true
        }
}
{
        "_id" : ObjectId("5cd14044799433c5d081c610"),
        "Studno" : "WDLTIF001",
        "Name" : "Tiffany Woodley",
        "Courseinfo" : {
                "Year" : 2011,
                "Coursecode" : "ZOO101",
                "Coursename" : "Zoology 101",
```

```
                "Mark" : 62,
                "Credits" : 24,
                "Major" : true,
                "Senior" : true
        }
}
{
        "_id" : ObjectId("5cd14044799433c5d081c613"),
        "Studno" : "DOEJON002",
        "Name" : "John Doe",
        "Courseinfo" : {
                "Year" : 2011,
                "Coursecode" : "ZOO101",
                "Coursename" : "Zoology 101",
                "Mark" : 65,
                "Credits" : 24,
                "Major" : true,
                "Senior" : true
        }
}
```

4.4.3 Check

This can be checked by looking at all the entries in the collection and checking everyone in the collection who had taken Zoology 101 was in the return of the previous statement. The whole collection can be viewed with the following statement

```
db.Academic.find().pretty()
```

4.5 Advanced query in database

There is a Zoology Olympiad coming up and the lecturers want to see which students they should approach. They are considering all students who have taken either a Major or Senior subject in Zoology an received a mark greater than 65

4.5.1 Statement Run

```
db.Academic.find({
                "Courseinfo.Coursecode":{\$regex:/^ZOO/},
```

```
                "Courseinfo.Mark":{\$gt:65},
                \$or: [
                        {"Courseinfo.Major":true},
                        {"Courseinfo.Senior":true}
                                ]
                        }).pretty()
```

4.5.2 Output

```
{
        "_id" : ObjectId("5cd14044799433c5d081c60d"),
        "Studno" : "HRRJES004",
        "Name" : "Jessica Harris",
        "Courseinfo" : {
                "Year" : 2011,
                "Coursecode" : "ZOO101",
                "Coursename" : "Zoology 101",
                "Mark" : 76,
                "Credits" : 24,
                "Major" : true,
                "Senior" : true
        }
}
```

4.5.3 Check

This can be checked by looking at all the entries in the collection and checking everyone in the collection who got above 65 % for a zoology senior or major subject against what the previous statement returned. The whole collection can be viewed with the following statement

```
db.Academic.find().pretty()
```

5 Program to implement mongoDB statements - Using Python

5.1 Code

```
import pymongo

## connecting to database

myclient = pymongo.MongoClient("mongodb://localhost:27017/")

mydb = myclient["mydatabase"]

mycol = mydb["Academic"]

## adding documents to the collection

inserts =[{"Studno":"HRRJES004",
                    "Name":"Jessica Harris",
                        "Courseinfo":{"Year":2011,
                                                "Coursecode":
                                                   "ZOO101",
                                                "Coursename":
                                                   "Zoology 10
                                                   1",
                                                "Mark":76,
                                                "Credits":24,
                                                "Major":True,
                                                "Senior":True
                                                   }},
                                {"Studno":"HRRJES004",
                                "Name":"Jessica Harris",
                                "Courseinfo":{"Year":2011,
                                                "Coursecode":
                                                   "CHM101",
                                                "Coursename":
                                                   "Chemisty 1
                                                   01",
                                                "Mark":66,
                                                "Credits":12,
                                                "Major":False
                                                   ,
```

```
                                "Senior":
                                   False}},
{"Studno":"HRRJES004",
"Name":"Jessica Harris",
"Courseinfo":{"Year":2011,
                                "Coursecode":
                                   "MTH101",
                                "Coursename":
                                   "MATHS 101"
                                   ,
                                "Mark":86,
                                "Credits":30,
                                "Major":True,
                                "Senior":
                                   False}},
{"Studno":"WDLTIF001",
 "Name":"Tiffany Woodley",
 "Courseinfo":{"Year":2011,
                                "Coursecode":
                                   "ZOO101",
                                "Coursename":
                                   "Zoology 10
                                   1",
                                "Mark":62,
                                "Credits":24,
                                "Major":True,
                                "Senior":True
                                   }},
{"Studno":"WDLTIF001",
"Name":"Tiffany Woodley",
"Courseinfo":{"Year":2011,
                                "Coursecode":
                                   "CHM101",
                                "Coursename":
                                   "Chemisty 1
                                   01",
                                "Mark":77,
                                "Credits":12,
                                "Major":False
                                   ,
                                "Senior":
                                   False}},
{"Studno":"WDLTIF001",
```

```
"Name":"Tiffany Woodley",
"Courseinfo":{"Year":2011,
                          "Coursecode":
                             "MTH101",
                          "Coursename":
                             "MATHS 101"
                             ,
                          "Mark":68,
                          "Credits":30,
                          "Major":True,
                          "Senior":
                             False}},
{"Studno":"DOEJON003",
"Name":"John Doe",
"Courseinfo":{"Year":2011,
                          "Coursecode":
                             "ZOO101",
                          "Coursename":
                             "Zoology 10
                             1",
                          "Mark":65,
                          "Credits":24,
                          "Major":True,
                          "Senior":True
                             }},
{"Studno":"DOEJON003",
"Name":"John Doe",
"Courseinfo":{"Year":2011,
                          "Coursecode":
                             "CHM101",
                          "Coursename":
                             "Chemisty 1
                             01",
                          "Mark":70,
                          "Credits":12,
                          "Major":False
                             ,
                          "Senior":
                             False}},
{"Studno":"DOEJON003",
"Name":"John Doe",
"Courseinfo":{"Year":2011,
                          "Coursecode":
```

```
                                                    "MTH101",
                                                "Coursename":
                                                    "MATHS 101"
                                                    ,
                                                "Mark":80,
                                                "Credits":30,
                                                "Major":True,
                                                "Senior":
                                                    False}}]

result = mycol.insert_many(inserts)
print("IDs of inserted items:")
print(result.inserted_ids)

## delete statement

result = mycol.delete_many({"Studno":"DOEJON003","Courseinfo.
    Coursecode":"MTH101"})
print("Count of deleted items:")
print(result.deleted_count, " documents deleted.")

## insert statement

myquery = {"Studno": "DOEJON003"}
newvalues = {"$set": {"Studno": "DOEJON002"}}
result = mycol.update_many(myquery, newvalues)
print("Count of updated items:")
print(result.modified_count, "documents updated.")

## simple query

print("Simple Query Return:")
for x in mycol.find({"Courseinfo.Coursecode": "ZOO101"}):
  print(x)

## complex query

query =  { "Courseinfo.Coursecode":{"$regex":"^ZOO"},
                        "Courseinfo.Mark":{"$gt":65},
                        "$or": [
                                        {"Courseinfo.Major":
                                            True},
                                        {"Courseinfo.Senior":
```

```
                                        True}
                                    ]
                            }
print("Complex Query Return:")
for x in mycol.find(query):
  print(x)
```

5.2 Output

```
IDs of inserted items:
[ObjectId('5cd16b00da133e7fded3c232'), ObjectId('5cd16b00da133e7fded3c233'),
ObjectId('5cd16b00da133e7fded3c234'), ObjectId('5cd16b00da133e7fded3c235'),
ObjectId('5cd16b00da133e7fded3c236'), ObjectId('5cd16b00da133e7fded3c237'),
ObjectId('5cd16b00da133e7fded3c238'), ObjectId('5cd16b00da133e7fded3c239'),
ObjectId('5cd16b00da133e7fded3c23a')]

Count of deleted items:
1  documents deleted.

Count of updated items:
2 documents updated.

Simple Query Return:
{'_id': ObjectId('5cd16b00da133e7fded3c232'), 'Studno': 'HRRJES004', 'Name':
'Jessica Harris', 'Courseinfo': {'Year': 2011, 'Coursecode': 'ZOO101',
'Coursename': 'Zoology 101', 'Mark': 76, 'Credits': 24, 'Major': True,
'Senior': True}}
{'_id': ObjectId('5cd16b00da133e7fded3c235'), 'Studno': 'WDLTIF001', 'Name':
'Tiffany Woodley', 'Courseinfo': {'Year': 2011, 'Coursecode': 'ZOO101',
'Coursename': 'Zoology 101', 'Mark': 62, 'Credits': 24, 'Major': True,
'Senior': True}}
{'_id': ObjectId('5cd16b00da133e7fded3c238'), 'Studno': 'DOEJON002', 'Name':
'John Doe', 'Courseinfo': {'Year': 2011, 'Coursecode': 'ZOO101', 'Coursename':
'Zoology 101', 'Mark': 65, 'Credits': 24, 'Major': True, 'Senior': True}}

Complex Query Return:
{'_id': ObjectId('5cd16b00da133e7fded3c232'), 'Studno': 'HRRJES004', 'Name':
'Jessica Harris', 'Courseinfo': {'Year': 2011, 'Coursecode': 'ZOO101',
'Coursename': 'Zoology 101', 'Mark': 76, 'Credits': 24, 'Major': True,
'Senior': True}}
```

6 Group Contributions

Question	Contribution
Question 1	Tiffany
Question 2.1 Key-Store	Tiffany
Question 2.2 Document	Jessica
Question 2.3 Graphical	Jessica
Question 2.4 Column Family	Jessica
Question 3	Jessica
Question 4	Tiffany
Question 5	Tiffany
Report Compilation	Jessica and Tiffany

References

[1] Riak. Nosql databases. https://riak.com/resources/nosql-databases/index.html?p=9937.html, 2019. Accessed: 2019-05-09.

[2] Kevin Tardivel. Nosql databases. https://sonra.io/2017/06/12/benefits-graph-databases-data-warehousing/, June 2019. Accessed: 2019-05-09.