

CS-A1153 Databases, Summer 2020

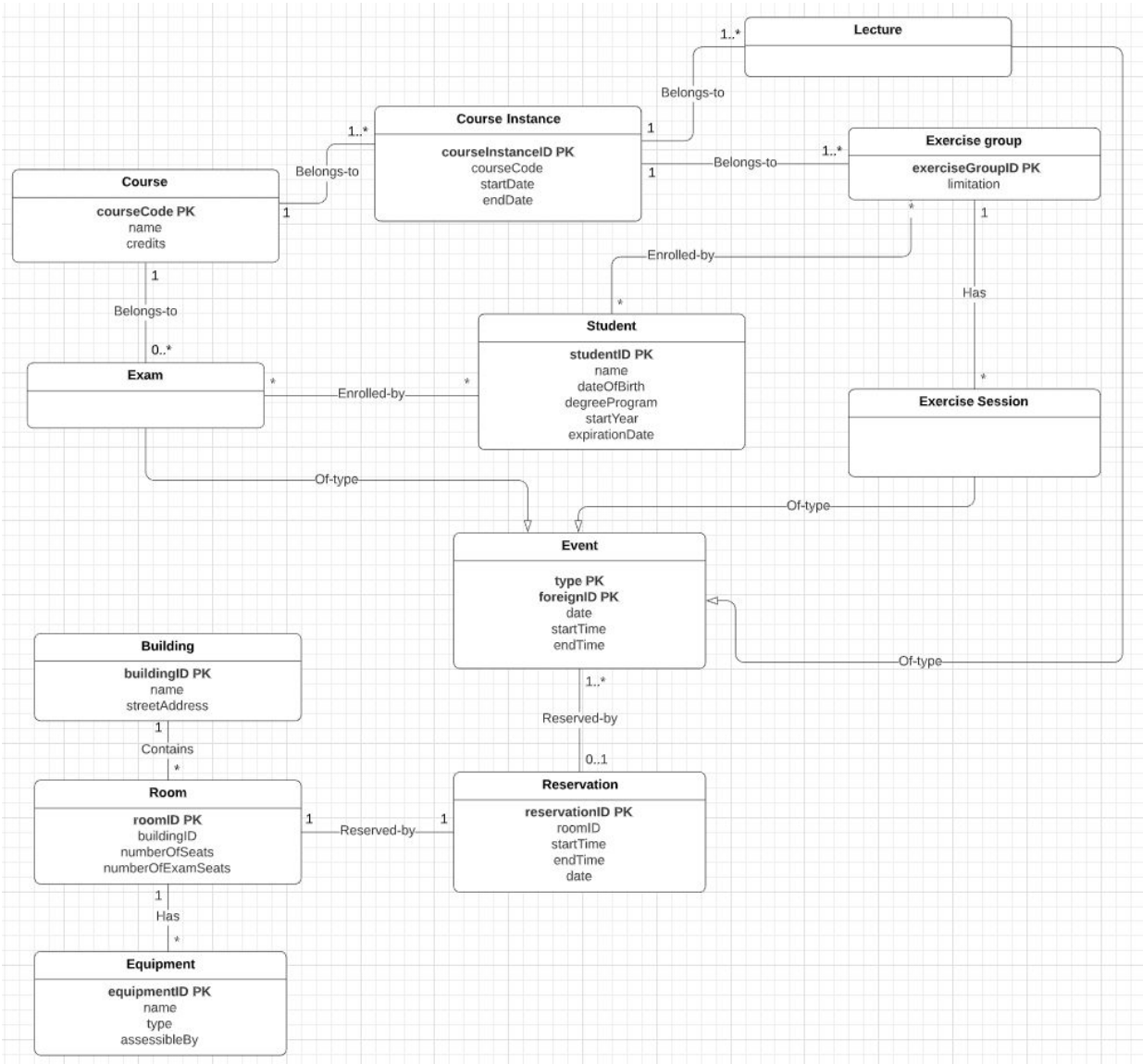
- Project Part 2 -

Dat Nguyen (717283, dat.t.nguyen@aalto.fi)

Khoa Lai (732255, khoa.lai@aalto.fi)

Jeheon Kim (716954, jeheon.kim@aalto.fi)

1. UML-diagram



2. Relational Schema

- **Student** (StudentID, Name, DateOfBirth, DegreeProgram, StartYear, ExpirationDate).
- **Event** (Type, ForeignID, StartTime, EndTime, Date, ReservationID).
- **Course** (CourseCode, Name, Credits).
- **Course Instance** (CourseInstanceID, CourseCode, StartDate, EndDate).
- **Exam** (ExamID, CourseCode).
- **Exam Enrollment** (ExamID, StudentID).
- **Lecture** (LectureID, CourseInstanceID).
- **Exercise group** (ExerciseGroupID, CourseInstanceID, Limitation).
- **Exercise group Enrollment** (ExerciseGroupID, StudentID).
- **Exercise session** (ExerciseSessionID, ExerciseGroupID).
- **Building** (BuildingID, Name, Address).
- **Room** (RoomID, BuildingID, NumberOfSeats, NumberOfExamSeats).
- **Reservation** (ReservationID, RoomID, StartTime, EndTime, Date).
- **Equipment** (EquipmentID, Name, Type, AssessibleBy).

3. Explanation of the Solution

Introduction

The above UML-diagram illustrates the university database and its components. Some of the components have the Many-To-Many relationship, such as Course - Course Instance, Student - Exam Enrollment, etc, while the other components are Many-To-One relationship, for instance: Building - Room, Room - Reservation, etc.

Course + Course Instance

The Course has the CourseCode as the primary key, and other foreign keys are *Name and Credits*. A course could have multiple instances. Each instance is different from each other through the time when the course is organized, that a course can be organized either in the same semester or in different semesters. A course may have lectures, exercise groups, and exams. Through the CourseInstanceID, it allows the student to query in the database and get information about the time of the Lecture and Exercise Groups. It is possible to query about exams of a certain course through the primary key CourseCode of the Course class.

Student

The student has the student ID as the primary key and other foreign keys, including *Name, DateOfBirth, DegreeProgram, StartYear, ExpirationDate*. Through the Student ID, it allows for the teacher/teaching assistant to perform the querying operations for various

purposes. For example: checking how many and who attends a specific exercise session, who registers for the lecture.

Building

The Building has 3 attributes (*BuildingID*, *Name*, *StreetAddress*), with *BuildingID* being the primary key. The university has several buildings, each of them contains rooms for teaching purposes (like lecture halls and classrooms). For each building, the database contains information about its *name and street address* through the corresponding attribute, and also it is possible to know which rooms are located in a certain building through querying using the *BuildingID* attribute.

Room

The Room has its ID as the primary key and other foreign keys are *the BuildingID*, *NumberOfSeats*, *NumberOfExamSeats* (*it could be lesser because the students cannot sit too near each other in the exam*). *RoomID* allows users to make queries about which rooms are currently being reserved, the Start Time and End Time of the reservation. A room can be used for either a lecture, an exercise session, or an exam. Room table has a one-to-one relationship with Reservation, where one Reservation record can only belong to one Room, and one Room can have only one Reservation at one time.

Reservation

In the Reservation, it has its ReservationID as its primary key, and other attributes, including the *RoomID*, *StartTime*, *EndTime*, *Date*. The StartTime and EndTime attribute allow the user to know the length of the reservation, so he/she could make the reservation at another vacant time. Reservation table has a one-to-many relationship with the Event table, where an Event record can only belong to one Reservation, but one Reservation can have multiple events happening at the same time. This enables the possibility of having multiple exams in the same room.

Equipment

The Room has the One-To-Many aggregation relationship with the Equipment, that one room could have many pieces of equipment inside the room (like a video projector, a computer for the teacher, a document camera, computers for students, etc.). Each Equipment has the EquipmentID as its primary key, and other attributes such as *RoomID*, *Name*, *Type*, *AccessibleBy*.

Event

Event class has subclasses for different types of events: Lecture, Exam, ExerciseSession. The Event class has a composite key, consisting of Type and ForeignID. Type field is either of "Exam", "Lecture", or "ExerciseSession", and ForeignID is the foreign key of the

aforementioned field. This means, if Type is “Exam” and ForeignID is 2, this record has a subclass record in the Exam table and its ID is 2.

The inheritance design allows for the reservations to be booked by multiple event classes. In the first iteration, we had 3 fields in the Reservation table, one for ExamID, one for LectureID and one for ExerciseSessionID. But this poses 2 problems:

- This doesn't allow multiple exams happening in the same room, since the ExamID field can only hold one value.
- If you want to add another table that can make a Reservation, we would need to add a field containing its foreign key to the Reservation table. This becomes more complicated when more event types are added to the database.

The Event table has a one-to-many relationship to the Reservation table, where an event can only belong to one Reservation, and a Reservation can have multiple events at the same time. As mentioned earlier, this allows for multiple exams in the same room.

Exam

The Exam has examID as the primary key and CourseCode as a foreign key to the one table Courses. The exam date and time are inherited from the superclass Event. Exam class has a one-to-many relationship with the Course table. An exam can only belong to one Course, while a Course can have multiple Exam records.

Lecture

The Lecture has 2 attributes, *LectureID* and *CourseInstanceID*, with *LectureID* being the primary key. A lecture belongs to a certain Course instance and it is possible to find the course in question by the foreign key, *CourseInstanceID*, in the table. Similar to the Exam table, Lecture inherits its date and time from the Event super class.

Exercise Group

The Exercise Group has its ID as the primary key and a foreign key: *CourseInstanceID*. The Limitation attribute allows the teaching assistant / student to see how many students have enrolled in an exercise group so far, and also the database could take into account its value to prevent more students from enrolling if it has reached its limit.

Exercise Session

Exercise Session has a primary key in *ExerciseSessionID* and a foreign key in *ExerciseGroupID*. Exercise Session also inherits its date and time from Event. Exercise Session class has a one-to-many relationship with the Exercise Group table. A session can only belong to one exercise group, while a group can have multiple sessions.

4. Functional Dependencies

Functional Dependency defines the relationship between Primary Key and other non-key attributes. For any relation, attribute Y is functionally dependent on attribute X (usually a Primary Key or a composite of Primary Keys), if for every valid instance of X, that value of X uniquely determines the value of Y. Therefore, in the case of our project database:

- In the relation “**Course**”
: CourseCode \rightarrow Name, Credits
- In the relation “**CourseInstance**”
: CourseInstanceID \rightarrow CourseCode, StartDate, EndDate, Semester
- In the relation “**Lecture**”
: LectureID \rightarrow CourseInstanceID
- In the relation “**Exercise Group**”
: ExerciseGroupID \rightarrow CourseInstanceID
- In the relation “**Exercise Session**”
: ExerciseSessionID \rightarrow ExerciseGroupID
- In the relation “**Exam**”
: ExamID \rightarrow CourseCode
- In the relation “**Student**”
: StudentID \rightarrow Name, DateOfBirth, DegreeProgram, StartYear, ExpirationDate
- In the relation “**Building**”
: BuildingID \rightarrow Name, StreetAddress
- In the relation “**Room**”
: RoomID \rightarrow BuildingID, NumberOfSeats, NumberOfExamSeats

- In the relation “**Reservation**”
: ReservationID → RoomID, StartTime, EndTime, Date
- In the relation “**Equipment**”
: EquipmentID → Name, Type, AssessibleBy

5. Anomalies

- **Redundancy:** None
- **Deletion Anomalies:** None
- **Update Anomalies:** None

6. BCNF Decomposition

All the tables are in BCNF form.

- **Student** (Name, StudentID, DateOfBirth, DegreeProgram, YearStarted, ExpirationDate).

FD: StudentID \rightarrow Name, DateOfBirth, DegreeProgram, StartYear, ExpirationDate.

StudentID is a superkey for Students \Rightarrow BCNF

- **Course** (CourseCode, Name, Credit).

FD: CourseCode \rightarrow Name, Credits

CourseCode is a superkey for Course \Rightarrow BCNF

- **Course Instance** (CourseInstanceID, CourseCode, StartDate, EndDate).

FD: CourseInstanceID \rightarrow CourseCode, StartDate, EndDate

CourseInstanceID is a super key for CourseInstance \Rightarrow BCNF

- **Exam** (ExamID, CourseCode).

FD: ExamID \rightarrow CourseCode

ExamID is a superkey for Exam \Rightarrow BCNF

- **Lecture** (LectureID, CourseInstanceID)

FD: LectureID \rightarrow CourseInstanceID

LectureID is a superkey for Lecture \Rightarrow BCNF.

- **Exercise group** (ExerciseGroupID, CourseInstanceID, Limit)

FD: ExerciseGroupID \rightarrow CourseInstanceID, Limit

- **Exercise Group Enrollment** (StudentID, ExerciseGroupID).

There are no functional dependencies for this table.

- **Exercise Session** (ExerciseSessionID, ExerciseGroupID).
FD: ExerciseGroupID → ExerciseGroupID
ExerciseGroupID is a superkey for this table => BCNF
- **Building** (BuildingID, Name, Address)
FD: BuildingID → Name, Address
BuildingID is a superkey for Building table => BCNF
- **Room** (RoomID, NumberOfSeats, NumberOfExamSeats, BuildingID).
FD: RoomID → NumberOfSeats, NumberOfExamSeats, BuildingID
RoomID is a superkey for Room table => BCNF
- **Reservation** (ReservationID, RoomID, StartTime, EndTime, Date).
FD: ReservationID → RoomID, StartTime, EndTime, Date
ReservationID is a superkey for Reservation => BCNF
- **Equipment** (EquipmentID, RoomID, Name, Type, AssessibleBy).
FD: EquipmentID → RoomID, Name, Type, AssessibleBy
EquipmentID is a superkey for Equipment => BCNF
- **Event** (Type, ForeignID, StartTime, EndTime, Date, ReservationID).
FD: Type, ForeignID → StartTime, EndTime, Date, ReservationID
Type, ForeignID is a superkey for Event => BCNF

7. Modifications and supplements to the solution of the first part

The biggest change from the original UML is the inclusion of an Event table as a subclass for Lecture, Exam and ExerciseSession. As mentioned in the previous part, the inheritance design provided by this Event table enables the reservations to be booked by multiple event classes. In the first iteration, we had 3 fields in the Reservation table, one for ExamID, one for LectureID and one for ExerciseSessionID. But this poses 2 problems:

- This doesn't allow multiple exams happening in the same room, since the ExamID field can only hold one value. The Event table inclusion solves this problem, thanks to the one-to-many relation between the Event table and the Reservation table.
- If you want to add another table that can make a Reservation, we would need to add a field containing its foreign key to the Reservation table. This becomes more complicated when more event types are added to the database. However, with the Event table, we only need the relation between the table of new event types and Event table.

8. Use cases

Explanations of the use cases and SQL statements run in SQLiteStudio environment (for example, the purpose of the various queries) and listings of the outputs of the statements. If some of the SQL statements purposely causes an error (for example, insertion does not succeed because it violates some consistency conditions), explain it in your documentation. Note that the SQL commands are listed both in the document and sql-file.

In this document, only the general form of the query is provided (without any specific values). However, the actual queries for testing with some example values are available in the attached file: UseCase_Queries.sql

1. The student wants to search the courses that are arranged during the certain time interval.

The query returns

: CourseName, CourseInstanceID, StartDate, EndDate

```
-----
SELECT Name, CourseInstanceID, StartDate, EndDate
FROM (SELECT *
      FROM Course JOIN CourseInstance
      ON Course.CourseCode = CourseInstance.CourseCode
      )
WHERE StartDate >= 'yyyy-mm-dd' AND EndDate <= 'yyyy-mm-dd';
-----
```

Course Table:

	CourseCode	Name	Credits
1	CS-A1111	Basic Course in Programming Y1	5
2	CS-A1120	Programming 2	4
3	CS-A1130	Superhuman Analysis	6
4	CS-A1140	Fus Ro Dah	3
5	CS-A1150	Test-Course	3

CourseInstance Table:

	CourseInstanceID	CourseCode	StartDate	EndDate
1	1	CS-A1111	2020-01-01	2020-03-01
2	2	CS-A1111	2020-04-01	2020-06-01
3	3	CS-A1120	2021-01-01	2021-03-01
4	4	CS-A1130	2021-04-01	2021-06-01
5	5	CS-A1140	2021-01-01	2021-03-01

Example Query Result:

```
1 SELECT Name, CourseInstanceID, StartDate, EndDate
2 FROM (SELECT *
3 FROM Course JOIN CourseInstance
4 ON Course.CourseCode = CourseInstance.CourseCode
5 )
6 WHERE StartDate >= '2020-01-02' AND EndDate <= '2025-01-01';
```

Grid view Form view

Total rows loaded: 4

	Name	CourseInstanceID	StartDate	EndDate
1	Basic Course in Programming Y1	2	2020-04-01	2020-06-01
2	Programming 2	3	2021-01-01	2021-03-01
3	Superhuman Analysis	4	2021-04-01	2021-06-01
4	Fus Ro Dah	5	2021-01-01	2021-03-01

2. The professor wants to check if there is any reservation for a room on the certain date and the certain time interval.

The query returns

: ReservationID, RoomID, StartTime, EndTime, Date

```
SELECT RoomID, ReservationID, Date, StartTime, EndTime
FROM Reservation
WHERE RoomID = ~ AND Date = 'yyyy-mm-dd'
      AND StartTime >= 'hh:mm:ss' AND EndTime <= 'hh:mm:ss';
```

Reservation Table:

	ReservationID	RoomID	StartTime	EndTime	Date
1	1	1	08:00:00	13:00:00	2020-05-01
2	2	2	08:00:00	13:00:00	2020-03-01
3	3	3	08:00:00	13:00:00	2020-03-01
4	4	4	12:00:00	17:00:00	2020-09-01
5	5	4	12:00:00	17:00:00	2020-09-01
6	6	4	12:00:00	17:00:00	2020-11-01
7	7	5	12:00:00	17:00:00	2020-01-31
8	8	6	12:00:00	17:00:00	2020-09-01
9	9	6	09:00:00	10:00:00	2020-01-01
10	10	5	09:00:00	10:00:00	2020-01-02
11	11	5	09:00:00	10:00:00	2020-01-03
12	12	5	09:00:00	10:00:00	2020-01-04
13	13	5	09:00:00	10:00:00	2020-01-05

Example Query Result:

```

30 SELECT RoomID, ReservationID, Date, StartTime, EndTime
31 FROM Reservation
32 WHERE RoomID = 6 AND StartTime >= '06:00:00' AND EndTime <= '16:00:00'
33    AND Date = '2020-01-01';

```

RoomID	ReservationID	Date	StartTime	EndTime
6	9	2020-01-01	09:00:00	10:00:00

- The student wants to know which exams a certain course has during the certain time interval.

The query returns

: CourseCode, CourseName, ExamDate, StartTime, EndTime

```

SELECT CourseCode, Name AS CourseName, Date AS ExamDate,
       StartTime, EndTime
FROM (SELECT *
      FROM Course, Exam
      WHERE Course.CourseCode = Exam.CourseCode), Event
WHERE ExamID = ForeignID AND Type = 'Exam'
      AND Name = '~'
      AND StartTime >= 'hh:mm:ss' AND EndTime <= 'hh:mm:ss';

```

Course Table:

	CourseCode	Name	Credits
1	CS-A1111	Basic Course in Programming Y1	5
2	CS-A1120	Programming 2	4
3	CS-A1130	Superhuman Analysis	6
4	CS-A1140	Fus Ro Dah	3
5	CS-A1150	Test-Course	3

Exam Table:

	ExamID	CourseCode
1	1	CS-A1111
2	2	CS-A1111
3	3	CS-A1120
4	4	CS-A1120
5	5	CS-A1130
6	6	CS-A1140

Event Table:

	Type	Date	StartTime	EndTime	ForeignID	Reservatic
1	Exam	2020-03-01	09:00:00	12:00:00	1	NULL
2	Exam	2020-05-01	09:00:00	12:00:00	2	1
3	Exam	2020-05-01	09:00:00	12:00:00	3	1
4	Exam	2020-07-01	13:00:00	16:00:00	4	NULL
5	Exam	2020-09-01	13:00:00	16:00:00	5	NULL
6	Exam	2020-11-01	13:00:00	16:00:00	6	NULL






Example Query Result:

```




47 SELECT CourseCode, Name AS CourseName, Date AS ExamDate, StartTime, EndTime
48 FROM (SELECT *
49       FROM Course, Exam
50       WHERE Course.CourseCode = Exam.CourseCode), Event
51 WHERE ExamID = ForeignID AND Type = 'Exam'
52       AND Name = 'Basic Course in Programming Y1'
53       AND StartTime >= '08:00:00' AND EndTime <= '13:00:00';

```

Grid view Form view



1



Total rows loaded: 2

	CourseCode	CourseName	ExamDate	StartTime	EndTime
1	CS-A1111	Basic Course in Programming Y1	2020-03-01	09:00:00	12:00:00
2	CS-A1111	Basic Course in Programming Y1	2020-05-01	09:00:00	12:00:00

4. The student wants to find out when a certain course which has been arranged or will be arranged.

The query returns

: CourseCode, CourseName, StartDate, EndDate

```
-----  
SELECT CourseCode, Name, StartDate, EndDate  
FROM (SELECT *  
      FROM Course, CourseInstance  
      WHERE Course.CourseCode = CourseInstance.CourseCode)  
WHERE Name = '~' OR CourseCode = 'CS-AXXXX';  
-----
```

Course Table:

	CourseCode	Name	Credits
1	CS-A1111	Basic Course in Programming Y1	5
2	CS-A1120	Programming 2	4
3	CS-A1130	Superhuman Analysis	6
4	CS-A1140	Fus Ro Dah	3
5	CS-A1150	Test-Course	3

CourseInstance Table:

	CourseInstanceID	CourseCode	StartDate	EndDate
1		1 CS-A1111	2020-01-01	2020-03-01
2		2 CS-A1111	2020-04-01	2020-06-01
3		3 CS-A1120	2021-01-01	2021-03-01
4		4 CS-A1130	2021-04-01	2021-06-01
5		5 CS-A1140	2021-01-01	2021-03-01

Example Query Result:

```
69 SELECT CourseCode, Name, StartDate, EndDate  
70 FROM (SELECT *  
71       FROM Course, CourseInstance  
72       WHERE Course.CourseCode = CourseInstance.CourseCode)  
73 WHERE Name = 'Basic Course in Programming Y1' OR CourseCode = 'CS-A9999';
```

Grid view Form view

Total rows loaded: 2

	CourseCode	Name	StartDate	EndDate
1	CS-A1111	Basic Course in Programming Y1	2020-01-01	2020-03-01
2	CS-A1111	Basic Course in Programming Y1	2020-04-01	2020-06-01

5. Find the lectures belonging to a certain course instance.

The query returns: LectureID, Date, StartTime, EndTime

```
SELECT LectureID, Date, StartTime, EndTime
FROM (SELECT *
      FROM Lecture JOIN Event
      ON Lecture.LectureID = Event.ForeignID )
WHERE Type = 'Lecture' AND CourseInstanceID = ~;
```

Lecture Table:

	LectureID	CourseInstanceID
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	6	2
7	10	2
8	11	2
9	12	2
10	7	3
11	13	3
12	8	4
13	9	5

Event Table (Type = 'Lecture'):

	Type	Date	StartTime	EndTime	ForeignID	Reservatic
1	Lecture	2020-01-01	09:00:00	10:00:00	1	9
2	Lecture	2020-01-02	09:00:00	10:00:00	2	10
3	Lecture	2020-01-03	09:00:00	10:00:00	3	11
4	Lecture	2020-01-04	09:00:00	10:00:00	4	12
5	Lecture	2020-01-05	09:00:00	10:00:00	5	13
6	Lecture	2020-01-01	09:00:00	10:00:00	6	NULL
7	Lecture	2020-02-27	09:00:00	10:00:00	7	NULL
8	Lecture	2020-03-03	09:00:00	10:00:00	8	NULL
9	Lecture	2020-03-04	09:00:00	10:00:00	9	NULL
10	Lecture	2020-02-05	09:00:00	10:00:00	10	NULL
11	Lecture	2020-02-05	09:00:00	10:00:00	11	NULL
12	Lecture	2020-03-05	09:00:00	10:00:00	12	NULL
13	Lecture	2020-04-05	09:00:00	10:00:00	13	NULL

Example Query Result:

```
2 SELECT LectureID, Date, StartTime, EndTime
3 FROM (SELECT *
4       FROM Lecture JOIN Event
5            ON Lecture.LectureID = Event.ForeignID
6       )
7 WHERE Type = 'Lecture' AND CourseInstanceID = 2;
```

Grid view Form view

Total rows loaded: 4

	LectureID	Date	StartTime	EndTime
1	6	2020-01-01	09:00:00	10:00:00
2	10	2020-02-05	09:00:00	10:00:00
3	11	2020-02-05	09:00:00	10:00:00
4	12	2020-03-05	09:00:00	10:00:00

6. The student wants to find the exercise groups belonging to the certain course instance along with the information on its limitation number and how many students have enrolled for the course.

The query returns

: ExerciseGroupID, CurrentlyEnrolledNumber, Limitation

```
SELECT ExerciseGroupID, Limitation, COUNT(StudentID) AS EnrolledNumber
FROM (SELECT *
      FROM ExerciseGroup LEFT OUTER JOIN ExerciseGroupEnrollment
        ON ExerciseGroup.ExerciseGroupID =
           ExerciseGroupEnrollment.ExerciseGroupID )
WHERE CourseInstanceID = ~
GROUP BY ExerciseGroupID;
```

ExerciseGroup Table:

	ExerciseGroupID	CourseInstanceID	Limitation
1	1	1	10
2	2	1	20
3	3	2	20
4	4	2	20
5	5	3	20
6	6	4	20
7	7	5	50

ExerciseGroupEnrollment Table:

	ExerciseGroupID	StudentID
1	1	112233
2	1	123456
3	1	224411
4	1	224412
5	1	224413
6	2	112233
7	3	112233
8	4	224411
9	4	224412

Example Query Result:






```

3 SELECT ExerciseGroupID, Limitation, COUNT(StudentID) AS EnrolledNumber
4     FROM (SELECT *
5           FROM ExerciseGroup LEFT OUTER JOIN ExerciseGroupEnrollment
6           ON ExerciseGroup.ExerciseGroupID = ExerciseGroupEnrollment.ExerciseGroupID
7           )
8     WHERE CourseInstanceID = 1
9     GROUP BY ExerciseGroupID;




```

Grid view

Form view



1



Total rows loaded: 2

	ExerciseGroupID	Limitation	EnrolledNumber
1	1	10	5
2	2	20	1

- The student wants to find out when and which room a certain exercise group is planned to meet for the exercise session.

The query returns

: ExerciseGroupID, ReservationID, RoomID, Date, StartTime, EndTime

```

SELECT ForeignID AS ExGroupID, ReservationID, RoomID, Date,
       Reservation.StartTime, Reservation.EndTime
FROM (SELECT *
      FROM Event
      WHERE Type = 'ExerciseSession') AS NewEvent LEFT JOIN Reservation
      ON NewEvent.ReservationID = Reservation.ReservationID
WHERE ExGroupID= ~;

```


Event Table (Type = 'ExerciseSession'):

	Type	Date	StartTime	EndTime	ForeignID	ReservationID
7	ExerciseSession	2020-09-01	13:00:00	16:00:00	1	NULL
8	ExerciseSession	2020-10-01	13:00:00	16:00:00	2	NULL
9	ExerciseSession	2020-01-01	11:00:00	18:00:00	3	NULL
10	ExerciseSession	2020-01-31	08:00:00	16:00:00	4	NULL
11	ExerciseSession	2019-09-01	13:00:00	16:00:00	5	NULL
12	ExerciseSession	2020-08-01	01:00:00	06:00:00	6	NULL
13	ExerciseSession	2020-09-01	13:00:00	16:00:00	7	8
14	ExerciseSession	2020-02-01	13:00:00	16:00:00	8	NULL
15	ExerciseSession	2020-09-01	13:00:00	16:00:00	9	NULL
16	ExerciseSession	2020-07-01	13:00:00	16:00:00	10	NULL

Reservation Table:

	ReservationID	RoomID	StartTime	EndTime	Date
1	1	1	08:00:00	13:00:00	2020-05-01
2	2	2	08:00:00	13:00:00	2020-03-01
3	3	3	08:00:00	13:00:00	2020-03-01
4	4	4	12:00:00	17:00:00	2020-09-01
5	5	4	12:00:00	17:00:00	2020-09-01
6	6	4	12:00:00	17:00:00	2020-11-01
7	7	5	12:00:00	17:00:00	2020-01-31
8	8	6	12:00:00	17:00:00	2020-09-01
9	9	6	09:00:00	10:00:00	2020-01-01
10	10	5	09:00:00	10:00:00	2020-01-02
11	11	5	09:00:00	10:00:00	2020-01-03
12	12	5	09:00:00	10:00:00	2020-01-04
13	13	5	09:00:00	10:00:00	2020-01-05

Example Query Result:

```

3 SELECT ForeignID AS ExGroupID, ReservationID, RoomID, Date, Reservation.StartTime, Reservation.EndTime
4 FROM (SELECT *
5       FROM Event
6       WHERE Type = 'ExerciseSession') AS NewEvent LEFT JOIN Reservation
7       ON NewEvent.ReservationID = Reservation.ReservationID
8 WHERE ExGroupID= 7;

```

Grid view

Form view

1

Total rows loaded: 1

	ExGroupID	ReservationID	RoomID	Date	StartTime	EndTime
1	7	8	6	2020-09-01	12:00:00	17:00:00

8. The professor wants to find a room which has at least a certain number of seats and is free for a reservation on the certain date and during the certain time interval.

The query returns: RoomID, BuildingID, NumberOfSeats


```
-----  
Select DISTINCT RoomID, BuildingID, NumberOfSeats  
FROM Room, Reservation  
WHERE NumberOfSeats >= ~ AND Room.RoomID NOT IN  
      (SELECT RoomID  
       FROM Reservation  
       WHERE Date = 'yyyy-mm-dd'  
            AND (('hh:mm:ss' > StartTime AND 'hh:mm:ss' < EndTime)  
            OR ('hh:mm:ss' > StartTime AND 'hh:mm:ss' < EndTime))  
      );  
-----
```

The SQL's BETWEEN operator is natively inclusive (Greater than Equal to / Less than or Equal to) so it wasn't appropriate to implement our idea. (For example, if the other reservation is until 10:00:00 then the new reservation should be possible from the same time 10:00:00) So we chose this bit of unclean query redundancy of StartTime and EndTime.

Room Table:

	RoomID	BuildingID	NumberOfSeats	NumberOfExamSeats
1	1	1	50	30
2	2	1	40	30
3	3	1	50	20
4	4	1	10	5
5	5	2	10	5
6	6	2	10	5
7	7	3	10	5
8	8	3	10	5
9	9	4	10	5

Reservation Table:

	ReservationID	RoomID	StartTime	EndTime	 Date
1	6	4	12:00:00	17:00:00	2020-11-01
2	4	4	12:00:00	17:00:00	2020-09-01
3	5	4	12:00:00	17:00:00	2020-09-01
4	8	6	12:00:00	17:00:00	2020-09-01
5	1	1	08:00:00	13:00:00	2020-05-01
6	2	2	08:00:00	13:00:00	2020-03-01
7	3	3	08:00:00	13:00:00	2020-03-01
8	7	5	12:00:00	17:00:00	2020-01-31
9	13	5	09:00:00	10:00:00	2020-01-05
10	12	5	09:00:00	10:00:00	2020-01-04
11	11	5	09:00:00	10:00:00	2020-01-03
12	10	5	09:00:00	10:00:00	2020-01-02
13	9	6	09:00:00	10:00:00	2020-01-01

Example Query Result:


```


3 Select DISTINCT RoomID, BuildingID, NumberOfSeats
4 FROM Room, Reservation
5 WHERE NumberOfSeats >= 5 AND Room.RoomID NOT IN
6   (SELECT RoomID
7    FROM Reservation
8    WHERE Date = '2020-09-01'
9      AND (('10:00:00' > StartTime AND '10:00:00' < EndTime)
10     OR ('13:00:00' > StartTime AND '13:00:00' < EndTime))
11   );


```


Grid view


Form view














1







Total rows loaded: 7

	RoomID	BuildingID	NumberOfSeats
1	1	1	50
2	2	1	40
3	3	1	50
4	5	2	10
5	7	3	10
6	8	3	10
7	9	4	10

9. Find out for what reservations the certain room has and their purpose of use.

The query returns: ReservationID, Purpose, Date, StartTime, Endtime

```
SELECT Event.ReservationID, Type AS PurposeOfUse, Reservation.Date,  
       StartTime, EndTime  
FROM Reservation LEFT JOIN Event  
       ON Reservation.ReservationID = Event.ReservationID  
WHERE Reservation.RoomID = ~;
```

Reservation Table:

	ReservationID	RoomID	StartTime	EndTime	Date
1	1	1	08:00:00	13:00:00	2020-05-01
2	2	2	08:00:00	13:00:00	2020-03-01
3	3	3	08:00:00	13:00:00	2020-03-01
4	4	4	12:00:00	17:00:00	2020-09-01
5	5	4	12:00:00	17:00:00	2020-09-01
6	6	4	12:00:00	17:00:00	2020-11-01
7	7	5	12:00:00	17:00:00	2020-01-31
8	8	6	12:00:00	17:00:00	2020-09-01
9	9	6	09:00:00	10:00:00	2020-01-01
10	10	5	09:00:00	10:00:00	2020-01-02
11	11	5	09:00:00	10:00:00	2020-01-03
12	12	5	09:00:00	10:00:00	2020-01-04
13	13	5	09:00:00	10:00:00	2020-01-05

Event Table (Tuples have constant value for Reservation ID are selected):


	Type	Date	StartTime	EndTime	ForeignID	ReservationID
22	Exam	2020-05-01	09:00:00	12:00:00	2	1
23	Exam	2020-05-01	09:00:00	12:00:00	3	1
24	ExerciseSession	2020-09-01	13:00:00	16:00:00	7	8
25	Lecture	2020-01-01	09:00:00	10:00:00	1	9
26	Lecture	2020-01-02	09:00:00	10:00:00	2	10
27	Lecture	2020-01-03	09:00:00	10:00:00	3	11
28	Lecture	2020-01-04	09:00:00	10:00:00	4	12
29	Lecture	2020-01-05	09:00:00	10:00:00	5	13


Example Query Result:


```
3 SELECT Event.ReservationID, Type AS PurposeOfUse, Reservation.Date, StartTime, EndTime
4 FROM Reservation LEFT JOIN Event
5     ON Reservation.ReservationID = Event.ReservationID
6 WHERE Reservation.RoomID = 6;
```


Grid view


Form view














1







Total rows loaded: 2

	ReservationID	PurposeOfUse	Date	StartTime	EndTime
1	8	ExerciseSession	2020-09-01	12:00:00	17:00:00
2	9	Lecture	2020-01-01	09:00:00	10:00:00

10. Enroll (Register) the certain student for the certain exam.

The query does not return any table but adds the new student to the certain exam. The result can be confirmed at the ExamEnrollment table.

```
INSERT INTO ExamEnrollment(StudentID, ExamID)
VALUES(~~~~~, ~);
```

Example Query:

```
1 INSERT INTO ExamEnrollment (StudentID, ExamID)
2 VALUES(112233,5);
```

Result:

	ExamID	StudentID
1	1	112233
2	2	112233
3	3	112233
4	1	123456
5	1	224411
6	4	224412
7	1	224412
8	1	224413
9	4	224411
10	5	112233

11. Enroll (Register) the certain student for the certain exercise group.

The query does not return a table but adds the new student to the certain exam. The result can be found in the ExerciseGroupEnrollment table.

```
-----  
INSERT INTO ExamEnrollment(StudentID, ExerciseGroupID)  
VALUES(~~~~~, ~);  
-----
```

Example Query:

```
1 INSERT INTO ExerciseGroupEnrollment (StudentID, ExerciseGroupID)  
2 VALUES(112233,5);
```

Result:

	ExerciseGroupID	StudentID
1	1	112233
2	1	123456
3	1	224411
4	1	224412
5	1	224413
6	2	112233
7	3	112233
8	4	224411
9	4	224412
10	5	112233

12. The student wants to know which exercise groups with a certain course instance, not full yet.

The query should return

: ExerciseGroupID, EnrolledNumber, Limitation, AvailableNumber

```
-----  
SELECT ExerciseGroupID, COUNT(StudentID) AS EnrolledNumber,  
        Limitation, Limitation - COUNT(StudentID) AS AvailableNumber  
FROM ExerciseGroup, ExerciseGroupEnrollment  
WHERE ExerciseGroup.ExerciseGroupID =  
        ExerciseGroupEnrollment.ExerciseGroupID
```

AND CourseInstanceID = ~
 GROUP BY ExerciseGroup.ExerciseGroupID
 HAVING AvailableNumber > 0

ExerciseGroup Table:

	ExerciseGroupID	CourseInstanceID	Limitation
1	1	1	10
2	2	1	20
3	3	2	20
4	4	2	20
5	5	3	20
6	6	4	20
7	7	5	50

ExerciseGroupEnrollment Table:

	ExerciseGroupID	StudentID
1	1	112233
2	1	123456
3	1	224411
4	1	224412
5	1	224413
6	2	112233
7	3	112233
8	4	224411
9	4	224412






Example Query Result:

```




3 SELECT ExerciseGroupID, COUNT(StudentID) AS EnrolledNumber, Limitation,
   Limitation - COUNT(StudentID) AS AvailableNumber
4 FROM ExerciseGroup, ExerciseGroupEnrollment
5 WHERE ExerciseGroup.ExerciseGroupID = ExerciseGroupEnrollment.ExerciseGroupID
6       AND CourseInstanceID = 2
7 GROUP BY ExerciseGroup.ExerciseGroupID
8 HAVING AvailableNumber > 0

```

Grid view Form view



1

Total rows loaded: 2

	ExerciseGroupID	EnrolledNumber	Limitation	AvailableNumber
1	3	1	20	19
2	4	2	20	18

13. List all students who have enrolled for the certain exercise group or the exam.

The query returns: StudentName, StudentID

```
-----  
SELECT DISTINCT Name, StudentID  
FROM Student  
WHERE StudentID IN  
    (SELECT StudentID  
     FROM ExerciseGroupEnrollment  
     WHERE ExerciseGroupID = ~  
    )  
OR StudentID IN  
    (SELECT StudentID  
     FROM ExamEnrollment  
     WHERE ExamID = ~  
    );  
-----
```

Student Table:

	StudentID	Name	DateOfBirth	DeegreePro	StartYear	ExpirationDate
1	112233	Teemu Teekkari	1995-12-25	TIK	2017	2021-05-30
2	123456	Tiina Teekkari	1991-11-26	TUO	2019	2023-05-30
3	224411	Van Anh Do	1998-03-27	AUT	2021	2025-12-31
4	224412	Khoa Lai	1998-03-27	TIK	2018	2021-12-31
5	224413	Jeheon Kim	1989-03-27	TIK	2018	2021-12-31

ExerciseGroupEnrollment Table:

	ExerciseGroupID	StudentID
1	1	112233
2	1	123456
3	1	224411
4	1	224412
5	1	224413
6	2	112233
7	3	112233
8	4	224411
9	4	224412

ExamEnrollment Table:

	ExamID	StudentID
1	1	112233
2	1	123456
3	1	224411
4	1	224412
5	1	224413
6	2	112233
7	3	112233
8	4	224411
9	4	224412

Example Query Result:

```
3 SELECT DISTINCT Name, StudentID
4 FROM Student
5 WHERE StudentID IN
6     (SELECT StudentID
7      FROM ExerciseGroupEnrollment
8      WHERE ExerciseGroupID = 4
9      )
10 OR StudentID IN
11     (SELECT StudentID
12      FROM ExamEnrollment
13      WHERE ExamID = 2
14     );
```

Grid view	Form view
Total rows loaded: 3	
Name	StudentID
1 Van Anh Do	224411
2 Khoa Lai	224412
3 Teemu Teekkari	112233

14. The professor wants to know the list of students who have enrolled for the certain exam.

The query returns: StudentName, StudentID

```
SELECT DISTINCT Name, StudentID
FROM Student
WHERE StudentID IN
    (SELECT StudentID
     FROM ExamEnrollment
     WHERE ExamID = ~ );
```

Student Table:

	StudentID	Name	DateOfBirth	DegreePro	StartYear	ExpirationDate
1	112233	Teemu Teekkari	1995-12-25	TIK	2017	2021-05-30
2	123456	Tiina Teekkari	1991-11-26	TUO	2019	2023-05-30
3	224411	Van Anh Do	1998-03-27	AUT	2021	2025-12-31
4	224412	Khoa Lai	1998-03-27	TIK	2018	2021-12-31
5	224413	Jeheon Kim	1989-03-27	TIK	2018	2021-12-31

ExamEnrollmentTable:

	ExamID	StudentID
1	1	112233
2	1	123456
3	1	224411
4	1	224412
5	1	224413
6	2	112233
7	3	112233
8	4	224411
9	4	224412

Example Query Result:


```


3 SELECT DISTINCT Name, StudentID
4 FROM Student
5 WHERE StudentID IN
6     (SELECT StudentID
7      FROM ExamEnrollment
8      WHERE ExamID = 1
9      );


```


Grid view


Form view














1







Total rows loaded: 5

	Name	StudentID
1	Teemu Teekkari	112233
2	Tiina Teekkari	123456
3	Van Anh Do	224411
4	Khoa Lai	224412
5	Jeheon Kim	224413

15. The supervisor of the exam wants to know the name of students who have enrolled for the certain course, and which student is taking which exam of the course (One course can have more than one exam).

The query returns: ExamID, StudentID, StudentName

```
-----  
SELECT ExamID, StudentID, Name  
FROM (SELECT *  
      FROM Exam, ExamEnrollment  
      WHERE Exam.ExamID = ExamEnrollment.ExamID) AS ExamCourse  
      JOIN Student  
      ON ExamCourse.StudentID = Student.StudentID  
WHERE CourseCode = 'CS-A1111';  
-----
```

Exam Table:

	ExamID	CourseCode
1	1	CS-A1111
2	2	CS-A1111
3	3	CS-A1120
4	4	CS-A1120
5	5	CS-A1130
6	6	CS-A1140

ExamEnrollment Table:

	ExamID	StudentID
1	1	112233
2	1	123456
3	1	224411
4	1	224412
5	1	224413
6	2	112233
7	3	112233
8	4	224411
9	4	224412

Student Table:

	StudentID	Name	DateOfBirth	DegreePrc	StartYear	ExpirationD
1	112233	Teemu Teekkari	1995-12-25	TIK	2017	2021-05-30
2	123456	Tiina Teekkari	1991-11-26	TUO	2019	2023-05-30
3	224411	Van Anh Do	1998-03-27	AUT	2021	2025-12-31
4	224412	Khoa Lai	1998-03-27	TIK	2018	2021-12-31
5	224413	Jeheon Kim	1989-03-27	TIK	2018	2021-12-31

Example Query Result:

```
3 SELECT ExamID, StudentID, Name
4 FROM (SELECT *
5       FROM Exam, ExamEnrollment
6       WHERE Exam.ExamID = ExamEnrollment.ExamID) AS ExamCourse JOIN Student
7       ON ExamCourse.StudentID = Student.StudentID
8 WHERE CourseCode = 'CS-A1111';
```

Grid view

Form view

1

Total rows loaded: 6

	ExamID	StudentID	Name
1	1	112233	Teemu Teekkari
2	1	123456	Tiina Teekkari
3	1	224411	Van Anh Do
4	1	224412	Khoa Lai
5	1	224413	Jeheon Kim
6	2	112233	Teemu Teekkari

16. Check all exams that are planned to be arranged on a certain date.

The query returns: CourseCode, ExamID, StartTime, EndTime

```
SELECT CourseCode, ExamID, Date, StartTime, EndTime
FROM Exam LEFT JOIN (SELECT *
                     FROM Event
                     WHERE Type = 'Exam') AS EventExam
ON Exam.ExamID = EventExam.ForeignID
WHERE date = 'yyyy-mm-dd';
```

Exam Table:

	ExamID	CourseCode
1	1	CS-A1111
2	2	CS-A1111
3	3	CS-A1120
4	4	CS-A1120
5	5	CS-A1130
6	6	CS-A1140

Event Table (Type = 'Exam'):

	Type	Date	StartTime	EndTime	ForeignID	ReservationID
1	Exam	2020-03-01	09:00:00	12:00:00	1	NULL
2	Exam	2020-05-01	09:00:00	12:00:00	2	1
3	Exam	2020-05-01	09:00:00	12:00:00	3	1
4	Exam	2020-07-01	13:00:00	16:00:00	4	NULL
5	Exam	2020-09-01	13:00:00	16:00:00	5	NULL
6	Exam	2020-11-01	13:00:00	16:00:00	6	NULL

Example Query Result:

```

3 SELECT CourseCode, ExamID, Date, StartTime, EndTime
4 FROM Exam LEFT JOIN (SELECT *
5                       FROM Event
6                       WHERE Type = 'Exam') AS EventExam
7                       ON Exam.ExamID = EventExam.ForeignID
8 WHERE date = '2020-05-01';

```

Grid view		Form view							
	<input checked="" type="checkbox"/>				1				Total rows loaded: 2
	CourseCode	ExamID	Date	StartTime	EndTime				
1	CS-A1111	2	2020-05-01	09:00:00	12:00:00				
2	CS-A1120	3	2020-05-01	09:00:00	12:00:00				

9. Index with justifications

Below are the explanations on “which indexes should be created” for each of the tables in the Database and its justification.

Building

For the Building, we would index on the **Name** column through the following command : ***CREATE UNIQUE INDEX Building_name ON Building (Name)***; When we create an index for that column, SQLite maintains an ordered list of the data within the index's columns as well as their records' primary keys. Indexing on BuildingID could reduce the time taken to perform an operation. Take 1 use case as an example (**Return the address of the Building Vare**), SQLite can use the ***Building_name*** index to perform a binary search on the BuildingID values to determine that the record we are looking for has **BuildingID = 6** and can then use that value in a second binary search against the actual Building table to return the full record.

In the worst case, we do not use any index method for the table. Thus, the time complexity it takes to query the use cases would be $O(n + n) = O(2n)$, as SQLite needs to linearly scan through all the records in the table and return the one that meets the requirement(s).

→ Using the ***Building_name*** index helps reduce the query's performance from $O(n)$ to $O(\log n + \log n) = O(\log n)$.

Course

For the Course, we would index on the **Name, Credits** through the following command: ***CREATE UNIQUE INDEX Course_Name_Credits ON Course (Name, Credits)***; The Credits column acts as an attribute that is clustered on a few storage pages, which results in a fewer page hit. That method is called “**Index on Clustered Attributes**”. Take 1 use case as an example (**Return a list of all courses that take place in a certain time interval**), so instead of linearly

scanning the whole record in the table and returning them one by one, now when you run the query, SQLite's query planner is smart enough to recognize that we *only* need data which is contained within the index and can return that data immediately after the index search without resorting to looking at the actual records in the `Course` table.

In addition, it is also common that the user would like to query the course that has a certain number of credits. Therefore, indexing on the `Name` attribute would also help to improve the performance of the query. Also, it is reasonable that besides showing how many credits a course awards, the user is likely to want to know what is the name of that course. From our point of view, many of the use cases are related to the operation among those 2 attributes, giving it a good reason to use what is called "Composite Indexes" for this table.

Assume that we use Index on only 1 attribute (either `Name` or `Credit`), the time complexity it takes for the operation is $O(\log n + \log n)$ (retrieve the `Name` / `Credit` and then use the returned value to query the second attribute). Using the "Composite Indexes" method, we are able to reduce the time complexity to $O(\log n)$, which is slightly faster given a reasonable small dataset. If the dataset becomes increasingly large, the improvement could become more noticeable.

In the worst case, we do not use any index method for the table. Thus, the time complexity it takes to query the use cases would be $O(n)$, as SQLite needs to linearly scan through all the records in the table and continually return the one that meets the requirement(s).

Equipment

For the `Equipment`, we would index on the `Name` attribute of the table ***CREATE INDEX Equipment_Name ON Equipment (Name);***. The `Name` attribute is not the primary key of the schema, however, there are only a handful of equipment that have the same name, so it is reasonable to make the index on it. In one use case when we want to query the rooms that have a particular

equipment, SQLite can use the ***Equipment_name*** index to perform a binary search on the name values to determine other attributes, thus being able to return the list of all rooms having the queried equipment. Thanks to SQLite maintaining an ordered list of the data within the index's columns, the time taken to perform the searching operation reduces from $O(n)$ to $O(\log n)$.

Event

For the Event, we would index on the **Type** attribute. The index operation could be done by using the following command ***CREATE INDEX Event_Type ON Event (Type)***; Due to the fact that after being indexed, the new index column represents its value in a list that is in sorted order, so it would avoid the linear scan across all records in a table for a particular use case. Some use cases that would benefit from having the Type indexed include **Return all the Exam / Exercise session / Lecture in a certain interval, Return which reservation(s) that have been made in a certain date, etc.**

Exam

For the Exam, we decided not to use the index for the table. It is because the Exam has 2 attributes (**ExamID and CourseCode**) and both of them have unique values, therefore, in any way the SQLite has to linearly scan through all the records in order to find the queried relation. Thus, no improvement in time complexity from indexing is found and the time taken is $O(n)$ due to the linear search.

ExerciseGroup

For the ExerciseGroup, we decided not to use the index for the table. It is because the ExerciseGroup has 2 attributes (**ExerciseID and limitation**) and both of them have unique values, therefore, in any way the SQLite has to linearly scan through all the records in order to find the queried relation. Thus, no improvement in time complexity from indexing is found and the time taken is $O(n)$ due to the linear search.

Reservation

For the Reservation, we use what are called “**Covering Indexes**” to eliminate the second binary search from the query planning operation. Here, the indexes are on 2 attributes of the table, namely **RoomID** and **Date**. The index operation could be done by using the following command ***CREATE INDEX***

Reservation_RoomID_Date ON ReservationEvent (RoomID, Date); It can be witnessed that most of the use cases regarding the Reservation would mainly need to know the information about these 2 attributes, i.g if the student would like to check if a room has been reserved on a particular date, return which rooms are still available on a certain date, etc. The choice of using the “Covering Indexes” is beneficial in this case, because SQLite’s query planner is smart enough to recognize that we only need data which is contained within the index and can return that data immediately after the index search without resorting to looking at the actual records in the Reservation table.

Assume that we use Index on only 1 attribute (either RoomID or Date), the time complexity it takes for the operation is $O(\log n + \log n)$ (retrieve the RoomID / Date and then use the returned value to query the second attribute). Using the “Covering Indexes” method, we are able to reduce the time complexity to $O(\log n)$, which is slightly faster given a reasonable small dataset. If the dataset becomes increasingly large, the improvement is more noticeable.

In the worst case, we do not use any index method for the table. Thus, the time complexity it takes to query the use cases would be $O(n)$, as SQLite needs to linearly scan through all the records in the table and continually return the one that meets the requirement(s).

Room

For the Room, we would index on the **buildingID** attribute. The index operation could be done by using the following command ***CREATE INDEX*** ***Room_buildingID ON Room (buildingID);*** Some use cases regarding

needing the information about the RoomID would be beneficial from the index. For example, when we want to know which rooms are in a particular building or when we know if a room in a particular building is available to be reserved, by having the index, SQLite will use the returned buildingID to narrow down the room in which it possibly yields the value that we need, instead of linearly scanning all the rooms in all buildings. This method is called ***“Index on ‘Almost’ Keys”*** because the buildingID is not the key of the Room table, but since it has unique value for each relations, therefore, indexing on it results in few page to hit in the query operation (compared to the total number of rooms in all buildings).

In the case we use Index, the time complexity reduces to $O(\log n)$ from $O(2n)$, which in the worst case means that we do not use any Index method and the SQLite needs to scan through all the data rows.

Student

For the Student, we would index on the **Name** attribute. The index operation could be done by using the following command ***CREATE UNIQUE INDEX Student_Name ON Student (Name);*** Although Name is not the key of Student, there are two reasons why indexing on it yields noticeable advantages:

- + In general, there are only a handful of students that have the same name.
- + So, few pages to read (compared to the total number of rows contained in the table).

The above mentioned method is called ***“Index on ‘Almost’ Keys”***, as the SQLite can narrow down the scope of possibilities when performing the query. For example, when the study planning officer would like to check how many courses / the expected Expiration Date of a certain student, the time taken to perform the query would be faster compared to the case where we do not index on the table ($O(\log n)$ vs $O(n)$).

In the worst case, we do not use any index method for the table. Thus, the time complexity it takes to query the use cases would be $O(n)$, as SQLite needs to linearly scan through all the records in the table and return the one that meets the requirement(s).

10. View Creation

A view is a result set of a stored query. A view is the way to pack a query into a named object stored in a database.

As an example, we create in total 5 views for 5 of our use cases, which are most likely to be common among the users of the Database:

1/ The student wants to find out when a certain course has been arranged or will be arranged.

/ 4. The student wants to find out when a certain course has been arranged or will be arranged.*

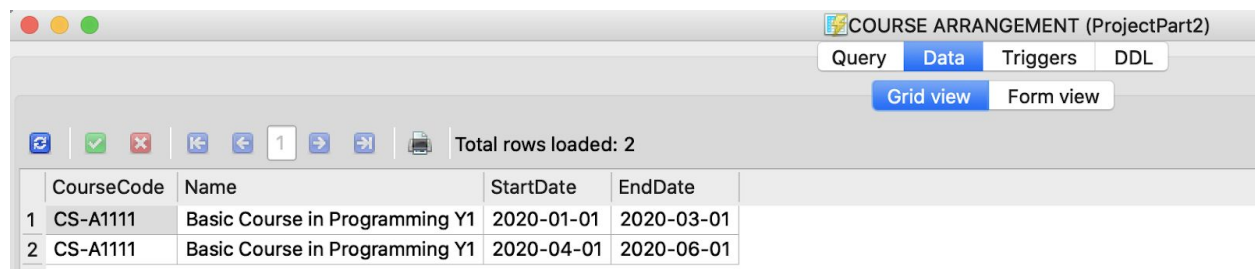
The query should return: CourseCode, CourseName, CourseStartDate, CourseEndDate

Here is an example query with following values

: CourseName = 'Basic Course in Programming Y1' or CourseCode = 'CS-A9999' */

```
DROP VIEW IF EXISTS [COURSE ARRANGEMENT];
CREATE VIEW[COURSE ARRANGEMENT] AS
SELECT CourseCode, Name, StartDate, EndDate
FROM (SELECT *
      FROM Course, CourseInstance
      WHERE Course.CourseCode = CourseInstance.CourseCode)
WHERE Name = 'Basic Course in Programming Y1' OR CourseCode = 'CS-A9999';
```

Result from the stored View:



	CourseCode	Name	StartDate	EndDate
1	CS-A1111	Basic Course in Programming Y1	2020-01-01	2020-03-01
2	CS-A1111	Basic Course in Programming Y1	2020-04-01	2020-06-01

2/ Find the lectures belonging to a certain course instance.

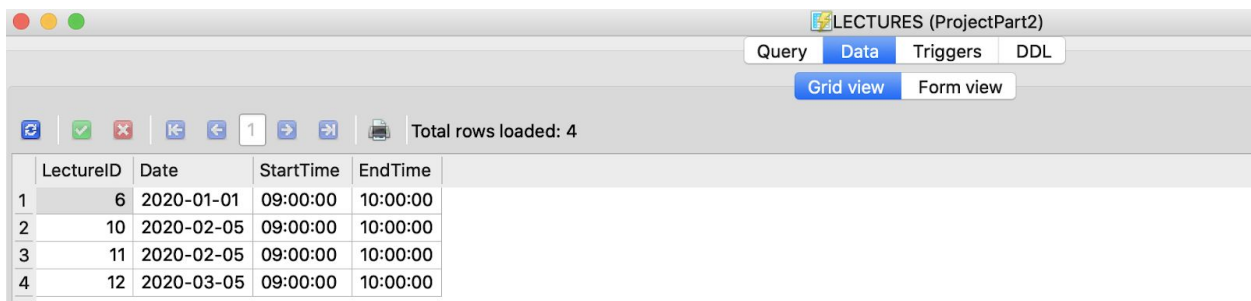
/ 5. Find the lectures belonging to a certain course instance.*

The query should return: LectureID, Date, StartTime, EndTime

Here is an example query with the following value
: CourseInstanceID = 2 */

```
DROP VIEW IF EXISTS [LECTURES];
CREATE VIEW[LECTURES] AS
SELECT LectureID, Date, StartTime, EndTime
FROM (SELECT *
      FROM Lecture JOIN Event
        ON Lecture.LectureID = Event.ForeignID
      )
WHERE Type = 'Lecture' AND CourseInstanceID = 2;
```

Result from the stored View:



	LectureID	Date	StartTime	EndTime
1	6	2020-01-01	09:00:00	10:00:00
2	10	2020-02-05	09:00:00	10:00:00
3	11	2020-02-05	09:00:00	10:00:00
4	12	2020-03-05	09:00:00	10:00:00

3/ The student wants to find out when and where a certain exercise group meets.

/ 7. The student wants to find out when and where a certain exercise group meets.*

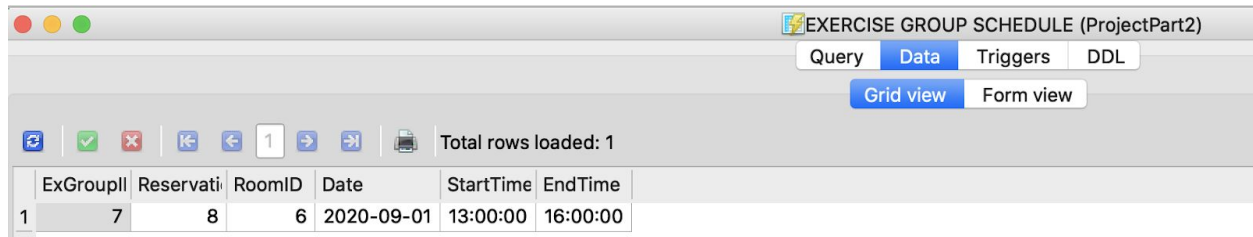
The query should return: ExerciseGroupID, ReservationID, RoomID, Date, StartTime, EndTime

Here are two example query with different following values:

1) ExerciseGroupID = 7 (which has a reservation on the certain room) */

```
DROP VIEW IF EXISTS [EXERCISE GROUP SCHEDULE];
CREATE VIEW[EXERCISE GROUP SCHEDULE] AS
SELECT ForeignID AS ExGroupID, ReservationID, RoomID, Date, StartTime, EndTime
FROM (SELECT *
      FROM Event
      WHERE Type = 'ExerciseSession') AS NewEvent LEFT JOIN Reservation
      ON NewEvent.ReservationID = Reservation.ReservationID
WHERE ExGroupID= 7;
```

Result from the stored View:



	ExGroupID	ReservationID	RoomID	Date	StartTime	EndTime
1	7	8	6	2020-09-01	13:00:00	16:00:00

4/ Find out for what reservations a certain room has and their purpose of use.

/* 9. Find out for what reservations a certain room has and their purpose of use.

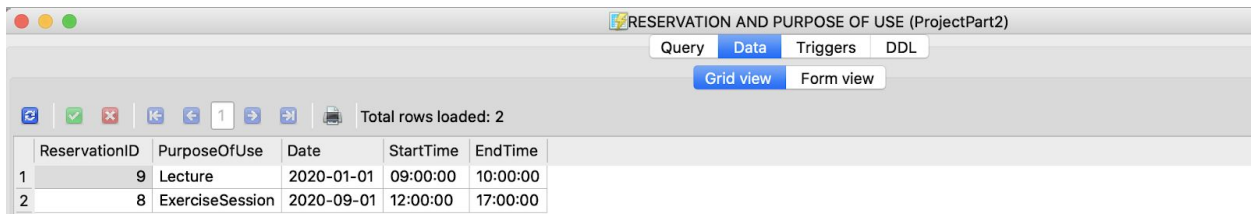
The query should return: ReservationID, Purpose, Date, StartTime, Endtime

Here is an example with the following value

: RoomID = 6 */

```
DROP VIEW IF EXISTS [RESERVATION AND PURPOSE OF USE];
CREATE VIEW[RESERVATION AND PURPOSE OF USE] AS
SELECT Event.ReservationID, Type AS PurposeOfUse, Reservation.Date, StartTime, EndTime
FROM Reservation LEFT JOIN Event
ON Reservation.ReservationID = Event.ReservationID
WHERE Reservation.RoomID = 6;
```

Result from the stored View:



	ReservationID	PurposeOfUse	Date	StartTime	EndTime
1	9	Lecture	2020-01-01	09:00:00	10:00:00
2	8	ExerciseSession	2020-09-01	12:00:00	17:00:00

5/ List all students who have enrolled for the certain exercise group or exam.

/ 13. List all students who have enrolled for the certain exercise group or exam.*

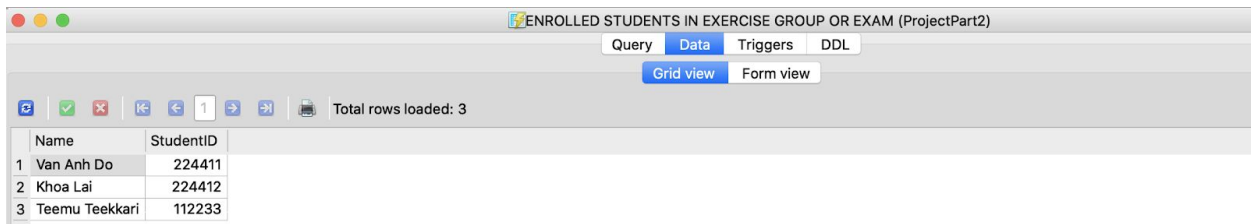
The query should return: StudentName, StudentID

Here is an example query with following values

: ExerciseGroupID = 4, ExamID = 2 */

```
DROP VIEW IF EXISTS [ENROLLED STUDENTS IN EXERCISE GROUP OR EXAM];
CREATE VIEW[ENROLLED STUDENTS IN EXERCISE GROUP OR EXAM] AS
SELECT DISTINCT Name, StudentID
FROM Student
WHERE StudentID IN
    (SELECT StudentID
     FROM ExerciseGroupEnrollment
     WHERE ExerciseGroupID = 4
    )
OR StudentID IN
    (SELECT StudentID
     FROM ExamEnrollment
     WHERE ExamID = 2
    );
```

Result from the stored View:



	Name	StudentID
1	Van Anh Do	224411
2	Khoa Lai	224412
3	Teemu Teekkari	112233