Problem Set 5, Part I

Problem 1: Data models for a NoSQL document database

1.
Artist
{
	_id: "9876543",
	name: "Sara Bareilles",
	label: "Epic",
	dob: "1979-12-07",
	genre: "pop",
	sings: ["**0123456789**"]
}
Song
{
	_id: "0123456789",
	sung_by: ["**9876543**"],
	name: "Brave",
	duration: 219,
	genre: "pop",
	best_chart_rank: 23,
	royalties_due: 1200,
}
played:
{
	_id: object_id("12345") // generated id
	song_id: "0123456789"
	date: "2015-04-22",
	time: "17:00"
}

2.
+ allows you to capture more complicated relationships
+ less duplicate data
+ reduced document size
+ less inconsistencies between documents
- need more requests
- more network/disk IO
- updates are not as easy (atomic - needs to look at multiple documents to update)

Summary according to the problem: When searching for a song via an artist, it is costly because you have to link with another document. Also, when grabbing artist's name, pob, etc via song is complicated is more costly, because you have to scan through both documents. The advantage is that, there are less duplicate fields and also, the keys for `name`, `genre` do not have to be unique. (e.g.) searching for artist name by song id would require you to loop through song document then the artist document (highly inefficient).

3.

```
{
        _id: "9876543",
        name: "Sara Bareilles",
        label: "Epic",
        dob: "1979-12-07",
        genre: "pop",
        songs: [
                {
                song_id: "0123456789",
                song_name: "Brave",
                duration: 219,
                song_genre: "pop",
                best_chart_rank: 23,
                royalties_due: 1200,
                played: [
                        {
                                date: "2015-04-22",
                                time: "17:00",
                        }
                ]
        ]
}
```

4.

+ less requests
+ less network + disk IO
+ updates are atomic
- duplicate data (songs can be sung by multiple artists, dup songs in this structure)
- inconsistency between copies of dup data
- large document

Summary According to document: Grabbing information is much easier, all the data is already available! O(1) cost for querying a map-like structure like JSON EXCEPT when searching for songs (it's an array). However, making the structure simpler creates some duplicate data, like songs sung by multiple artists. Also, there are ambiguous keys like `name` and `id`, so they were changed to `song_name` and `song_id`.

5.

Each time a song is played in the music service, the date and time would be added to the array of "played" field inside both json representations in *1* and *3*. This may be useful in recording what time/date the song is played and how many times the song is played. Also, this can be super useful for doing analysis on frequency of songs played by time/season.

Problem 2: Logging and recovery

1.

Transactions	In Memory	Possible On Disk
A	~~130,~~ 150	100,130,150
B	~~510~~, 570	500, 510, 570
C	430	400, 430
D	210	200, 210

2. Redo Only

Values gets pinned to memory (e.g. new: <val>) can't go to disk until commit;

Values get unpinned in memory, (e.g. new: <val>) can go to disk, but not forced, so old values are still possible

570, 430, 210 are no longer possible, because that is part of txn2, and txn2 does not commit

Transactions	In Memory	Possible On Disk
A	~~130,~~ 150	100,130,150
B	~~510~~, 570	500, 510
C	430	400
D	210	200

3. Undo Only

Values do not get pinned to memory, so new values can go to disk at any time

Dirty values are forced to disk at commit

For A, value in memory is forced to memory, so old values 100, 130 aren't possible anymore

For B, value in memory is forced to disk, so old value 500 is no longer possible, but 570 happens after commit, so can still go to disk

Transactions	In Memory	Possible On Disk
A	~~130,~~ 150	150
B	~~510~~, 570	510,570
C	430	400, 430
D	210	200, 210

4. undo-redo **without** logical logging

LSN	**backward pass**	**forward pass**
0	skip	skip
10	skip	A=130
20	skip	skip
30	C=400	skip
40	skip	A=150
50	skip	B=510
60	add to commit list	skip
70	D=200	skip
80	B=510	skip

5. undo-redo **with** logical logging

LSN	**backward pass**	**forward pass**
0	skip	skip
10	skip, on commit list	redo, datum olsn = 0, A = 130
20	skip	skip
30	skip, datum lsn 0 != 30	skip, not committed
40	skip, on commit list	redo, datum lsn = 40, A =150
50	skip, on commit list	redo, datum lsn = 50, B = 510
60	skip	skip
70	undo: datum lsn=0 D=200	skip not committed
80	skip, datum lsn 50 != 80	skip not committed

6.

A and C would be undone, because the checkpoint occurs before the COMMIT on lsn 60. Checkpoint forces the old values to disk so new values before and including LSN 30 are no longer possible values (e.g. A=130, C=430). However, everything after the checkpoint must be the same, checkpoints would guarantee that the datum lsn's could not be the olsn. As a result, we only need to redo the committed transactions values (A, B from txn 1), which is the same as redo logs

Problem 3: Two-phase commit

1. Steps Taken for an Undo Only 2PC Scheme
 a) Coordinator sends Prepare message to other sites
 b) Each site forces all dirty logs to disk; in the case of undo only, transactions that are committed get forced to memory and transactions that have not committed are undone.
 c) Each site, if force to disk transaction was successful, tells the coordinator it is ready. However, if the transaction fails, an abort message is given.

2. The case when the coordinator is required to recover is when all sites that were polling for the coordinator are in ready state. Those sites CANNOT determine whether a transaction was aborted or committed, because they can not determine based on the previous subtransaction since the sites may have missed an entire sbtxn / or aborted even though it's in ready state. As a result, the coordinator must be brought back up to tell the sites whether they are truly in sync. The coordinator would have to check whether the previous subtransactions were committed or aborted in the logs, and then send the appropriate commands to the other sites.