Lab 1: ER diagrams and the relational model

Task 0: Preliminaries
Task 1: Constructing an ER diagram
Task 2: Interpreting and transforming an ER diagram

Task 0: Preliminaries

- ***If you have any questions about this lab or anything else regarding the course, you can post them on Piazza, our course's Q&A site.*** Use the link in the left-hand navigation bar to access Piazza.

  Please take advantage of Piazza's **search feature**. Before posting a question, you should use this feature to see if someone else has already asked a similar question!

- The labs consist of practice exercises that are designed to reinforce the key concepts from lecture and to prepare you for the problem sets.
- Feel free to ask questions at any point during the lab.
- We may not finish all of the lab exercises during the actual lab session. Every Wednesday, we will update the current week's lab on the Labs page of the course website to include the solutions.
- We don't grade the lab exercises. ***However, starting with next week's lab, your attendance at lab will count towards your participation grade.*** If you attend 85% of the lab sessions over the course of the semester, you will get full credit for lab participation. See the syllabus for more details.
- To get full credit for participation in lab, you must work productively throughout the lab session. You will not be penalized if you cannot finish all of the lab exercises.
- Feel free to work on the lab exercises with your classmates, and don't hesitate to ask a staff member for help!

Task 1: Constructing an ER diagram

In this example, we'll consider a database of movie information. Suppose we want to allow users to get information about:

- movies
- actors
- directors
- Academy Awards

We'll assume that each movie has a single director, and we will only concern ourselves with the following awards: Best Picture, Best Actor, Best Actress, Best Supporting Actor, and Best Supporting Actress.

Let's construct an ER diagram for this domain. While there are many possible answers for this problem, you should try to include at least one example of each of the following:

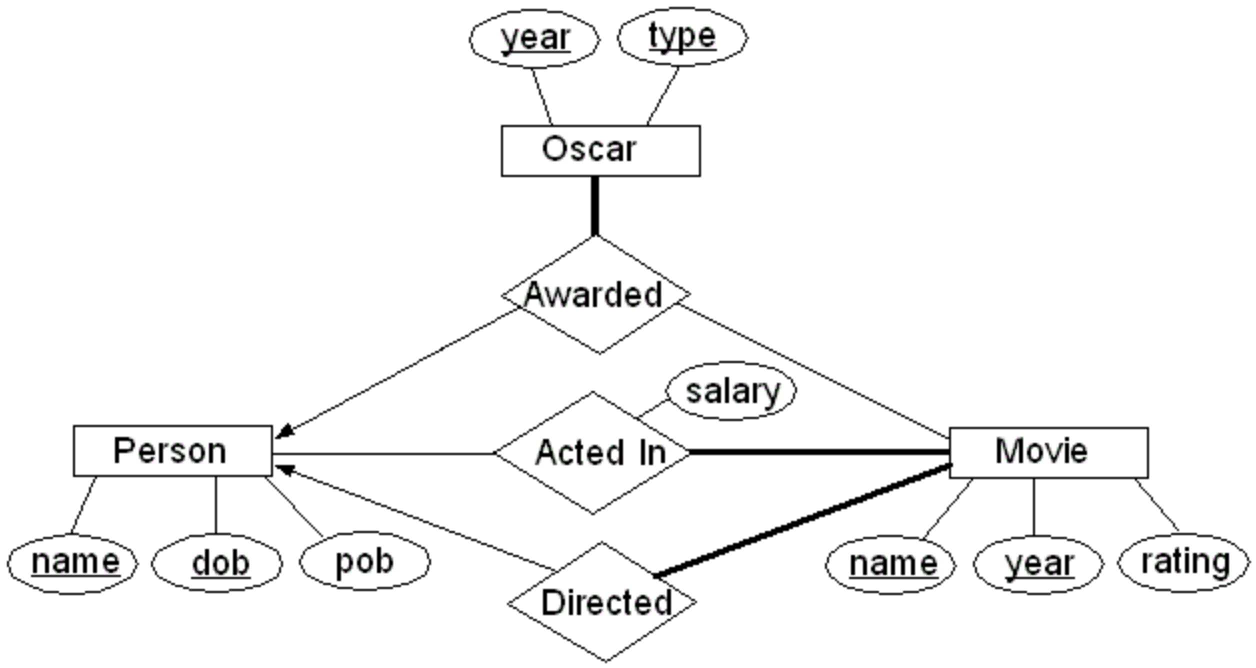
- a relationship set with one or more descriptive attributes

- a key constraint
- a participation constraint
- a ternary (three-way) relationship set

In addition to constructing the ER diagram, specify the primary-key attributes of all entity sets by underlining the attributes on the diagram.

Solution

The following is one possible solution:



Here are some points worth noting about this diagram:

1. The diagram uses a single entity set, *Person*, for both actors and directors. The advantage to using a single set for both is that some people may be both actors and directors. If we used two separate entity sets, we would end up storing information about some people twice—which wastes space and makes it possible for inconsistencies to arise in the database.

2. The *Directed* and *Acted In* relationship sets are binary relationship sets—they each store associations between **two** entities. *Awarded* is a ternary relationship set, because it involves associations between **three** entities: *Oscar* entities, *Person* entities, and *Movie* entities, such as the fact that Tom Hanks won the Best Actor award in 1994 for his performance in “Philadelphia”.

3. The *Awarded* relationship is many-to-one, because a given Oscar statue is awarded to one person. (We are assuming that the only Oscars we care about are the four major acting awards, all of which are won by one person, and the Best Picture award, which for our purposes is won/received by the director.) We show this many-to-one relationship by using an arrow from *Awarded* to *Person*. Strictly speaking, this indicates that a given Oscar-movie combination is associated with at most one person; while this is more than we really need to say, it’s not incorrect.

4. We also have an arrow from *Directed* to *Person*, because we’re assuming that there is a single director for each movie, and thus *Directed* is a function that maps a given movie to its director.

5. A given entity set can participate either totally or partially in a given relationship set. In this case, *Oscar* has total participation in *Awarded*, because each entity in *Oscar* has a corresponding relationship in *Awarded*. Similarly, *Movie* participates totally in both *Acted In* and *Directed* if we assume that we always store the director and at least one actor for each movie in the database. We indicate total participation by using either a thick line (as shown here) or a double line. Thin/single lines indicate partial participation. For example, *Person* participates only partially in *Acted In*, because only some people in the database are actors.

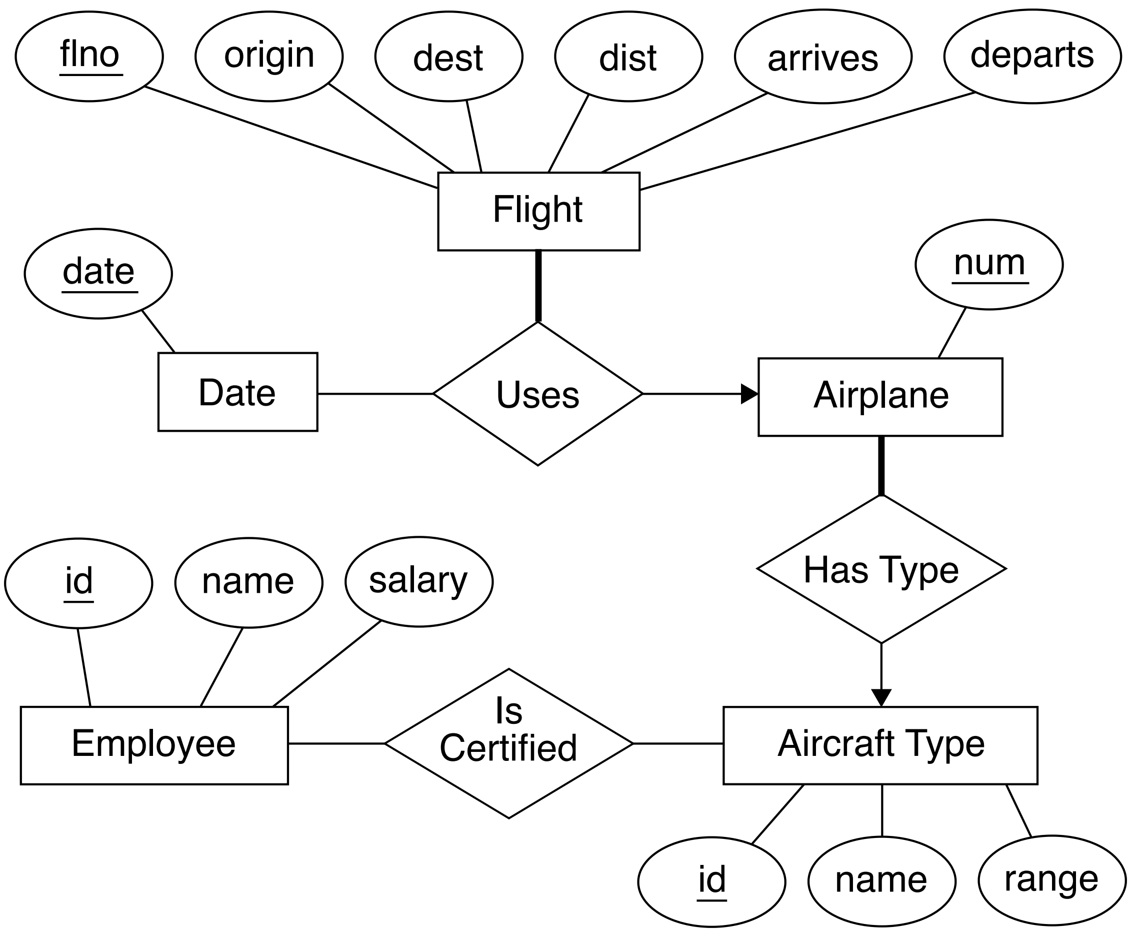
6. The diagram shows a descriptive attribute for the *Acted In* relationship set. This salary attribute is associated with the relationship set because an actor earns a separate salary for each movie in which he or she acts.

7. The attributes that constitute the primary keys for the *Oscar*, *Person*, and *Movie* entity sets are underlined. Note that we need multiple attributes for each primary key (e.g., name and year for *Movie*). If we were to implement this database, we might well associate a unique ID with each entity in a given entity set, and use the IDs as the primary keys instead.

Task 2: Interpreting and transforming an ER diagram

Now let’s switch to a different domain: airline information—more specifically, information about flights, aircraft, and employees.

The following is an ER diagram for this domain:



Here are some facts about the diagram:

- The *Employees* entity set includes both pilots and other kinds of employees.
- The *Is Certified* relationship set specifies the types of aircraft that a pilot is certified to fly.
- The *Uses* relationship set specifies the airplane used by a particular flight on a particular date.
- The *Has Type* relationship set specifies a particular airplane's aircraft type.

Answer the following questions:

1. Why is there an arrow from *Has Type* to *Aircraft Type*?

> **Solution**
>
> Because a given aircraft has at most one aircraft type. Thus, *Has Type* is a many-to-one relationship set from *Airplane* to *Aircraft Type*.

2. Why isn't there an arrow from *Is Certified* to *Aircraft Type*?

> **Solution**

> Because a given employee may be certified to fly more than one type of aircraft. Thus, *Is Certified* is a many-to-many relationship set.

3. Why is there a thick line from *Airplane* to *Aircraft Type*?

> **Solution**
>
> Because every aircraft has at least one aircraft type. Thus, *Airplane* has total participation in *Aircraft Type*.

4. Why isn't there a thick line from *Employee* to *Is Certified*?

> **Solution**
>
> Because not every employee is certified to fly. Thus, *Employee* has partial participation in *Is Certified*.

5. In transforming this diagram to a relational schema, would we need a separate relation to represent the *Has Type* relationship set? Why or why not?

> **Solution**
>
> No. Because *Has Type* is many-to-one from *Airplane* to *Aircraft Type*, we can combine the separate relations that would ordinarily be formed for *Airplane* and *Has Type* into a single relation:
>
> *Airplane*(*num*, *type_id*)

Now, transform this diagram to a relational schema. Specify the name and attribute names of each relation in the schema, underlining the primary-key attributes of each relation.

> **Solution**
>
> We will underline the primary-key attributes below.
> *Flight*(<u>*flno*</u>, *origin*, *dest*, *dist*, *departs*, *arrives*)
> *Airplane*(<u>*num*</u>, *type*)
> *AircraftType*(<u>*id*</u>, *name*, *range*)
> *Employee*(<u>*id*</u>, *name*, *salary*)
> *Uses*(<u>*flno*</u>, <u>*date*</u>, *airplane_num*)
> *IsCertified*(<u>*employee_id*</u>, <u>*aircraft_type_id*</u>)
>
> **Note:** We don't need a separate relation for the `Date` entity set because the only attribute of `Date` is the date itself! As a result, we can fully capture the information from that entity set in the relation that we create for the `Uses` relationship set. Another way to think of it is that we never need to access the date information on its own. It is only ever needed in conjuction with `Uses`.
>
> By contrast, note that the other entity sets (e.g., `Flight`) have a number of additional attributes besides the ones that are relevant to their associated relationship sets. As a result, we need a separate relation for each

of those entity sets.

Finally, specify any foreign keys that are present in your relations along with the associated referential-integrity constraints.

Solution

Here are the foreign keys and their associated constraints:

- *type* in *Airplane*; each value of that attribute must match a value of the *id* attribute from the *AircraftType* relation.
- *flno* in *Uses*; each value of that attribute must match a value of the *flno* attribute from the *Flight* relation.
- *airplane_num* in *Uses*; each value of that attribute must match a value of the *num* attribute from the *Airplane* relation.
- *employee_id* in *IsCertified*; each value of that attribute must match a value of the *id* attribute from the *Employee* relation.
- *aircraft_type_id* in *IsCertified*; each value of that attribute must match a value of the *id* attribute from the *AircraftType* relation.

Lab 2: SQL and relational algebra

Task 0: Preliminaries
- Policy reminders
- Create the necessary folder
- Download the software and database

Task 1: Explore DB Browser for SQLite

Task 2: Understand and write SQL queries

Task 3: Understand and write relational-algebra queries

Task 0: Preliminaries

Policy reminders

- Feel free to ask questions at any point during the lab.
- We may not finish all of the lab exercises during the actual lab session. Every Wednesday, we will post solutions to the current week's lab on the Labs page of the course website.
- We don't grade the lab exercises. ***However, starting with today's lab, your attendance at lab will count towards your participation grade.*** If you attend 85% of the lab sessions over the course of the semester, you will get full credit for lab participation. See the syllabus for more details.
- To get full credit for participation in lab, you must work productively throughout the lab session. You will not be penalized if you cannot finish all of the lab exercises.
- Feel free to work on the lab exercises with your classmates, and don't hesitate to ask a staff member for help!

Create the necessary folder

1. If you haven't already created a folder named `cs460` for your work in this course, follow these instructions to do so.
2. Then create a subfolder called `lab2` within your `cs460` folder, and put all of the files for this lab in that folder.

Download the software and database

1. If you haven't already done so, download and install DB Browser for SQLite following the instructions in Problem Set 1.
2. To download the database we will be using in this lab, click on the following link:

   `university.sqlite`

   and save the file in your `lab2` folder.

Using the virtual desktop

If you encounter issues installing the DB Browser software, you can use the copy of it that is available on the virtual desktop. Instructions for doing so can be found here.

Task 1: Explore DB Browser for SQLite

1. Launch DB Browser for SQLite.
2. Click the *Open Database* button, and find and open the `university.sqlite` database file that you downloaded above.
3. Click on the *Browse Data* tab and explore the contents of the tables. Initially, you will see the *Course* table, but you should explore the other tables as well by choosing the appropriate table name from the drop-down menu.

   Note that the tables have the same column names as the university database from the lecture notes, but the actual rows are different. In particular:

   - We have included some additional data to allow for more interesting queries and results.
   - We use the full names of the departments in the *Department* and *MajorsIn* tables. In the lecture notes, the department names are abbreviated.
4. Select the *Execute SQL* tab, which is where you will perform queries on the database.

   For example, try entering the following SQL command in the space provided:

   ```
   SELECT name, credit_status
   FROM Student, Enrolled
   WHERE id = student_id
     AND course_name = 'CS 460';
   ```

   To run it, you can do any of the following:

   - Press F5
   - Press Ctrl-R
   - Click the small button with a triangular shape that looks like the Play button of a music player.
5. What results do you see? What do they represent?

   > **Solution**
   >
   > The query produces a table with six rows and two columns. They represent the names of all students in the database who are enrolled in CS 460 and the credit status for which they are enrolled.

6. Edit the previous query to remove the first condition in the WHERE clause:

   ```
   SELECT name, credit_status
   FROM Student, Enrolled
   WHERE course_name = 'CS 460';
   ```

   How does removing that condition affect the results?

> **Solution**
>
> The first condition in the original query was a join condition. When you remove it, you end up including combinations of tuples from Student and Enrolled that don't make sense.
>
> *Every* student from the Student table is paired with all six of the tuples from Enrolled in which course_name is `'CS 460'`, and thus every student shows up six times in the results.

7. Enter and run the following query:

   ```
   SELECT name
   FROM Course
   WHERE room_id IS NULL;
   ```

   What does it do?

   > **Solution**
   >
   > The query produces the names of all courses from the Course table with a room_id value of `NULL` – i.e., all courses without a room.

8. Does the following query do the same thing as the previous one?

   ```
   SELECT name
   FROM Course
   WHERE room_id = NULL;
   ```

   Why or why not?

   > **Solution**
   >
   > No. This query produces no results because when you attempt to check for `NULL` with a standard comparison operator like `=` or `!=`, the result is always false. Instead, you need to use `IS NULL` or `IS NOT NULL`.

Task 2: Understand and write SQL queries

Download the following file:

lab2_queries.py

and put it in your `lab2` folder.

Open `lab2_queries.py` in a Python IDE or a text editor.

Important

- If you choose to use a text editor, instead of double-clicking on the downloaded file, you should launch your text editor and use its *File->Open* menu option to open the file. You may need to change the type of file that the text editor is looking for to *All Files*.

- **The file that you create must be a plain-text file.** More information about what this means is available here.

In this task, you will continue to work with the university database from the previous task. Here is its schema:

Schema of the database

Student(*id*, name)
Department(*name*, office)
Room(*id*, name, capacity)
Course(*name*, start_time, end_time, room_id)
MajorsIn(*student_id*, *dept_name*)
Enrolled(*student_id*, *course_name*, credit_status)

When constructing each query, we recommend that you follow the rules of thumb given in the lecture notes.

Feel free to work with another student on these problems, and to ask the TA for help as needed.

1. Write a query that finds the names and capacities of all rooms in the CAS and CGS buildings.

   Test your query, revise it as needed, and copy the final version into your file, putting it between the triple quotes for `query1`.

   Solution

   ```
   SELECT name, capacity
   FROM Room
   WHERE name LIKE 'CAS%'
      OR name LIKE 'CGS%';
   ```

2. Write a query that finds the number of students who are enrolled in a course for undergraduate (`'ugrad'`) credit.

   Test your query. **The correct result is 6. If you don't get this result, revise it as needed.** Copy the final version into your file, putting it between the triple quotes for `query2`.

   Solution

   ```
   SELECT COUNT(DISTINCT student_id)
   FROM Enrolled
   WHERE credit_status = 'ugrad';
   ```

3. Write a query that finds the earliest (i.e., smallest) start time of any CS course.

   Test your query, revise it as needed, and copy the final version into your file, putting it between the triple quotes for `query3`.

   > **Solution**
   >
   > ```
   > SELECT MIN(start_time)
   > FROM Course
   > WHERE name LIKE 'CS%';
   > ```

4. Modify your previous query to find the **name and start time** of the CS course with the earliest start time. ***In order to adhere to standard SQL, you will need to use a subquery.***

   Important: You should ***not*** do something like the following:

   ```
   SELECT name, MIN(start_time)
   ...
   ```

   Although this works in SQLite, it is ***not*** allowed in standard SQL, because the SELECT clause combines a regular column (`name`) with an aggregate function (`MIN(start_time)`).

   Test your query, revise it as needed, and copy the final version into your file, putting it between the triple quotes for `query4`.

   > **Solution**
   >
   > ```
   > SELECT name, start_time
   > FROM Course
   > WHERE name LIKE 'CS%';
   >   AND start_time = (SELECT MIN(start_time)
   >                     FROM Course
   >                     WHERE name LIKE 'CS%');
   > ```

5. Write a query that finds, for each type of credit status, the number of students who are enrolled in a course with that credit status.

   Test your query, revise it as needed, and copy the final version into your file, putting it between the triple quotes for `query5`.

   > **Solution**
   >
   > ```
   > SELECT credit_status, COUNT(DISTINCT student_id)
   > FROM Enrolled
   > GROUP BY credit_status;
   > ```

Note: For this query, you *are* allowed to combine a regular column with an aggregate function. Why is it allowed in this case?

> **Solution**
>
> Because we were grouping on the column that was being combined with the aggregate function.

6. Write a query that finds the names of all students majoring in `'computer science'`. *Hint:* Don't forget to include an appropriate join condition.

   Test your query, revise it as needed, and copy the final version into your file, putting it between the triple quotes for `query6`.

> **Solution**
>
> ```
> SELECT name
> FROM Student, MajorsIn
> WHERE id = student_id
>   AND dept_name = 'computer science';
> ```

7. Write a query that finds the names of all students who are ***not*** enrolled in CS 460. *Hint:* You will need a subquery.

   Test your query, revise it as needed, and copy the final version into your file, putting it between the triple quotes for `query7`.

> **Solution**
>
> ```
> SELECT name
> FROM Student
> WHERE id NOT IN (SELECT student_id
>                  FROM Enrolled
>                  WHERE course_name = 'CS 460');
> ```

Task 3: Understand and write relational-algebra queries

On paper, write a relational-algebra query for each of the following problems from Task 2:

1. problem 1 (names and capacities of all rooms in CAS and CGS)
2. problem 6 (names of students majoring in computer science)
3. problem 7 (names of students *not* enrolled in CS 460). *Hint:* Take advantage of the set difference operator (-). Also, in order to make this query easier to write, you may assume that all student names are unique.

> **Solution**

> The solutions to these problems can be found here.

Lab 3: SQL revisited

Task 0: Prepare for lab
Task 1: Create a table and insert rows
Task 2: Practice SQL queries

Task 0: Prepare for lab

1. In last week's lab, you should have downloaded:

   - DB Browser for SQLite
   - the university database

   If you didn't, you should go ahead and do so now.

2. Launch DB Browser for SQLite.

3. Click the *Open Database* button, and find and open the university database, which you should have saved in your `lab2` folder in a file named `university.sqlite`.

Task 1: Create a table and insert rows

1. Click on the *Database Structure* tab. You should see the list of tables that are currently in the database, along with the `CREATE TABLE` command for each table.

2. Some students have work-study jobs in which they work for a department at the university. Let's assume that we want to create a table called `WorksFor` to capture the relationships between students and the departments for which they work. Each row in this new table should include:

   - the id number of a student
   - the name of a department
   - the title of the job that the student holds for that department

   Construct a `CREATE TABLE` command for this new table. Choose appropriate column names and types, and make sure to specify the primary key of the table and any foreign keys.

   You may assume that:

   - Students have at most one job for a department, and thus a given student will appear at most once in this table.
   - Job titles will have lengths that vary widely, but they will be no more than 30 characters in length.

   Use the `CREATE TABLE` commands for the existing tables as models. Among other things, you should consult them to ensure that your foreign key(s) have data types that are consistent with the types of the columns to which they refer.

> **Solution**
>
> ```
> CREATE TABLE WorksFor(student_id CHAR(8) PRIMARY KEY,
>     dept_name VARCHAR(20), job_title VARCHAR(30),
>     FOREIGN KEY (student_id) REFERENCES Student(id),
>     FOREIGN KEY (dept_name) REFERENCES Department(name));
> ```
>
> Note: We only need student_id for the primary key, because a given student appears at most once in this table.

3. Try entering your new `CREATE TABLE` command in the *Execute SQL* tab of DB Browser. Remember that you can run a SQL command by doing any of the following:

   - Press F5
   - Press Ctrl-R
   - Click the small button with a triangular shape that looks like the Play button of a music player.

   At the bottom of the DB Browser window, you should see a message indicating that the command executed successfully. If you don’t, make whatever fixes are needed, and try again. Feel free to ask for help as needed!

4. Construct the `INSERT` commands needed to add **two rows** to this table. You may use whatever values you want, as long as you observe the uniqueness and referential-integrity constraints on the table. You may find it helpful to consult the contents of the other tables, which you can do by using the *Browse Data* tab in DB Browser.

   > **Solution**
   >
   > Here are two possible commands:
   >
   > ```
   > INSERT INTO WorksFor VALUES ('U00000001', 'computer science', 'assistant');
   > INSERT INTO WorksFor VALUES ('U00000010', 'mathematics', 'grader');
   > ```

5. Try entering your `INSERT` commands in the *Execute SQL* tab, and make corrections as needed until they execute successfully.

6. Return to the *Browse Data* tab, and look at the contents of the `WorksFor` table, which should include the rows you just added.

Once you are done with Task 1, you should show the results of your commands to the TA.

Task 2: Practice SQL queries

Download the following file:

lab3_queries.py

and put it in your folder for this lab.

Open `lab3_queries.py` in a Python IDE or a text editor.

Important

- If you choose to use a text editor, instead of double-clicking on the downloaded file, you should launch your text editor and use its *File->Open* menu option to open the file. You may need to change the type of file that the text editor is looking for to *All Files*.
- **The file that you create must be a plain-text file.** More information about what this means is available here.

In this task, you will continue to work with the university database. Here is its schema:

Schema of the database

Student(*id*, name)
Department(*name*, office)
Room(*id*, name, capacity)
Course(*name*, start_time, end_time, room_id)
MajorsIn(*student_id*, *dept_name*)
Enrolled(*student_id*, *course_name*, credit_status)

When constructing each query, we recommend that you follow the rules of thumb given in the lecture notes.

Feel free to work with another student on these problems, and to ask the TA for help as needed.

1. Write a query that finds the major(s) of each student in the database. The results of your query should be tuples of the form (student name, dept name).

   Test the revised SQL command in DB Browser to make sure that it works. ***Once you have finalized the command, copy the command into your `lab3_queries.py` file, putting it between the triple quotes provided for the variable query1.*** We have included a sample query to show you what the format of your answers should look like.

   Solution

   ```
   SELECT name, dept_name
   FROM Student, MajorsIn
   WHERE id = student_id;
   ```

2. Now revise the previous query so that its results will include students who don't have a major. *Hint:* You will need to perform a `LEFT OUTER JOIN`, with the join condition moved to the `ON` clause.

   Test your query, revise it as needed, and copy the final version into your file, putting it between the triple quotes for `query2`.

   Solution

```
SELECT name, dept_name
FROM Student LEFT OUTER JOIN MajorsIn
  ON id = student_id;
```

3. Let's say that we want to rewrite the previous query so that its results are limited to students whose names begin with either `C` or `S`. Should the new condition(s) go in the `WHERE` clause or the `ON` clause? Why?

   > **Solution**
   >
   > The new conditions should go in the WHERE clause, because they are *not* a join condition that is being used by the LEFT OUTER JOIN. If we put the new conditions in the ON clause, we would also get tuples for students whose names do not begin with C or S, because the LEFT OUTER JOIN includes results for tuples from the left table (Student) that do not have a match based on the condition in the ON clause.

   Go ahead and revise the query, and put the final version of the query between the triple quotes provided for `query3`.

   > **Solution**

   ```
   SELECT name, dept_name
   FROM Student LEFT OUTER JOIN MajorsIn
     ON id = student_id
   WHERE name LIKE 'C%' OR name LIKE 'S%';
   ```

4. Write a query that determines the number of students majoring in each department. The result should be tuples of the form (department name, number of students majoring), and the departments should be listed in decreasing order based on how many students the department has. This query does **not** need to include departments with no majoring students.

   Put your final query between the triple quotes provided for `query4`.

   > **Solution**

   ```
   SELECT name, COUNT(*)
   FROM Department, MajorsIn
   WHERE name = dept_name
   GROUP BY name
   ORDER BY COUNT(*) DESC;
   ```

5. Now rewrite the previous query so that it includes departments with no majoring students.

   Hints:

   - You will need a `LEFT OUTER JOIN`.
   - You may need to revise the aggregate function in the `SELECT` clause. Make sure that you get a count of 0 for any department that did not appear in your results for the previous query.

Put your final query between the triple quotes provided for `query5`.

> **Solution**
>
> ```
> SELECT name, COUNT(student_id)
> FROM Department LEFT OUTER JOIN MajorsIn
>   ON name = dept_name
> GROUP BY name
> ORDER BY COUNT(student_id) DESC;
> ```
>
> Note: We must use COUNT(student_id) or COUNT(dept_name), so that the department with no match in MajorsIn (basket weaving) will end up with a count of 0, rather than a count of 1.

6. Rewrite the previous query so that it only includes departments with fewer than two students majoring in them.

   Put your final query between the triple quotes provided for `query6`.

> **Solution**
>
> ```
> SELECT name, COUNT(student_id)
> FROM Department LEFT OUTER JOIN MajorsIn
>   ON name = dept_name
> GROUP BY name
> HAVING COUNT(student_id) < 2
> ORDER BY COUNT(student_id) DESC;
> ```

7. Write a query to find the name and capacity of the room with the smallest capacity. *Hint:* You will need a subquery.

   Put your final query between the triple quotes provided for `query7`.

> **Solution**
>
> ```
> SELECT name, capacity
> FROM Room
> WHERE capacity = (SELECT MIN(capacity)
>                   FROM Room);
> ```

8. Write a query that finds the name and the capacity of the room that CS 460 is taught in. *Hint:* You will need to prepend a table name or a table alias before the names of at least some of the columns in your query.

   Put your final query between the triple quotes provided for `query8`.

> **Solution**

```
SELECT R.name, capacity
FROM Room R, Course C
WHERE R.id = C.room_id
  AND C.name = 'CS 460';
```

9. Write a query that finds the names of all departments whose names consist of more than one word.

   Put your final query between the triple quotes provided for `query9`.

   > **Solution**
   >
   > ```
   > SELECT name
   > FROM Department
   > WHERE name LIKE '% %';
   > ```

10. Write a query that finds the names and capacities of all rooms that have a capacity between 50 and 250 inclusive ***or*** that have a name that does ***not*** begin with the letter ‘C’.

    Put your final query between the triple quotes provided for `query10`.

    > **Solution**
    >
    > ```
    > SELECT name, capacity
    > FROM Room
    > WHERE (capacity >= 50 AND capacity <= 250)
    >    OR name NOT LIKE 'C%';
    > ```

11. Write a query that counts the number of students majoring in computer science. *Note:* A `GROUP BY` clause is **not** needed here. Why not?

    > **Solution**
    >
    > A GROUP BY clause is not needed, because we are only computing a single count, rather than separate counts for different subgroups.

    Put your final query between the triple quotes provided for `query11`.

    > **Solution**
    >
    > ```
    > SELECT COUNT(*)
    > FROM MajorsIn
    > WHERE dept_name = 'computer science';
    > ```

12. Write a query that finds the names of all students taking a course for undergraduate credit, along with the names and start times of those courses, and the names of the rooms in which the courses are offered. The result of the query should be tuples of the form (student name, course name, start_time, room_name).

    Put your final query between the triple quotes provided for `query12`.

    Solution

    ```
    SELECT S.name, C.name, C.start_time, R.name
    FROM Student S, Course C, Enrolled E, Room R
    WHERE S.id = E.student_id
      AND C.name = E.course_name
      AND C.room_id = R.id
      AND E.credit_status = 'ugrad';
    ```

Lab 4: Record formats; index structures

Fixed-length and variable-length records
B-trees and B+trees
Linear dynamic hashing

Fixed-length and variable-length records

Imagine that we are working with the `Room` table from the university database that we used in Lab 2 and Lab 3. That table has the following schema:

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

Consider the following tuple from that table:

```
('0123', 'PHO 206', 199)
```

1. If we use the fixed-length record format discussed in lecture, what would the record look like for this tuple?

> **Solution**
>
> | | | |
> |---|---|---|
> | 0123 | PHO 206#------------ | 199 |
>
> Note that we use a special delimiter (#) to indicate the end of the `VARCHAR` value and a hyphen (-) to indicate a "wasted" byte (i.e., a byte that is part of the record but isn't storing useful data or metadata).
>
> With the exception of the delimiter, no per-record metadata is needed, because each field has the same length in every record (including the `VARCHAR`, which is given its maximum length).

2. What is the length in bytes of this record if we assume that:
   - characters are one byte each
   - integer data values are four bytes each.

> **Solution**
>
> 28 bytes – the same length that *all* records for this table would have, since they are fixed-length! `CHAR` and `INT` fields always have the same length, and fixed-length records give `VARCHAR` fields their maximum length.
>
> ```
>    len(id) + max_len(name) + len(capacity)
>  =    4    +      20       +       4       = 28
> ```

3. If we use the second type of variable-length record discussed in lecture – in which each field is preceded by its length – what would the record for the above tuple look like?

   Solution

4	`0123`	7	`PHO 206`	4	`199`

4. What is the length in bytes of the record from part 3? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.

   Solution

   21 bytes. We only allocate the bytes needed for the actual `VARCHAR` value, but there are now three per-record metadata values (the lengths), each of which is 2 bytes.

   ```
     len(id) + len('PHO 206') + len(capacity) + metadata
   =    4    +        7       +       4       +   3*2    = 21
   ```

5. Now assume that we are using the third type of variable-length record discussed in lecture, in which each record begins with a header of field offsets. What will the record look like for the above tuple? Use the same assumptions about the sizes of characters, integer data values, and integer metadata that we made above.

   Solution

0	2	4	6	8	12	19
8	12	19	23	`0123`	`PHO 206`	`199`

   Note that we include the offset of the end of the record, since we may need that value to compute the length of one of the fields.

6. What is the length in bytes of the record from part 5?

   Solution

   23 bytes. We only allocate the bytes needed for the actual `VARCHAR` value, but we need to include four per-record metadata values (the offsets), each of which is 2 bytes.

   ```
     len(id) + len('PHO 206') + len(capacity) + metadata
   =    4    +        7       +       4       +   4*2    = 23
   ```

7. Now imagine that the room didn't have a name and we used a value of `NULL` to indicate this:

   ```
   ('0123', NULL, 199)
   ```

   How would the answer to part 5 change?

Solution

0	2	4	6	8	12
8	-1	12	16	0123	199

We use a special offset of for the `NULL` value. It is the second offset, because the `NULL` value is the value of the second field.

B-trees and B+trees

1. Insert the following sequence of keys into an initially empty B-tree of order 2:

```
51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14
```

Solution

The sequence of insertions is as follows. We indicate inserted values using an asterisk.

```
-------
| *51 |
-------

-----------
| *3 | 51 |
-----------

-----------------
|  3 | *10 | 51 |
-----------------

----------------------
|  3 | 10 | 51 | *77 |
----------------------

           -------
           | *20 |
           +-----+
          /       \
-----------       -----------
|  3 | 10 |       | 51 | 77 |
-----------       -----------

            ------
            | 20 |
           _+----+_
          /        \
-----------          -----------------
|  3 | 10 |          | *40 | 51 | 77 |
-----------          -----------------

            ------
            | 20 |
           _+----+_
          /        \
-----------        ----------------------
|  3 | 10 |        | *34 | 40 | 51 | 77 |
-----------        ----------------------

              -----------
              | 20 | 40 |
            __+----+----+___
           /       |        \
-----------   ------------   -----------
|  3 | 10 |   | *28 | 34 |   | 51 | 77 |
-----------   ------------   -----------

              -----------
              | 20 | 40 |
            __+----+----+___
           /       |        \
-----------   ------------   -----------------
|  3 | 10 |   | 28 | 34 |    | 51 | *61 | 77 |
-----------   ------------   -----------------

              -----------
              | 20 | 40 |
            __+----+----+__
           /       |       \
-----------   -----------   ----------------------
|  3 | 10 |   | 28 | 34 |   | 51 | 61 | 77 | *80 |
-----------   -----------   ----------------------

                   -----------------
                   | 20 | 40 | *68 |
                  _+----+----+-----+_
           ______/  ___/      \__    \____
          /        /             \        \
-----------   -----------   -----------   -----------
|  3 | 10 |   | 28 | 34 |   | 51 | 61 |   | 77 | 80 |
-----------   -----------   -----------   -----------

                   ----------------
                   | 20 | 40 | 68 |
                  _+----+----+----+_
           ______/  ___/      \__   \_____
          /        /             \        \
-----------   -----------   -----------   -----------------
|  3 | 10 |   | 28 | 34 |   | 51 | 61 |   | 77 | 80 | *93 |
-----------   -----------   -----------   -----------------

                          ----------------
                          | 20 | 40 | 68 |
                  ________+----+----+----+
           ______/  __________/   _/      \____
          /        /             /             \
-----------   -----------   -----------   ----------------------
|  3 | 10 |   | 28 | 34 |   | 51 | 61 |   | 77 | 80 | *90 | 93 |
-----------   -----------   -----------   ----------------------
```

```
                         --------------------
                         | 20 | 40 | 68 | 90 |
                 ________+----+----+----+----+____
          ______/   __________/   _/      \____    \_________
         /        /             /             \             \
-----------   -----------   -----------   -----------   ------------
|  3 | 10 |   | 28 | 34 |   | 51 | 61 |   | 77 | 80 |   | 93 | *97 |
-----------   -----------   -----------   -----------   ------------

                         ---------------------
                         | 20 | 40 | 68 | 90 |
                        _+----+----+----+----+_________
                 _______/     /      \__  \__________   \____________
                /            /          \            \               \
----------------      -----------    -----------    -----------    -----------
| 3 | 10 | *14 |      | 28 | 34 |    | 51 | 61 |    | 77 | 80 |    | 93 | 97 |
----------------      -----------    -----------    -----------    -----------
```

2. Insert the same sequence of keys into an initially empty B+tree of order 2.

   a. Before performing the insertions, do you expect the result to be the same or different? Why?

   > **Solution**
   >
   > Different. Because data items must be only in the leaf nodes, the rules for splitting a leaf node change somewhat. The key of the middle item is copied up into the parent so that it appears in both the parent and the new leaf node created as a result of the split. In addition, the leaf nodes will be linked together.

   b. Perform the sequence of insertions and draw the B+tree as it is formed.

   > **Solution**
   >
   > We indicate inserted values using a prefixed asterisk.

```
-------
| *51 |
-------

-----------
| *3 | 51 |
-----------

-----------------
|  3 | *10 | 51 |
-----------------

----------------------
|  3 | 10 | 51 | *77 |
----------------------

              -------
```

```
           | *20 |
           +-----+
          /       \
-----------      -----------------
|  3 | 10 |------| *20 | 51 | 77 |
-----------      -----------------


           ------
           | 20 |
           +----+
          /      \
-----------      ----------------------
|  3 | 10 |------| 20 | *40 | 51 | 77 |
-----------      ----------------------


            -----------
            | 20 | 40 |
           _+----+----+_______
          /       \           \
-----------   ------------    ----------------
|  3 | 10 |---| 20 | *34 |----| 40 | 51 | 77 |
-----------   ------------    ----------------


                -----------
                | 20 | 40 |
           _____+----+----+________
          /          |             \
-----------   -----------------    ----------------
|  3 | 10 |---| 20 | *28 | 34 |----| 40 | 51 | 77 |
-----------   -----------------    ----------------


                -----------
                | 20 | 40 |
           _____+----+----+________
          /          |             \
-----------   ----------------    ----------------------
|  3 | 10 |---| 20 | 28 | 34 |----| 40 | 51 | *61 | 77 |
-----------   ----------------    ----------------------


                       ----------------
                       | 20 | 40 | 61 |
           ____________+----+----+----+_________
          /                 |     \             \
-----------   ----------------    -----------   -----------------
|  3 | 10 |---| 20 | 28 | 34 |----| 40 | 51 |---| 61 | 77 | *80 |
-----------   ----------------    -----------   -----------------


                       ----------------
                       | 20 | 40 | 61 |
           ____________+----+----+----+_________
          /                 |     \             \
-----------   ----------------    -----------   ----------------------
|  3 | 10 |---| 20 | 28 | 34 |----| 40 | 51 |---| 61 | *68 | 77 | 80 |
-----------   ----------------    -----------   ----------------------


                       ---------------------
                       | 20 | 40 | 61 | 77 |
                 ______+----+----+----+----+__________
          ______/           |     \    \_________     \_________
         /                 /       \             \              \
-----------   ----------------    -----------   -----------   -----------------
|  3 | 10 |---| 20 | 28 | 34 |----| 40 | 51 |---| 61 | 68 |---| 77 | 80 | *93 |
-----------   ----------------    -----------   -----------   -----------------

                       ---------------------
                       | 20 | 40 | 61 | 77 |
                 ______+----+----+----+----+__________
          ______/           |     \    \_________     \_________
         /                 /       \             \              \
-----------   ----------------    -----------   -----------   -----------------
-----
|  3 | 10 |---| 20 | 28 | 34 |----| 40 | 51 |---| 61 | 68 |---| 77 | 80 | *90 |
93 |
-----------   ----------------    -----------   -----------   -----------------
-----

                                      ------
                                      | 61 |
                                 _____+----+_____________
                                /                        \
                       -----------                     -----------
                       | 20 | 40 |                     | 77 | 90 |
                 ______+----+----+                     +----+----+_____
          ______/           |     \                   /      \___      \______
         /                 /       \                 /           \            \
-----------   ----------------    -----------   -----------   -----------   ---
--------------
|  3 | 10 |---| 20 | 28 | 34 |----| 40 | 51 |---| 61 | 68 |---| 77 | 80 |---|
90 | 93 | *97 |
-----------   ----------------    -----------   -----------   -----------   ---
--------------

                                         ------
                                         | 61 |
                                    _____+----+_____________
                                   /                        \
                             -----------                     -----------
                             | 20 | 40 |                     | 77 | 90 |
                       ______+----+----+                     +----+----+_____
                ______/           |     \                   /      \___
\______
               /                 /       \                 /           \
\
-----------------   ----------------    -----------   -----------   -----------
----------------
|  3 | 10 | *14 |---| 20 | 28 | 34 |----| 40 | 51 |---| 61 | 68 |---| 77 | 80
|---| 90 | 93 | 97 |
-----------------   ----------------    -----------   -----------   -----------
----------------
```

Linear dynamic hashing

Insert the following sequence of keys into an initially empty hash table with 2 buckets.

```
1000, 972, 713, 12, 436, 113, 116
```

Use linear dynamic hashing with the hash function *h*(*x*) = *x* % 10

Grow the table whenever the number of items (*f*) becomes more than twice the number of buckets (*n*).

> **Solution**
>
> First we calculate *i*. $i = \log_2 n = \log_2 2 = 1$.
>
> We start with the following hash table:
>
> ```
> 0 []
> 1 []
> ```
>
> 1. Insert 1000: *h*(1000) = 0 = 0000. The rightmost *i* bit of *h*(1000), where *i* = 1, is 0, so we add 1000 to bucket 0. *f* is now 1.
>
>    ```
>    0 [1000]
>    1 []
>    ```
>
> 2. Insert 972. *h*(972) = 2 = 0010. The rightmost 1 bit is 0, so add 972 to bucket 0. *f* = 2.
>
>    ```
>    0 [1000, 972]
>    1 []
>    ```
>
> 3. Insert 713. *h*(713) = 3 = 0011. The rightmost bit is 1, so add 713 to bucket 1. *f* = 3.
>
>    ```
>    0 [1000, 972]
>    1 [713]
>    ```
>
> 4. Insert 12. *h*(12) = 2 = 0010. The rightmost bit is 0, so add 12 to 0. *f* = 4 = 2*n, so on the next insert we will have to add a bucket.
>
>    ```
>    0 [1000, 972, 12]
>    1 [713]
>    ```
>
> 5. Insert 436. *h*(436) = 6 = 0110. The righmost bit is 0, so we add it to bucket 0.
>
>    ```
>    0 [1000, 972, 12, 436]
>    1 [713]
>    ```
>
>   We now have *f* = 5 > 2*2, so we have to grow the table by 1 bucket. *n* += 1 yields *n* = 3.
>
>   Now we must test to see whether we need to add to *i*. $i = \log_2 3$, $1 < i < 2$, so we round up to *i* = 2. Now we grow the table by a bucket to produce bucket 10, and add a leading zero to the current hash values.
>
>    ```
>    00 [1000, 972, 12, 436]
>    01 [713]
>    ```

```
10 []
```

To determine which bucket to split/rehash, we note that keys that would have belonged in bucket 01 if it had been there were put in bucket 00 instead, so we split bucket 00 to obtain:

```
00 [1000] (because 1000 hashes to 0000, or 00)
01 [713] (we don't need to look at the contents of this bucket at all)
10 [972, 12, 436] (972 and 12 hash to 0010; 436 hashes to 0110 or 10)
```

6. Insert 113. *h*(113) = 3 = 0011. The rightmost *i* = 2 bits are 11. However, there is no bucket 11 so we ignore the leading digit and use the bucket specified by the rightmost *i* - 1 bits. So, we insert 113 into bucket 01. Now *f* = 6, so on the next insert we'll have to add a bucket.

```
00 [1000]
01 [713, 113]
10 [972, 12, 436]
```

7. Insert 116. *h*(116) = 6 = 0110, so we put it in bucket 10.

```
00 [1000]
01 [713, 113]
10 [972, 12, 436, 116]
```

We now have *f* = 7 > 2*3, so we have to grow the table by 1 bucket. *n* += 1 yields *n* = 4.

i = $\log_2 4 = 2$, which is unchanged, so we add bucket 11:

```
00 [1000]
01 [713, 113]
10 [972, 12, 436, 116]
11 []
```

To determine which bucket to split/rehash, we note that keys that would have belonged in bucket 11 if it had been there were put in bucket 01 instead, so we split bucket 01 to obtain:

```
00 [1000] (we don't need to look at the contents of this bucket at all)
01 [] (nothing remains after rehashing!)
10 [972, 12, 436, 116] (we don't need to look at the contents of this bucket at
all)
11 [713, 113] (they both hash to 0011 or 11)
```

Notice that linear hashing doesn't necessarily split most efficiently. We're splitting a bucket with only 2 values, and we're not even splitting it evenly. If we were limited to, say, 3 values per bucket, we would have to use an overflow bucket to store 116 because it is the fourth value with a hash of 10 (and so we have to "chain" on an overflow bucket):

```
00 [1000]
01 []
10 [972, 12, 436]---[116]
11 [713, 113]
```

Note: You don't have to worry about overflow buckets in the linear-hashing problem in the problem set!

Lab 5: Transactions and schedules; locking

Transactions and schedules

Locking

Transactions and schedules

1. Consider this schedule of two transactions:

T1	T2
r(X)	
	r(X)
w(Y)	
	w(Y)
commit	
	commit

Is this schedule:

- serializable?

> **Solution**
>
> Yes, because its effects are equivalent to those of the serial schedule T1;T2 – i.e., the schedule in which T1 performs all of its operations and then T2 performs all of its operations.
>
> In both the schedule above and the schedule T1;T2:
>
> - Y is the only data item that is changed, and T2 writes it after T1 does.
> - T2 reads the original value of X, and it does *not* read the value of Y that X wrote.
>
> Thus, the schedule above *is* serializable.

- conflict serializable?

> **Solution**
>
> Yes, when it comes to conflicting actions, the schedule above is equivalent to the serial schedule T1;T2.

> To see this, note that there is only one pair of conflicting actions: T1’s write of A, and T2’s write of A. This means that T1 must come before T2 in any equivalent serial schedule. And since this is the only conflict-based constraint, the original schedule is equivalent to T1;T2.
>
> An alternative way of showing this is to note that T2’s read of X doesn’t conflict with T1’s write of Y because the two operations involve different data items. Therefore, we can swap those two operations to produce the serial schedule T1;T2. (Note that the commits do *not* matter when it comes to determining conflict serializability).

- recoverable?

> **Solution**
>
> Yes, because there are no dirty reads—i.e., there are no situations in which one transaction reads a value that has been written by another transaction that has yet to commit. Therefore, there’s no need to worry about the order in which the transactions commit.

- cascadeless?

> **Solution**
>
> Yes, because there are no dirty reads.

2. Consider this schedule of two transactions:

T1	T2
r(Y)	
	r(Y)
	w(X)
r(X)	
w(Y)	
	w(Y)
	commit
commit	

Is this schedule:

- serializable?

> **Solution**
>
> No. This schedule is not equivalent to either of the two possible serial schedules, T1;T2 or T2;T1.

It’s not equivalent to T1;T2 because:

- In the schedule above, T2 reads the initial value of Y, and it performs the final writes of both X and Y.
- In T1;T2, T2 still performs the final writes of X and Y, but it would read the value of Y written by T1 and thus it might write different values for those data items.

It’s not equivalent to T2;T1 because:

- In the schedule above, T2, writes the final value of Y.
- In T2;T1, T1 would write the final value of Y.

Thus, it is *not* serializable.

- conflict serializable?

Solution

No. Examining the constraints imposed by conflicting actions, we see that:

- `r1(Y)...w2(Y)` (the fact that T1 reads Y before T2 writes Y) means that T1 must come before T2 in the equivalent serial schedule.
- `r2(Y)...w1(Y)` means that T2 must come before T1.

These constraints contradict each other, so the original schedule is *not* conflict serializable.

- recoverable?

Solution

Yes. Although T1 performs a dirty read in this schedule (reading the value of X that T2 writes before T2 has committed), T1 commits **after** T2, which satisfies the requirement for a recoverable schedule. If T2’s write is undone by a crash or abort, T1 can also be rolled back, since it won’t have committed yet. While such a cascading rollback is undesirable, it prevents us from a possible inconsistent state.

- cascadeless?

Solution

No, because T1 performs a dirty read, and thus we can end up with a cascading rollback as described above.

3. Would any of the answers for schedule 2 change if we swapped the order of the two commits?

Solution

The schedule would no longer be recoverable.

Now that T1 commits before T2 does, if the system were to crash after T1 committed but before T2 committed—or if T2 were to be rolled back after T1 committed—the system could be put into an inconsistent state.

This is because:

- T1 performed a dirty read of the value of X that T2 wrote (and may have based its write of Y on that value).
- A rollback of T2 will undo T2's write of X.
- Because T1 read that value, it should be rolled back, too, but if T2's rollback happens after T1 commits, it's too late to roll back T1.

4. Consider this schedule of four transactions:

	T1	T2	T3	T4
1	r(X)			
2		r(X)		
3	w(Y)			
4			r(Y)	
5		r(Y)		
6		w(X)		
7			r(W)	
8			w(Y)	
9				r(W)
10				r(Z)
11				w(W)
12	r(Z)			
13	w(Z)			

Draw a precedence graph to determine if this schedule is conflict serializable.

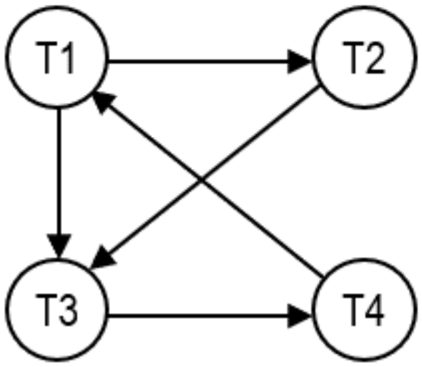
Solution

We add an edge from transaction A to transaction B in the graph if transaction A would have to come before transaction B in an equivalent serial schedule.

Here are all of the conflicting actions from different transactions and their corresponding constraints:

conflicting actions	ordering constraint
r1(X)...w2(X)	T1 → T2
w1(Y)...r3(Y)	T1 → T3
w1(Y)...r2(Y)	T1 → T2
w1(Y)...w3(Y)	T1 → T3
r2(Y)...w3(Y)	T2 → T3
r3(W)...w4(W)	T3 → T4
r4(Z)...w1(Z)	T4 → T1
r1(X)...w2(X)	T1 → T2

And here's the precedence graph:


$$\begin{tikzcd} T1 \arrow[r] \arrow[d] & T2 \arrow[dl] \\ T3 \arrow[r] & T4 \arrow[ul] \end{tikzcd}$$

As you can see, there are two cycles in the graph:

T1→T3→T4→T1
T1→T2→T3→T4→T1

This means that the schedule is *not* conflict serializable.

Locking

Consider the following schedule of two transactions:

T1	T2
	r(X)
	r(Y)

T1	T2
	w(X)
r(X)	
	w(Y)
	commit
r(Y)	
w(X)	
commit	

1. Which of the reads is a dirty read?

> **Solution**
>
> T1's read of X is a dirty read because the value of X was written by T2, which has not yet committed.
>
> Note that T1's read of Y is *not* dirty, because it occurs *after* the writer (T2) commits.

2. Would two-phase locking prevent the dirty read from occurring? If so, explain why. If not, modify the schedule so that it uses two-phase locking but still includes a dirty read.

> **Solution**
>
> No, regular two-phase locking would not necessarily prevent the dirty read. Here's a schedule that demonstrates this fact:

T1	T2
	xl(X); r(X)
	xl(Y); r(Y)
	w(X); **u(X)**
xl(X); r(X)	
	w(Y); **u(Y)**
	commit
sl(Y); r(Y)	
w(X)	

T1	T2
u(X); **u(Y)**	
commit	

Note that both T1 and T2 employ two-phase locking: once they begin releasing locks, they don’t acquire any additional locks. However, because T2 releases its exclusive lock on X before it commits, T1 is able to read X after T2 has written it and before T2 has committed, which constitutes a dirty read.

3. How can we use locking to prevent dirty reads? Explain how your proposed approach would prevent the dirty read in the schedule from problem 1.

Solution

We can use *strict* locking, which requires that transactions hold all exclusive locks until they complete. Under strict locking, T2 would hold its exclusive lock on X until it committed. Thus, T1 would not be able to acquire the lock needed to read X until after T2 commits. Thus, it would not be able to perform the dirty read.

4. Does the approach that you specified in your answer to problem 3 replace two-phase locking (2PL), or do we still need 2PL as well? Explain your answer.

Solution

No, strict locking does not replace 2PL; it augments it. We still need 2PL, because strict locking only governs the exclusive locks, and we still need to require that transactions do not acquire any shared locks after they have begun releasing them. For example, here is an example of a problematic schedule that uses strict locking but does not use 2PL:

T1	T2
	xl(X); r(X)
	sl(Y); r(Y)
	w(X)
sl(Y); r(Y)	
u(Y)	
	xl(Y); w(Y)
	u(X); **u(Y)**; commit
xl(X); r(X)	

T1	T2
w(X)	
u(X); commit	

Note that both T1 and T2 employ strict locking: they hold their exclusive locks until they commit. However, T1 does not employ 2PL: it acquires a lock for X after having released a lock for Y.

The resulting schedule is *not* serializable. To see this, note that T2 writes both X and Y. T1 reads the value of X written by T2, but it reads the prior value of Y. Thus, this transaction is equivalent to neither of the two possible serial orderings.

Lab 6: Timestamp-based concurrency control; deadlock detection

Timestamp-based concurrency control

- Preliminaries
- 1) no commit bits; one version of each data item
- 2) commit bits; one version of each data item
- 3) no commit bits; multiple versions of each data item

Locking and deadlock detection

Timestamp-based concurrency control

Preliminaries

We will be analyzing a schedule under several variations of timestamp-based concurrency control.

You may find it helpful to consult the following summary of the rules related to timestamps from the coursepack: timestamp review

In each exercise, we will be considering the following schedule:

```
s1; r1(Y); s2; w2(Y); w2(X); s3; r3(Z); r2(Z); w3(Y); r3(X); c2; c3; w1(Y); r1(X); c1
```

You should assume that:

- The read and write timestamps of all of the data items are initially 0.
- Assume that the timestamps assigned to the transactions are:
  - TS(T1) = 100
  - TS(T2) = 200
  - TS(T3) = 300

  Note that these timestamps are consistent with the fact that T1 starts first, followed by T2, followed by T3.

In giving your answers, you should fill in a table that begins as follows:

request	response	state changes or explanation
r1(Y)	allowed	RTS(Y) = 100

If a request is allowed, the third column should include any changes to the state maintained for the data item. If a request is denied or ignored, that column should include a brief explanation of why.

You do ***not*** need to restart a transaction that is rolled back, which means that you can skip any requests by a transaction that come after the point at which it is rolled back.

1) *no* commit bits; one version of each data item

Here again is the schedule that we're considering:

```
s1; r1(Y); s2; w2(Y); w2(X); s3; r3(Z); r2(Z); w3(Y); r3(X); c2; c3; w1(Y); r1(X); c1
```

If the DBMS uses regular timestamp-based concurrency control **without** commit bits, what would be the response of the system to each of the requested actions in the schedule?

Note: Because commits don't have an effect when we're not using commit bits, you do ***not*** need to include the commit actions (c1, c2, etc.) in your table.

Solution

request	response	state changes or explanation
r1(Y)	allowed	RTS(Y) = 100
w2(Y)	allowed	WTS(Y) = 200
w2(X)	allowed	WTS(X) = 200
r3(Z)	allowed	RTS(Z) = 300
r2(Z)	allowed	RTS is not changed (see below)
w3(Y)	allowed	WTS(Y) = 300
r3(X)	allowed	RTS(X) = 300
w1(Y)	ignored	TS(T1) >= RTS(Y) but TS(T1) < WTS(Y)
r1(X)	denied; roll back	TS(T1) < WTS(X); RTS(Y) = 0

Here are some additional notes about each request:

1. **r1(Y)**:
   - test: TS(T1) ≥ WTS(Y), 100 ≥ 0, true, so allow
   - change: RTS(Y) = max(RTS(Y),TS(T1)) = max(0,100) = 100
2. **w2(Y)**:
   - first test: TS(T2) ≥ RTS(Y), 200 ≥ 100, true
   - second test: TS(T2) ≥ WTS(Y), 200 ≥ 0, true, so allow
   - change: WTS(Y) = TS(T2) = 200
3. **w2(X)**:
   - first test: TS(T2) ≥ RTS(X), 200 ≥ 0, true
   - second test: TS(T2) ≥ WTS(X), 200 ≥ 0, true, so allow

- change: WTS(X) = TS(T2) = 200

4. **r3(Z)**:
   - test: TS(T3) ≥ WTS(Z), 300 ≥ 0, true, so allow
   - change: RTS(Z) = max(RTS(Z),TS(T3)) = max(0,300) = 300
5. **r2(Z)**:
   - test: TS(T2) ≥ WTS(Z), 200 ≥ 0, true, so allow
   - change: RTS(Z) = max(RTS(Z),TS(T3)) = max(300,200) = 300
   - Note that the RTS doesn't change in this case! The RTS is *not* necessarily the timestamp of the most recent reader. Rather, it is the timestamp of the reader that comes latest in the equivalent serial schedule. These values may not be the same because reads don't conflict with each other and thus they don't need to follow the timestamp ordering.
6. **w3(Y)**:
   - first test: TS(T3) ≥ RTS(Y), 300 ≥ 100, true
   - second test: TS(T3) ≥ WTS(Y), 300 ≥ 200, true, so allow
   - change: WTS(Y) = TS(T3) = 300
   - Note that the WTS is always the timestamp of the most recent writer, since writes are only allowed if they follow the timestamp ordering – i.e., if the transaction requesting the write comes later in the timsteamp ordering than the writer of the previous value.
7. **r3(X)**:
   - test: TS(T3) ≥ WTS(X), 300 ≥ 200, true, so allow
   - change: RTS(Y) = max(RTS(Z),TS(T3)) = max(200,300) = 300
   - Note that that this is a dirty read, because the writer of the prior value of X (T2) has not yet committed. Without the use of commit bits, dirty reads may occur!
8. **w1(Y)**:
   - first test: TS(T1) ≥ RTS(Y), 100 ≥ 100, true
   - second test: TS(T1) ≥ WTS(Y), 100 ≥ 300, false, so ignore
   - Note that the write cannot be allowed, because the current value of Y was written by T3, which comes after T1 in the equivalent serial schedule based on the timestamps. The DBMS ignores the request—treating T1's write as if it occurred before the write by T3 and was overwritten.
9. **r1(X)**:
   - test: TS(T1) ≥ WTS(X), 100 ≥ 200, false, so deny and roll back
   - Note that the WTS tells us that T2 wrote the current value of X. T1 comes before T2 in the equivalent serial schedule, so it should have read the original value of X instead of its current value. Thus, this read cannot be allowed. The system rolls back T1 and will later restart it with a larger timestamp.

2) commit bits; one version of each data item

When using timestamp-based concurrency control, the DBMS can optionally maintain commit bits. What is the purpose of doing so?

> **Solution**
>
> To prevent dirty reads, and to thereby produce schedules that are recoverable and cascadeless.

Consider again the following schedule:

```
s1; r1(Y); s2; w2(Y); w2(X); s3; r3(Z); r2(Z); w3(Y); r3(X); c2; c3; w1(Y); r1(X); c1
```

What would the system's response be to each of the requested actions if it uses commit bits? Make sure to include the responses to the commit requests (c1, c2, c3), since they are relevant when a system uses commit bits.

In addition to our prior assumptions, you should assume that all of the commit bits are initially true.

If a transaction is made to wait, you should include the relevant request a second time when the transaction is allowed to try again.

Solution

request	response	state changes or explanation
r1(Y)	allowed	RTS(Y) = 100
w2(Y)	allowed	WTS(Y) = 200, c(Y) = false
w2(X)	allowed	WTS(X) = 200, c(X) = false
r3(Z)	allowed	RTS(Z) = 300
r2(Z)	allowed	RTS is not changed (see below)
w3(Y)	allowed	WTS(Y) = 300, C(Y) remains false
r3(X)	denied; make wait	TS(T3) >= WTS(X) but c(X) == false
c2	allowed	c(X) = true
r3(X)	now allowed!	RTS(X) = 300
c3	allowed	c(Y) = true
w1(Y)	ignored	TS(T1) >= RTS(Y) but TS(T1) < WTS(Y)
		and c(Y) == true
r1(X)	denied; roll back	TS(T1) < WTS(X); RTS(Y) = 0
c1	doesn't happen	T1 has already been rolled back

Here are some additional notes about each request:

1. **r1(Y)**: same as before, but we also check that c(Y) == true before allowing the read.
2. **w2(Y)**: same as before, but we also set c(Y) to false to indicate that the writer of Y has not yet committed.
3. **w2(X)**: same as before, but we also set c(X) to false.
4. **r3(Z)**: same as before, but we also check that c(Z) == true before allowing the read
5. **r2(Z)**: same as before, but we also check that c(Z) == true.
6. **w3(Y)**: same as before; c(Y) remains true, and T3 becomes Y's most recent writer.
7. **r3(X)**:
   - first test: TS(T3) ≥ WTS(X), 300 ≥ 200, true
   - second test: check c(X), it's false so deny and make wait
   - Since X's commit bit is false, we know that the writer of its current value has not yet committed. Thus, T3 is made to wait to avoid a dirty read.
8. **c2**: T2 commits.
   - c(X) is set to true because T2 is the writer of X's current value.
   - Note that we do *not* set c(Y) to true. T2 did write Y, but T3 overwrote that value, so T2 is not the writer of Y's current value.
9. **r3(X)**: now that c(X) is true, T3 is allowed to try again and the read is allowed.
10. **c3**: T3 commits.
    - c(Y) is set to true because T3 is the writer of Y's current value.
11. **w1(Y)**:
    - first two tests: same as before
    - third test: c(Y) is true, so ignore
    - Note: If c(Y) were false in this case, T1 would be made to wait, since the value that it wants to write may be needed if (1) the writer of Y's current value is rolled back and (2) there is another transaction waiting to read Y when the rollback occurs.
12. **r1(X)**:
    - test: TS(T1) ≥ WTS(X), 100 ≥ 200, false, so deny and roll back
    - since T1 is the latest reader of Y (RTS(Y) == 100), we restore RTS(Y) to its prior value of 0.
13. **c1**: T1 has already been rolled back, so this action doesn't occur.

3) *no* commit bits; multiple versions of each data item

When using timestamp-based concurrency control, the DBMS can optionally maintain multiple versions of each item. What is the purpose of doing so?

Solution

To prevent rollbacks that stem from reading an item.

Consider again this schedule:

```
s1; r1(Y); s2; w2(Y); w2(X); s3; r3(Z); r2(Z); w3(Y); r3(X); c2; c3; w1(Y); r1(X); c1
```

If the DBMS uses timestamp-based concurrency control **without** commit bits but with multiple versions of data items, what would be the response of the system to each of the requested actions in the schedule?

You should make the same assumptions as before about the timestamps of the transactions and the initial timestamps of the data items, and you should assume that all of the commit bits are initially true.

Note: Because commits don't have an effect when we're not using commit bits, you do ***not*** need to include the commit actions (`c1`, `c2`, etc.) in your table.

Remember to create a new version of a data item whenever necessary.

Solution

request	response	state changes or explanation
r1(Y)	allowed to read Y(0)	RTS(Y(0)) = 100
w2(Y)	allowed	create Y(200) with RTS = 0
w2(X)	allowed	create X(200) with RTS = 0
r3(Z)	allowed to read Z(0)	RTS(Z(0)) = 300
r2(Z)	allowed to read Z(0)	RTS(Z(0)) is unchanged
w3(Y)	allowed	create Y(300) with RTS = 0
r3(X)	allowed to read X(200)	RTS(X(200)) = 300
w1(Y)	allowed	create Y(100) with RTS = 0
r1(X)	allowed to read X(0)	RTS(X(0)) = 100

Here are some additional notes about each request:

1. **r1(Y)**:
   - test: none, because reads are always allowed when there are multiple versions
   - the DBMS finds the appropriate version for T1 to read based on its timestamp, which in this case is the original version, Y(0)

- change: RTS(Y(0)) = max(RTS(Y(0),TS(T1)) = max(0,100) = 100

2. **w2(Y)**:
    - start by finding the version of Y that T2 should be overwriting based on its timestamp, which in this case is Y(0)
    - only test: TS(T2) ≥ RTS(Y(0)), 200 ≥ 100, true, so allow
    - the write creates a new version of Y, Y(200) – i.e., a version of T whose WTS is 200. The RTS of the new version is initially 0.
3. **w2(X)**:
    - T2 should be overwriting X(0)
    - only test: TS(T2) ≥ RTS(X(0)), 200 ≥ 0, true, so allow
    - a new version X(200) is created with RTS = 0
4. **r3(Z)**:
    - the DBMS finds the appropriate version for T3 to read based on its timestamp, which in this case is the original version, Z(0)
    - change: RTS(Z(0)) = max(RTS(Z(0),TS(T3)) = max(0,300) = 300
5. **r2(Z)**:
    - T2 should read Z(0)
    - no state change, since T2 isn't the reader of Z(0) that comes latest in the timestamp ordering.
6. **w3(Y)**:
    - there are two versions of Y: Y(0) and Y(200)
    - based on its timestamp, T3 should be overwriting Y(200)
    - only test: TS(T3) ≥ RTS(Y(200)), 300 ≥ 0, true, so allow
    - a new version Y(300) is created with RTS = 0
7. **r3(X)**:
    - there are two versions of X: X(0) and X(200)
    - based on its timestamp, T3 should be reading X(200)
    - change: RTS(X(200)) = max(RTS(X(200),TS(T3)) = max(0,300) = 300
8. **w1(Y)**:
    - there are three versions of Y: Y(0), Y(200) and Y(300)
    - based on its timestamp, T1 should be overwriting Y(0)
    - only test: TS(T1) ≥ RTS(Y(0)), 100 ≥ 100, true, so allow
    - a new version Y(100) is created with RTS = 0
9. **r1(X)**:

- T1 should read X(0), and it is allowed to do so!
- reads never lead to rollbacks when there are multiple versions.
- change: RTS(X(0)) = max(RTS(X(0),TS(T1)) = max(0,100) = 100

Locking and deadlock detection

Assume that:

- A DBMS is using *rigorous* two-phase locking.
- An exclusive lock for an item is requested just before writing that item.
- A shared lock for an item is requested (if needed) just before reading that item.
- Upgrades of shared locks *are* allowed.
- Update locks are *not* being used.
- The system performs deadlock detection using a waits-for graph.
- A transaction commits immediately after completing its final read or write.

Consider the following schedule involving the transactions T1, T2, and T3:

```
r1(X); r1(Z); w2(Y); w2(X); r3(Z); w3(Y); w1(Z)
```

Determine if this schedule would lead to deadlock.

If deadlock occurs, show the partial schedule in table form (including lock operations and any commits) up to the point of deadlock, along with the waits-for graph at the point of deadlock.

If deadlock does not occur, show the full schedule, including lock operations and commits. In that case, you do **not** need to include the waits-for graph.

In either case, your schedule should include any changes to the sequence of operations that occur because a transaction is forced to wait for a lock. If a transaction waits for a lock that is subsequently granted, you should include two lock requests – one for the initial request, and one at the point at which the lock is granted. If two transactions wait for a lock held by the same other transaction, and that other transaction subsequently commits, you should assume that the waiting transaction with the smaller transaction number is granted its request first.

Solution

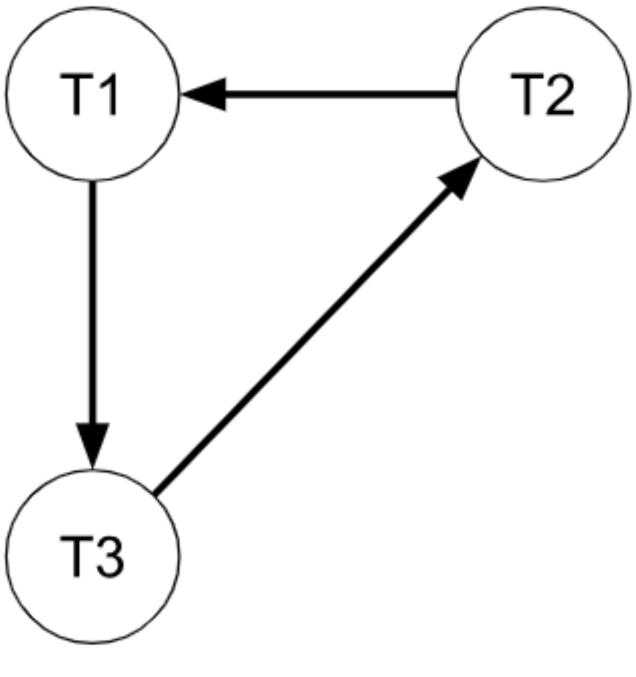
1. **r1(X)**: T1 requests and acquires a shared lock for X.
2. **r1(Z)**: T1 requests and acquires a shared lock for Z.
3. **w2(Y)**: T2 requests and acquires an exclusive lock for Y.
4. **w2(X)**: T2 requests an exclusive lock for X, but T1 already holds a shared lock for X, so the request is denied and T2 is made to wait for T1. Add a directed edge from T2 to T1 in the waits-for graph.
5. **r3(Z)**: T3 requests a shared lock for Z. T1 already holds a shared lock for Z, but multiple transactions are allowed to hold shared locks for the same data element, and thus T3 acquires the lock.
6. **w3(Y)**: T3 requests an exclusive lock for Y, but T2 already holds an exclusive lock for Y, so the request is denied and T3 is made to wait for T2. Add a directed edge from T3 to T2 in the waits-for graph.
7. **w1(Z)**: T1 attempts to upgrade its shared lock for Z to an exclusive lock, but T3 also holds a shared lock for Z, and thus the request is denied and T1 is made to wait for T3. Adding a directed edge from T1 to

T3 in the waits-for graph creates a cycle: T1 → T3 → T2 → T1.

Here is the partial schedule in table form, including the lock requests:

T1	T2	T3
sl(X)		
r(X)		
sl(Z)		
r(Z)		
	xl(Y)	
	w(Y)	
	xl(X)	
	denied	
		sl(Z)
		r(Z)
		xl(Y)
		denied
xl(Z)		
denied		

Here is the waits-for graph at the point of deadlock:



The DBMS would detect the cycle in the waits-for graph and conclude that the three transactions are deadlocked. The DBMS would then abort and restart one of the transactions so that the others could make progress.

Depending on the transaction chosen to be aborted and restarted and when it is restarted, it's possible that additional conflicts could occur, and even that deadlock could occur once again.

Lab 7: XML databases and XQuery

Designing an XML database

XQuery

- XPath expressions
- FLWOR expressions

If time permits

Designing an XML database

We want to represent a bank's data in XML. It currently uses a relational DBMS with the following relations:

```
Account(number CHAR(10) PRIMARY KEY, branch VARCHAR(20), balance FLOAT);
Customer(number CHAR(8) PRIMARY KEY, name VARCHAR(20), address VARCHAR(30));
Owns(account_number CHAR(10), customer_number CHAR(8),
    PRIMARY KEY (account_number, customer_number),
    FOREIGN KEY (account_number) REFERENCES Account,
    FOREIGN KEY (customer_number) REFERENCES Customer
);
```

1. What would be some of the advantages and disadvantages of representing this data in XML?

> **Solution**
>
> In general, representing the data in XML makes it easier to exchange data among applications and to integrate it with data from other sources. However, storing the data in XML is typically less efficient (e.g., tags are repeated in the file and queries on the data may be less efficient than querying a relational DBMS.

2. What are the main types of elements that you would need?

> **Solution**
>
> You would need elements for each type of entity in the database:
>
> - `account` elements for the data found in the `Account` table
> - `customer` elements for the data found in the `Customer` table.
>
> In theory, you could also have elements for the data in the `Owns` table. However, since the sole purpose of that table is to capture relationships between accounts and customers, we can actually capture that data using attributes, as described below.
>
> Finally, we would use nested child elements for some of the information about accounts (e.g., `balance`) and customers (e.g., `name` and `address`).

3. How could we use attributes to capture something like the primary-key and foreign-key constraints from the relational schema?

> **Solution**
>
> We could use attributes of type `ID` to capture the primary keys of `Account` and `Customer`.
>
> In addition, rather than having a separate type of element for the `Owns` table, we could do one or both of the following:
>
> - add an attribute of type `IDREFS` to each `account` element to store the customer numbers of all of the account's owners
> - add an attribute of type `IDREFS` to each `customer` element to store the account numbers of all their accounts.
>
> Recall that:
>
> - attributes of type `IDREF` must have a single value that comes from an `ID` attribute elsewhere in the document
> - attributes of type `IDREFS` must consist of a list of `ID` values that appear elsewhere in the document.
>
> **Important:** XML databases are able to capture many-to-many relationships without needing a separate type of element for the relationships themselves! In the relational model, we need a separate table for many-to-many relationships, since we can't have multi-valued attributes.

XQuery

For the rest of this lab, we'll write queries for the XML version of the bank database discussed above. You can see a small instance of this database here: sample bank database in XML

Testing your queries

You can test the queries that you write by taking the following steps:

- Download the following XML file to a known directory:
  `bank.xml`

  Note: Depending on your browser, you may need to right-click the link and click *Save Link As...* (or similar).

- Open the file in BaseX. (For details about how to use BaseX, see the last problem of Problem Set 3. Remember that if you can't get BaseX to work on your own machine, you can use it on the virtual desktop.)

- Test your queries by entering them in the Editor window in BaseX.

XPath expressions

Recall that XPath models an XML document as a tree in which the nodes represent elements and attributes, and that it allows us to access collections of nodes from the tree.

1. Write an XPath expression to obtain:

   a. the `name` elements of all the bank's customers

   > **Solution**
   >
   > ```
   > /bank/customer/name
   > ```
   >
   > or
   >
   > ```
   > //customer/name
   > ```

   b. the account numbers of all accounts with a balance that is greater than 400

   > **Solution**
   >
   > ```
   > /bank/account[balance > 400]/@account_num
   > ```
   >
   > or
   >
   > ```
   > //account[balance > 400]/@account_num
   > ```
   >
   > or
   >
   > ```
   > //account/@account_num[../balance > 400]
   > ```

FLWOR expressions

1. Write an XQuery FLWOR expression that includes a `where` clause to obtain the account numbers of all accounts with a balance that is greater than 400.

> **Solution**
>
> ```
> for $a in /bank/account
> where $a/balance > 400
> return $a/@account_num
> ```
>
> or
>
> ```
> for $a in //account
> where $a/balance > 400
> return $a/@account_num
> ```

or

```
for $a in //account[balance > 400]
return $a/@account_num
```

2. Write an XQuery FLWOR expression to produce the join of the `account` and `customer` elements. The result of the query should consist of new elements of type `cust_acct`, each of which contains a nested `customer` element followed by an associated nested `account` element. Order the results by customer number.

Solution

```
for $a in //account,
    $c in //customer
where contains($a/@owners, $c/@customer_num)
order by $c/@customer_num
return <cust_acct>{ $c, $a }</cust_acct>
```

or

```
for $a in //account,
    $c in //customer
where contains($c/@owns, $a/@account_num)
order by $c/@customer_num
return <cust_acct>{ $c, $a }</cust_acct>
```

or

```
for $a in //account,
    $c in //customer[contains(@owns, $a/@account_num)
order by $c/@customer_num
return <cust_acct>{ $c, $a }</cust_acct>
```

or

```
for $c in //customer,
    $a in //account[contains(@owners, $c/@customer_num)]
order by $c/@customer_num
return <cust_acct>{ $c, $a }</cust_acct>
```

Notes:

- We use the `contains` function to determine if the current account `$a` is owned by the current customer `$c`. This will the case:
  - if the value of `$c`'s `customer_num` attribute is contained in the list of values assigned to `$a`'s `owners` attribute

    or, equivalently,

  - if the value of `$a`'s `account_num` attribute is contained in the list of values assigned to `$c`'s `owns` attribute.

- Because we are creating new elements, we need to use curly braces around the contents of the new start and end tags. Doing so tells XQuery to evaluate what is inside of the curly braces, rather than treating it as literal text.

3. Write an XQuery FLWOR expression to create new elements of type `customer` that include:

   - a `name` child element for the name of the customer
   - new child elements for each account that the customer owns. These new `account` elements should have as their value the location of the account's branch, followed the account's balance in the form `($balance)`.

   For example:

```
<customer>
  <name>Jose Delgado</name>
  <account>Burlington, MA ($7000)</account>
  <account>Burlington, MA ($300)</account>
</customer>
```

Solution

```
for $c in //customer
return <customer>
       {
          $c/name,
          for $a in //account
          where contains($a/@owners, $c/@customer_num)
          return <account>
                 { $a/branch/text(),
                   " ($", $a/balance/text(), ")"
                 }
                 </account>
       }
       </customer>
```

Notes:

- We need a subquery so that we can group together all of a customer's accounts into a single new `customer` element. This is in contrast to the prior query, in which customers with multiple accounts ended up with multiple `<cust_acct>` elements.
- We use `$a/branch/text()` and `$a/balance/text() to remove the tags for thebranchandbalanceelements. This allows us to use just the values of those elements as the basis of the newaccount`` elements. This type of transformation can be useful when exchanging data or integrating data from heterogeneous sources.

4. Write an XQuery FLWOR expression to create new `branch` elements for each branch of the bank. These `branch` elements should have nested child elements for the location of the branch and the total balance of all of the accounts at that branch. For example:

```
<branch>
  <location>Burlington, MA</location>
  <total_balance>8300</total_balance>
</branch>
```

Hint: There is an aggregate function called `sum()` that can be used to add up a set of numeric element or attribute values.

Solution

```
for $b in distinct-values(//branch)
let $balances := //account[branch = $b]/balance
return <branch>
       {
          <location>{ $b }</location>,
          <total_balance>{ sum($balances) }<total_balance>
       }
       </branch>
```

Notes:

- We use `distinct-values()` to avoid considering a given branch more than once. Note that `distinct-values()` gives us the distinct *values* of each `branch` element. This means that `$b` will contain strings throughout the FLWOR expression.

  If we weren't using `distinct-values()` here, we'd need to use `$b/text()` to properly extract the value inside the `branch` elements so that we could use it as the value of the new location elements.

- We need a `let` clause so that we can assign the entire set of `balance` elements for a given branch to the variable `$balances`. This is what allows us to then use the aggregate function `add` to compute the sum of those balances.

  If we had instead tried to obtain the balances in the `for` clause, we would have assigned only one balance at a time to the variable, and this wouldn't have allowed us to compute the total balance.

- It isn't necessary to use the `text()` function to extract the values of the `balance` elements. The `sum` function takes care of extracting the values for us.

5. Write an XQuery FLWOR expression to create elements of type `customer_summary` that include, for each customer, child elements for the name of the customer, the address of the customer, and the number of accounts that the customer owns. For example:

```
<customer_summary>
  <name>Jose Delgado</name>
  <address>Zero Longhorn Ave., Belmont, MA</address>
  <num_accounts>2</num_accounts>
</customer_summary>
```

Solution

```
for $c in //customer
let $accounts := //account[contains(@owners, $c/@customer_num)]
return <customer_summary>
       {
          $c/name, $c/address,
          <num_accounts>{ count($accounts) }</num_accounts>
       }
       </customer_summary>
```

If time permits

If you have extra time, feel free to work on PS 3: Part II and ask the TAs for help.

Lab 8: Distributed locking and replication; MapReduce

Distributed locking and replication
An application of distributed locking and replication
MapReduce

Distributed locking and replication

Synchronous replication guarantees that transactions see the most up-to-date value when they read a replicated data item. In lecture, we discussed two different ways of implementing it:

- read-any, write-all
- voting.

Both of these approaches are characterized by three numbers:

- *n*, the total number of copies/replicas of a data item
- *w*, the number of copies that must be written
- *r*, the number of copies that must be read.

In addition, we discussed *fully-distributed locking*, which is parameterized by three numbers:

- *n*, which means the same thing as before
- *x*, the number of copies of an item that must be locked to acquire a global exclusive lock
- *s*, the number of copies that must be locked to acquire a global shared lock.

Answer the following questions:

1. What are the constraints on the values of the variables for fully-distributed locking?

> **Solution**
>
> The constraints are:
>
> - $x > n/2$, which guarantees no two transactions can acquire a global exclusive lock at one time
> - $s > n - x$, which guarantees that if one transaction has a global exclusive lock, another can't have a global shared lock, and vice versa

2. Explain how read-any, write-all replication obeys these constraints.

> **Solution**
>
> Read-any, write-all replication means that:

- When writing, a transaction updates (and thus acquires exclusive locks) for all *n* replicas ($x = w = n$)
- When reading, a transaction can access (and thus acquire a shared lock for) any replica ($s = r = 1$)

If we substitute for *x* and *s* in the above inequalities, we get $n > n/2$ (which is true) and $1 > n - n$ (which is true). This means that read-any, write-all replication always obeys the inequalities for fully distributed locking.

3. Explain how voting-based replication can obey these constraints.

Solution

When using voting with fully distributed locking, a *majority* of the copies must be written. In general, voting replication does not require this, but it's necessary when using fully distributed locking to ensure that we satisfy the inequalities for global exclusive and global shared locks. By requiring that *w* be more than half of the copies, we get the following:

- When writing, a transaction updates (and thus acquires exclusive locks for) a majority of the copies ($x = w > n/2$)
- When reading, a transaction reads (and thus acquries shared locks for) more than $n - x$ copies ($s = r > n - x$)

4. Under voting-based replication with fully distributed locking, are all combinations of *s* and *x* that satisfy the constraints equally good?

Solution

No, some may be more efficient than others. For example, when $n = 10$, we could in theory use $s = 6$ and $x = 7$. The constraints are satisfied, but we are requiring that more shared locks be obtained than are strictly necessary.

Ultimately, the choice of values for *s* and *x* will depend on the nature of your workload. For example, if you knew that you had a read-intensive workload, you might prefer to make *x* larger so that *s* can be smaller.

5. Do the same constraints apply if we're using *primary-copy* locking in combination with voting?

Solution

No. Because primary-copy locking involves locking only the primary copy of a replicated item – rather than acquiring local locks for each copy that is accessed – it is **not** necessary to write more than half of the copies. Rather, we can choose any combination of *w* and *r* that satisfies $r > n - w$.

An application of distributed locking and replication

Imagine that you are hired as a database administrator for a new chain of banks. Your job is to design a database to hold customer and account data. Your manager explains to you that he thinks asynchronous replication of data should be used between the various branches because he wants the database to be faster than those of competing banks. As a student of CS 460, you know this could be a bad idea.

1. What is the primary reason that asynchronous replication in a bank database would be a bad idea?

   > **Solution**
   >
   > Asynchronous replication is faster than synchronous replication, but it cannot guarantee that the most up-to-date value is present. For a bank application, this would be an especially poor choice, since if a customer did a balance query on an out-of-date, secondary copy of the account, they might think they had more money than was actually in the account.

2. Come up with an alternate distributed database scheme to present to your manager. In particular:

   - Which would be the better form of synchronous replication: read-any/write-all or voting? Think about what must take place for balance queries and withdrawal/deposit transactions.

     > **Solution**
     >
     > Neither approach is clearly better. Balance queries only require reads, deposits only require writes, and withdrawals require both reads and writes. If withdrawals and deposits are more common than balance queries, it may make sense to use voting, since writes are very expensive in the read-any/write-all scheme.

   - Which would be the best form of distributed concurrency control: centralized locking, primary-copy locking, or fully distributed locking? You can assume that speed is still important, and it is expected that there will be many bank branches.

     > **Solution**
     >
     > Primary-copy locking may be a good compromise in this case. A single point of failure and a bottleneck for lock requests is enough to rule out centralized locking. If there are a large number of branches, fully distributed locking may be too slow. Therefore, primary-copy locking may be the best compromise between safety and speed.

MapReduce

1. Using **pseudocode**, design the mapper and reducer functions for counting the occurrences of words in a set of input files.

   > **Solution**
   >
   > The mapper accepts a (key, value) pair in which the value is one of the lines in one of the input files. The key represents the offset of the line in the input file and is not useful for this application.
   >
   > The line should be split into individual words, and then each word should be written/emitted with a count of 1.

```
mapper(key, line):
    words[] = split(line)
    for each word in words:
        emit(word, 1)
```

The reducer also accepts a (key, value) pair, where the key is a word and the value is a list of counts for that word. After the counts for the word are summed, the word and the total is output.

```
reducer(word, counts[]):
    total = 0
    for each count in counts:
        total += count
    emit(word, total)
```

Note: In this application, all of the counts will be 1, but it still makes sense to use the actual values in the list instead of assuming all ones. That way, the reducer will still work if we decide to add some preliminary combining of values before the final reduction.

To see a full implementation of this mapper and reducer for Apache Hadoop, see the `WordCount.java` file that was included in the code provided for PS 4.

Lab 9: MongoDB queries

Our MongoDB movie database

Practice writing MongoDB queries

Our MongoDB movie database

Recall that the MongoDB version of our movie database consists of three collections:

1. one called `movies` containing documents about movies
2. one called `people` containing documents about actors and directors
3. one called `oscars` containing documents about Academy Awards.

In addition, remember that:

- Whenever we refer to a person or movie, we also embed the associated entity's name.
- Person documents include one or both of the boolean fields `hasActed` and `hasDirected`. These fields are only included when their values are true.

Here are some sample documents:

```
{
    _id:             "0499549",
    name:            "Avatar",
    year:            2009,
    rating:          "PG-13",
    runtime:         162,
    genre:           "AVYS",
    earnings_rank:   4,
    actors: [
        { id: "0000244", name: "Sigourney Weaver" },
        { id: "0002332", name: "Stephen Lang" },
        { id: "0735442", name: "Michelle Rodriguez" },
        { id: "0757855", name: "Zoe Saldana" },
        { id: "0941777", name: "Sam Worthington" }
    ],
    directors: [ { id: "0000116", name: "James Cameron" } ]
}

{
    _id:          "0000059",
    name:         "Laurence Olivier",
    dob:          "1907-5-22",
    pob:          "Dorking, Surrey, England, UK",
    hasActed:     true,
    hasDirected:  true
}

{
    _id:     ObjectID("528bf38ce6d3df97b49a0569"),
    year:   2013,
    type:   "BEST-ACTOR",
    person: { id: "0000358", name: "Daniel Day-Lewis" },
    movie:  { id: "0443272", name: "Lincoln" }
}
```

Practice writing MongoDB queries

If you have installed and configured everything according to the directions in PS 5, feel free to try out your queries using MongoDB Compass. However, doing so is not required.

Notes:

- You may only use aspects of the MongoDB query language that we discussed in lecture.
- The results of the query should include only the requested information, with no extraneous fields. In particular, you should exclude the `_id` field from the results of your queries unless the problem indicates otherwise.
- You do ***not*** need to worry about the order of the fields in the results.

Here are the problems:

1. Find the top ten highest grossing movies in the database. For each such movie, output a document with the movie's name and the name(s) of the director(s).

   a. How could we solve this problem using the `find` method?

   Solution

   ```
   db.movies.find( { earnings_rank: { $lte: 10 }},
                   { name: 1, "directors.name": 1, _id: 0 }
                 )
   ```

   b. How could we solve this problem using an aggregation pipeline (i.e., the `aggegrate` method)? In addition to finding the same results as before, rename the field for the names of the directors, giving it the name `directors`.

   Solution

   ```
   db.movies.aggregate(
       { $match:   { earnings_rank: { $lte: 10 } } },
       { $project: { name: 1, directors: "$directors.name", _id: 0 } }
   )
   ```

2. How many actors in the database were born in California?

   a. How could we solve this problem using the `count` method?

Solution

```
db.people.count({ hasActed: true, pob: /, California/ })
```

b. How could we solve this problem using an aggregation pipeline? The result should be a single document with a single field named `num_actors`.

Solution

```
db.people.aggregate(
    { $match:   { hasActed: true, pob: /, California/ } },
    { $group:   { _id: null, num_actors: { $sum: 1 } } },
    { $project: { num_actors: 1, _id: 0 } }
)
```

Note: Using `null` as the basis of the `$group` stage creates a single group for all of the documents that satisfy the `$match` stage, since we just want a single count of all of those documents.

3. Which people in the database have directed a movie in which Tom Hanks has acted? Use the `distinct` method to produce a list containing the names of these directors, with each name appearing only once.

Solution

```
db.movies.distinct("directors.name", { "actors.name": "Tom Hanks" })
```

4. Find all years before 1970 that have 6 or more movies in the database. Each result document should include a field called `year` for the year and a field called `count` for the number of movies from that year.

Solution

```
db.movies.aggregate(
    { $match:   { year: { $lt: 1970 } } },
    { $group:   { _id: "$year", count: { $sum: 1 } } },
    { $match:   { count: { $gte: 6 } } },
    { $project: { year: "$_id", count: 1, _id: 0 } }
)
```

5. Find the person in the database who has acted in the most movies. The final result should be a single document with fields named `actor` and `num_movies`. You may assume that actor names are unique. If there are multiple actors who are tied for the most movies, the result can be any one of them.

   Hint: Begin by using an $unwind stage to “unwind” the arrays of actors in all of the movie documents. Doing so will allow you to create a separate group for each actor.

Solution

```
db.movies.aggregate(
    { $unwind:  "$actors" },
    { $group:   { _id: "$actors.name",
                  num_movies: { $sum: 1 } } },
    { $sort:    { num_movies : -1 } },
    { $limit: 1  },
    { $project: { _id: 0, actor: "$_id", num_movies: 1 } }
)
```

Note: If we don't assume that actor names are unique, we would need to create subgroups based on **actor id**. Then, in order to include the name of the actor in the results of the `$group` stage, we could use an aggregator called `$first` as shown below:

```
db.movies.aggregate(
    { $unwind:  "$actors" },
    { $group:   { _id: "$actors.id",
                  actor: { $first: "$actors.name" },
                  num_movies: { $sum: 1 } } },
    { $sort:    { num_movies : -1 } },
    { $limit: 1  },
    { $project: { _id: 0 } }
)
```

For each subgroup of documents, the `$first` aggregator evaluates the specified expression using the *first document* in that subgroup. And since we know that *all* of the documents in a given `actors.id` subgroup will have the same value for `actors.name`, this approach gives us the name of the actor for that subgroup!

Note that because we're not grouping on `actors.name`, we can use the desired field name for it (`actor`) in the `$group` stage, which simplifies the final `$project` stage.

6. Who directs the longest movies? Of all directors who have directed 5 or more movies, find the 10 directors whose movies have the longest average runtime. The final result should consist of documents with fields named `director` (for the director's name), `avg_runtime`, and `num_movies`.

Solution

```
db.movies.aggregate(
    { $unwind:  "$directors" },
    { $group:   { _id: "$directors.name",
                  avg_runtime: { $avg: "$runtime" },
                  num_movies: { $sum: 1 }
                }
    },
    { $match:   { num_movies: { $gte: 5 } } },
    { $sort:    { avg_runtime : -1 } },
    { $limit: 10  },
    { $project: { _id: 0, director: "$_id",
                  avg_runtime: 1, num_movies: 1
```

```
                             }
            }
      )
```

Lab 10: Logging and recovery

Lab evaluations
Logging
Standard log-based recovery
Checkpoints
Log-based recovery that uses the LSNs

Lab evaluations

We will begin by taking a few minutes to complete evaluations for the lab component of the course.

- Your evaluations are **anonymous**, and we will not receive the results until after all final grades have been submitted.
- Comments in the text fields are valued and encouraged. Please try to answer all questions, but if a question is not applicable or you do not wish to answer it, you can simply skip it.

Here is the URL that you should use: https://bu.bluera.com/bu/

- Enter your BU login name and Kerberos password, and complete the evaluation for your CS 460 **lab** section (the one that is **not** A1). Please do ***not*** evaluate the lectures at this time.
- When you are done with the survey, please close the separate browser tab.

Thanks in advance for taking the time to provide your feedback about the labs!

Logging

Based on the log shown below, answer the following questions.

LSN	transaction	action	item	old value	new value
00	T1	begin			
10	T1	update	D1	100	70
20	T2	begin			
30	T1	commit			
40	T2	update	D1	70	400
60	T2	update	D3	“hello”	“howdy”
70	T3	begin			
80	T3	update	D2	15	70

LSN	transaction	action	item	old value	new value
90	T2	commit			
100	T3	update	D1	400	55
		crash			

1. Recall that we discussed three types of logging: undo-redo, undo-only and redo-only.

   How can we can determine from looking at the log that the system is performing undo-redo logging?

   > **Solution**
   >
   > The update log records include both the old and the new values of the data item being updated.

2. At the start of recovery, what are the possible on-disk values of each data item?

   > **Solution**
   >
   > In undo-redo logging, there is no guarantee about when an updated database page will go to disk unless there is a checkpoint. As a result, any of the values in the log might be on disk at the start of recovery.
   >
   > **D1**: 100 (its original value), 70 (from T1's update), 400 (from T2's update) or 55 (from T3's update)
   >
   > **D2**: 15 (its original value) or 70 (from T3's update)
   >
   > **D3**: "hello" (its original value) or "howdy" (from T2's update)

3. If the system were using **undo-only logging** instead, what are the possible on-disk values of each data item at the start of recovery? You should assume that each data item is on a different page of the database file.

   > **Solution**
   >
   > In undo-only logging, the system needs to ensure that no updates will need to be redone during recovery. As a result, when a transaction commits, any pages that it has modified must be forced to disk.
   >
   > Before a transaction commits, its updates may or may not make it to disk, depending on whether the relevant pages are evicted from the cache.
   >
   > If we build a table for in-memory and possible on-disk values like the one used in lecture, we start by filling in the original on-disk values:

item	in-memory value	possible on-disk values
D1		100
D2		15
D3		“hello”

Tracing through the actions in the log in the order in which they occurred, the first update is T1’s update of D1, which creates an in-memory version of D1 and a new possible on-disk value:

item	in-memory value	possible on-disk values
D1	70	100, 70
D2		15
D3		“hello”

When T1 then commits, the updated in-memory value of D1 is forced to disk, which means that 100 is no longer possible on disk:

item	in-memory value	possible on-disk values
D1	70	~~100~~, 70
D2		15
D3		“hello”

T2 then updates D1 and D3, and T3 updates D2:

item	in-memory value	possible on-disk values
D1	~~70~~ 400	~~100~~, 70, 400
D2	70	15, 70
D3	“howdy”	“hello”, “howdy”

When T2 commits, the items it modified (D1 and D3) are forced to disk, which means those values are guaranteed to be on disk:

item	in-memory value	possible on-disk values
D1	~~70~~ 400	~~100, 70,~~ 400

item	in-memory value	possible on-disk values
D2	70	15, 70
D3	“howdy”	~~“hello”,~~ “howdy”

Finally, after T3’s update of D1, we have:

item	in-memory value	possible on-disk values
D1	~~70 400~~ 55	~~100, 70,~~ 400, 55
D2	70	15, 70
D3	“howdy”	~~“hello”,~~ “howdy”

Thus, the possible on-disk values at the start of recovery are:

- **D1**: 400 or 55
- **D2**: 15 or 70
- **D3**: “howdy”

4. If the system were using **redo-only logging** instead, what are the possible on-disk values of each data item at the start of recovery? You should assume that each data item is on a different page of the database file.

Solution

In redo-only logging, the system needs to ensure that no updates will need to be undone during recovery. As a result, it “pins” a modified database page in memory and doesn’t let it go to disk until the transaction that made the change commits.

We again start with the original on-disk values:

item	in-memory value	possible on-disk values
D1		100
D2		15
D3		“hello”

When T1 updates D1, we get an in-memory version of D1, but because it is pinned in memory, that value can’t be on disk at first:

item	in-memory value	possible on-disk values
D1	70	100
D2		15
D3		“hello”

When T1 then commits, the updated in-memory value of D1 is unpinned, which means it can go to memory at any point after the commit. However, it is **not** forced to disk, so 100 is still possible:

item	in-memory value	possible on-disk values
D1	70	100, 70
D2		15
D3		“hello”

After T2 updates D1 and D3, and T3 updates D2, we have new in-memory values, but they can’t be on disk yet because they are pinned until the transactions commit:

item	in-memory value	possible on-disk values
D1	~~70~~ 400	100, 70
D2	70	15
D3	“howdy”	“hello”

When T2 commits, the items it modified (D1 and D3) are unpinned and allowed to go to disk:

item	in-memory value	possible on-disk values
D1	~~70~~ 400	100, 70, 400
D2	70	15
D3	“howdy”	“hello”, “howdy”

After T3’s update of D1, we have a new in-memory value, but it can’t be on disk because T3 hasn’t committed:

item	in-memory value	possible on-disk values
D1	~~70 400~~ 55	100, 70, 400

item	in-memory value	possible on-disk values
D2	70	15
D3	“howdy”	“hello”, “howdy”

And thus the possible on-disk values at the start of recovery are:

- **D1**: 100, 70 or 400
- **D2**: 15
- **D3**: “hello” or “howdy”

Standard log-based recovery

Consider again the log shown below:

LSN	transaction	action	item	old value	new value
00	T1	begin			
10	T1	update	D1	100	70
20	T2	begin			
30	T1	commit			
40	T2	update	D1	70	400
60	T2	update	D3	“hello”	“howdy”
70	T3	begin			
80	T3	update	D2	15	70
90	T2	commit			
100	T3	update	D1	400	55
		crash			

1. During recovery, what steps would be taken during the first pass through the log if the system is using undo-redo logging?

Solution

The first pass is the backwards pass, which starts with the last log record and works backward.

Updates by *uncommitted* transactions are undone, and committed transactions are added to the commit list. Other log records are skipped.

Here is a summary of what happens:

- LSN 100: T3 is not on the commit list. Undo the update, writing a value of 400 for D1.
- LSN 90: Add T2 to the commit list.
- LSN 80: T3 is not on the commit list. Undo the update, writing a value of 15 for D2.
- LSN 70: skip
- LSN 60: T2 is on the commit list; skip.
- LSN 40: T2 is on the commit list; skip.
- LSN 30: Add T1 to the commit list.
- LSN 20: skip
- LSN 10: T1 is on the commit list; skip.
- LSN 00: skip

2. What steps would be taken during the second pass through the log?

Solution

The second pass is the forwards pass, which starts with the first log record and works forward.

Updates by *committed* transactions are redone, and other log records are skipped.

Here is a summary of what happens:

- LSN 00: skip
- LSN 10: T1 is on the commit list.
  Redo the update, writing a value of 70 for D1.
- LSN 20: skip
- LSN 30: skip
- LSN 40: T2 is on the commit list.
  Redo the update, writing a value of 400 for D2.
- LSN 60: T2 is on the commit list.
  Redo the update, writing a value of “howdy” for D3.
- LSN 70: skip
- LSN 80: T3 is not on the commit list; skip.
- LSN 90: skip
- LSN 100: T3 is not on the commit list; skip.

3. How would the log and the recovery process be different under **undo-only** logging?

> **Solution**
>
> With undo-only logging, we only store the information needed to *undo* operations (the *old* values). Our log would not need the "new value" column. During recovery, we would only need to perform the backward pass.

4. How would the log and the recovery process be different under **redo-only** logging?

> **Solution**
>
> With redo-only logging, we only store the information needed to *redo* operations (the *new* values). Our log would not need the "old value" column. During recovery, the backward pass would only build the commit list, and then the forward pass would be performed as usual.

Checkpoints

Now let's assume that a dynamic checkpoint occurred between log records 40 and 60:

LSN	transaction	action	item	old value	new value
00	T1	begin			
10	T1	update	D1	100	70
20	T2	begin			
30	T1	commit			
40	T2	update	D1	70	400
50		**checkpoint**			
60	T2	update	D3	"hello"	"howdy"
70	T3	begin			
80	T3	update	D2	15	70
90	T2	commit			
100	T3	update	D1	400	55
		crash			

1. What other information must be included in log record 50?

> **Solution**
>
> The list of active transactions, which in this case is only T2.

2. Given the checkpoint, do all of the log records shown need to be examined during the backward pass? During the forward pass? Why or why not?

> **Solution**
>
> **Backward pass**
> For this log, the backward pass only needs to go back to log record 50 — the one describing the checkpoint.
>
> In general, we need to go back to the oldest begin record for any transaction that needs to be undone (i.e., a transaction that is listed as active in the checkpoint record but isn't on the commit list). This allows us to undo any changes made by such transactions.
>
> In this case, the only transaction listed in the checkpoint record is T2, and it's on the commit list. Because any changes that T2 made before the checkpoint were forced to disk during the checkpoint, we don't need to undo them, and thus there's no need to go any further back.
>
> **Forward pass**
> Because a checkpoint forces all changed data items to disk, the forward pass can always begin after the checkpoint. In this case, this means that it can begin at log record 60.

3. Does the existence of the checkpoint change the possible on-disk values at the start of recovery under undo-redo logging?

> **Solution**
>
> Yes. Because the checkpoint forces all changed data items to disk, it is no longer possible for D1 to have a value of 100 or 70 at the start of recovery. Rather, it must either be 400 (the value forced to disk at the checkpoint) or 55 (the value written by T3 after the checkpoint).

Log-based recovery that uses the LSNs

Now assume that the system is performing logical logging, which means:

- each on-disk data item must have a corresponding *datum LSN*, which is the LSN of the log record that produced the value
- each update log record must include the old log sequence number (the *olsn*), as shown below. The olsn is the LSN of the log record that produced the data item's previous value.

LSN	transaction	action	item	old value	new value	olsn
00	T1	begin				

LSN	transaction	action	item	old value	new value	olsn
10	T1	update	D1	100	70	0
20	T2	begin				
30	T1	commit				
40	T2	update	D1	70	400	10
50		*checkpoint*				
60	T2	update	D3	“hello”	“howdy”	0
70	T3	begin				
80	T3	update	D2	15	70	0
90	T2	commit				
100	T3	update	D1	400	55	40
		crash				

1. What other changes to the log would typically be made under logical logging?

> **Solution**
>
> Instead of storing the old and new values of the data items, we would store a logical description of *how* to get the new value from the old value (e.g., log record 10 could be stored as “decrement D1 by 30”).

2. What is the purpose of storing the olsn and datum LSN values?

> **Solution**
>
> It allows us to only redo and undo updates that are necessary based on the values that made it to disk before the crash.
>
> This is especially important under true logical logging, because logical updates may not be *idempotent* – i.e., doing them multiple times may not lead to the same final value – and thus it is important to only undo changes that actually made it to disk and to only redo changes that didn’t make it to disk.

3. At the start of recovery, assume that we have the following on-disk datum LSNs:

- D1: 100
- D2: 0
- D3: 60

Trace through the steps taken during the backward and forward passes for the log above, which includes the added checkpoint record.

Solution

Backward pass

- **100**: T3 is not on the commit list
  D1’s datum LSN = 100
  LSN of log record = 100
  because these LSNs are equal, we undo this change
  this gives D1 a value of 400 and a datum LSN of 40
- **90**: add T2 to the commit list
- **80**: T3 is not on the commit list
  D2’s datum LSN = 0
  LSN of log record = 80
  not equal, so no need to undo, because this change didn’t make it to disk
- **70**: skip
- **60**: T2 is on the commit list, so we skip this record for now
- **50**: the only active transaction in the checkpoint record (T2) is on the commit list, so we can stop here.

Forward pass

- Start at the first record after the checkpoint (60).
- **60**: T2 is on the commit list
  D3’s datum LSN = 60
  OLSN in log record = 0
  datum LSN != OLSN, so no need to redo, because this change is already on disk (or was overwritten by a later change)
- **70**: skip
- **80**: T3 is not on the commit list; skip
- **90**: skip
- **100**: T3 is not on the commit list; skip