**Problem Set 5, Part I**

**Problem 1: Data models for a NoSQL document database**

1-1)

**Author Document:**

```
{
	_id: “56789”,
	name: “Barbara Kingsolver”,
	dob: “1955-04-08”
}
```

**Book Document:**

```
{
	_id: “9780063251922”,
	title: “Demon Copperhead”,
	publisher: “Harper”,
	num_pages: 560,
	genre: “fiction”,
	numInStock: 15,
	authors: [ "56789" ]
}
```

**Sales Document:**

```
{
	 _id: <ObjectID1>,
	date: "2024-04-11",
	time: "10:45",
	book: "9780063251922",
	numSold: 1
}
```

**Note:** <ObjectID1> is a system-generated ObjectID.

1-2)

Advantages:

- Avoid possible duplication of author and book information
- Avoid possible inconsistencies between different copies of author and book info
- Book documents will be smaller than they would be if we embedded the author info

Disadvantages:

- Because joins aren't supported in all versions of MongoDB, we need to make multiple requests for queries that involve two or more types of documents
- More network/disk I/O for such queries

1-3)

**Author Document:**

```
{
        _id: "56789",
        name: "Barbara Kingsolver",
        dob: "1955-04-08"
}
```

**Book Document:**

```
{
        _id: "9780063251922",
        title: "Demon Copperhead",
        publisher: "Harper",
        num_pages: 560,
        genre: "fiction",
        numInStock: 15,
        authors: [ {     id: "56789",
                         name: "Barbara Kingsolver"
                   }]
```

**Sales Document:**

```
{
         _id: <ObjectID1>,
        date: "2024-04-11",
        time: "10:45",
        book: {          id: "9780063251922",
                         title: "Demon Copperhead"
                }
        numSold: 1
}
```

1-4)

Advantages:

- Don't need to make multiple requests for queries that involve at least certain types of info about two or more types of entities
- Less network/disk I/O for such queries

Disadvantages:

- In the first approach duplication of the author name and book name, with the resulting possibility of inconsistencies
- Book and/or sales documents will be larger than if we used references.

1-5)

When a bookseller attempts to sell one or more copies of a given book, they must make the following updates:

- Record information about the sale (the info from the Sales table)
- Update the value of the book's numInStock field, reducing it by the number of copies that were sold.

In light of this use of the database, the second embedded-document approach from part 3 would be more appropriate. MongoDB only provides atomicity for operations on a single document, so we need the sales info to be in the same document as the numInStock field so that both of the necessary changes can be made as part of an atomic sequence of operations. Otherwise, we might end up changing numInStock without recording the sales information, or vice versa.

**<u>Problem 2: Logging and recovery</u>**

2-1) undo-redo **without** logical logging

| LSN | backward pass | forward pass |
|---|---|---|
| 0 | skip | skip |
| 10 | skip | redo: B = 240 |
| 20 | skip | redo: D = 420 |
| 30 | skip | skip |
| 40 | undo: C = 300 | skip |
| 50 | skip | redo: B = 290 |
| 60 | undo: A = 100 | skip |
| 70 | add T1 to commit list | skip |
| 80 | undo: D = 420 | skip |

2-2) undo-redo **with** logical logging

| LSN | backward pass | forward pass |
|---|---|---|
| 0 | skip | skip |
| 10 | skip | 0 != 10<br>don’t redo |
| 20 | skip | 0 != 80<br>don’t redo |
| 30 | skip | skip |
| 40 | 40 = 40<br>undo: C = 300<br>datum lsn = 0 | skip |
| 50 | skip | 10 = 10<br>redo: B = 290<br>datum lsn = 50 |
| 60 | 0 != 60<br>don’t undo | skip |
| 70 | add T1 to commit list | skip |
| 80 | 80 = 80<br>undo: D = 420<br>datum lsn = 20 | skip |

2-3)

a) The backwards pass would only need to go back to record 30. Txn 2 was active at the time of the checkpoint and it did not commit before the crash, so we need to go back to its begin record so that we can undo its changes that the checkpoint forced to disk. Because txn 1 has committed since the checkpoint, we don't need to go back to its begin record, since its changes don't need to be undone.

b) The forward pass would begin after the checkpoint, at record 50, because all changes from before the checkpoint would have been forced to disk at the checkpoint, and thus nothing would need to be redone from before that point.

## Problem 3: Logging and on-disk values

3-1) A: 100, 150
B: 200, 240, 290
C: 300, 370
D: 400, 420, 480

3-2) A: 100, 150
B: 290
C: 300, 370
D: 420, 480

**explanation of the differences:**
Undo-only logging forces modified database pages to disk when the writer of those changes commits. As a result:

- When txn 1 commits, the most recent value that it wrote for B (290) is forced to disk, and thus B's prior values (200 or 240) cannot possibly be on disk at the time of the crash.
- Similarly, the value that txn 1 writes for D (420) is forced to disk when it commits, so D's prior value (400) cannot be on disk at the time of the crash.

3-3) A: 100
B: 200, 290
C: 300
D: 400, 420

**explanation of the differences:**

- Because redo-only logging pins modified database pages in memory until the writer of those changes commits, it isn't possible for values written by the uncommitted txn 2 (150 for A, 370 for C, 480 for D) to be on disk at the time of the crash.
- In addition, txn 1's write of 240 for B can't be on disk, because it was retained in memory while txn 1 was active and was overwritten by txn 1's subsequent write of 290.