Instructions:
- The total time for the exam is 50 minutes; be sure to budget your time accordingly.
- The exam is closed book and closed notes but “open cheat sheet”.
- Read each question first, in its entirety, and then carefully answer each part of the question.
- If you don’t understand something, ask one of the exam proctors for clarification.
- If you still find ambiguities in a question, note the interpretation you are taking.

<table>
<thead>
<tr>
<th>QUESTION</th>
<th>TOPIC</th>
<th>POINTS</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Indexing</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Transactions</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Physical Database Design</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NoSQL Databases</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>All</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
Question 1: Indexing (35 points)

Consider the following sketch of a secondary B+ Tree index on a table containing Patient data, created as indicated. The pages of the index are labeled for use in the questions that follow. As you are answering the questions, base your answers on using the search, insert, and delete algorithms (policies) recommended in the lectures, notes, and textbook.

```
CREATE INDEX Agelndx ON Patient(age);
```

(Order d=2)

```
P5
  25  35  44

P4
  44  45  51  63

P3
  35  37  41  42

P2
  29  33

P1
  12  22
```

(a) (6 pts) Assume that this index resides on disk and hasn’t been accessed recently. Indicate below which pages the DBMS will read from disk, in which order, to run SELECT * FROM Patient WHERE age > 43. To indicate the order, put a “1” by the first page that it will read, a “2” by the next page, and so on. Leave the space next to any untouched page empty.

```
P1 _____  P2 _____  P3 _____  P4 _____  P5 _____
```

(b) (4 pts) Suppose the user’s very next query is SELECT * FROM Patient WHERE age >= 30 AND age < 40. Indicate below which pages the DBMS will read from disk, and again in which order, if this query is run immediately after the query in (a).

```
P1 _____  P2 _____  P3 _____  P4 _____  P5 _____
```

(c) (12 pts) Listed below are several different keys or key sequences (Patient ages) that could be added to the Patient table and its associated indexes. Put an X next to any/all of the sequences that will lead to page P5 becoming “full” if that sequence of inserts is performed on the state of the tree as shown above.

```
30, 34 _____  43, 39 _____  1, 2, 25 _____  80 _____
```

(d) (9 pts) Listed below are several different keys or key sequences (Patient ages) that could be added to the Patient table and its associated indexes. Put an X next to any/all of the sequences that will lead to an increase in the height of the tree if that sequence of inserts is performed on the state of the tree as shown above.

```
23, 24, 25 _____  50, 64, 65, 66 _____  50, 40 _____
```

(e) (4 pts) How many keys will page P5 contain if the statement DELETE FROM Patient WHERE age = key is executed starting from the state of the tree shown above? Next to each key value below, indicate your answer (i.e., the resulting number of root page key entries) for the deletion of that key.

```
22 _____  25 _____
```
Question 2: Transactions (20 points)

Consider the database for a web application being created to serve lower income patients in a new medical network, Don'tCare, being proposed by the recently elected leaders of a certain country. The primary keys for the two tables, Patient and Doctor, are pid and did, respectively.

<table>
<thead>
<tr>
<th>Patient</th>
<th></th>
<th>Doctor</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>lastname</td>
<td>firstname</td>
</tr>
<tr>
<td>1</td>
<td>Doe</td>
<td>Jane</td>
</tr>
<tr>
<td>2</td>
<td>Smith</td>
<td>Laura</td>
</tr>
<tr>
<td>3</td>
<td>Smith</td>
<td>David</td>
</tr>
<tr>
<td>4</td>
<td>Lee</td>
<td>Richard</td>
</tr>
</tbody>
</table>

Now consider the following SQL transactions that users might wish to run against this table:

**Transaction T1:**
BEGIN TRANSACTION;
INSERT INTO Patient VALUES
(4, ‘Doe’, ‘John’, 21, 1, ‘pains’, 1000);
INSERT INTO Patient VALUES
(5, ‘Deere’, ‘John’, 61, null, ‘aches’, 1000);
UPDATE Patient
SET spouse = 4 WHERE pid = 1;
END TRANSACTION;

**Transaction T2:**
BEGIN TRANSACTION
SELECT COUNT(*)
FROM Patient p, Doctor d
WHERE p.doc = d.did
AND d.name = ‘Frankenstein’;
END TRANSACTION;

(a) (5 pts) Suppose that you were running transaction T1 on your laptop during some after-lecture Q&A time with Professor Carey, but he clumsily spilled a bit of soda on your keyboard and your machine powered down. Argh! After immersing your laptop in rice overnight, you had the database system run recovery and then ran transaction T2 – a query whose answer will depend on how far along the system was with T1 when the failure occurred. Circle all of the possible post-recovery outcomes of T2’s query.

0 1 2 3 4

(b) (5 pts) Now suppose that you were to run transactions T1 and T2 concurrently in different windows on your laptop, and further suppose that you wisely configured your database system’s consistency level to be SERIALIZABLE. Again, circle all of the possible numerical outcomes from T2’s query (depending on how it interleaves with T1).

0 1 2 3 4
**Question 2: Transactions (cont’d.)**

(c) (10 pts) Circle the correct answer(s) to each of the following short questions about transactions.

(i) (2 pts) Logging is used in database systems in support of Isolation, i.e., to ensure that no two concurrent transactions can simultaneously read and update the same database record.

   TRUE          FALSE

(ii) (2 pts) Logging is used in database systems in support of Atomicity, i.e., to ensure that a transaction’s effects on the database can be rolled back (undone) if the transaction fails to commit successfully.

   TRUE          FALSE

(iii) (2 pts) Logging is used in many database systems in support of Durability, i.e., to ensure that a transaction’s effects on the database can be reconstructed (redone) if the transaction commits right before a system crash and the affected file or index pages were still in the buffer pool at failure time.

   TRUE          FALSE

(iv) (2 pts) A schedule is said to be a serializable schedule if and only if all concurrent transactions execute just one at a time in some serial order.

   TRUE          FALSE

(v) (2 pts) Most application developers and database administrators choose to run their database systems with the SQL consistency level set to **SERIALIZABLE** because of the superior performance offered by this level of consistency.

   TRUE          FALSE
Question 3: Physical Database Design (25 points)

Consider the database for a web application being created to serve lower income patients in a new medical network, Don'tCare, being proposed by the recently elected leaders of a certain country:

<table>
<thead>
<tr>
<th>Patient</th>
<th>Doctor</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>did</td>
</tr>
<tr>
<td>1</td>
<td>1002</td>
</tr>
<tr>
<td>2</td>
<td>1001</td>
</tr>
<tr>
<td>3</td>
<td>1002</td>
</tr>
</tbody>
</table>

Assume the relational DBMS that’s been chosen to hold the application’s data stores records in files (one file per table) and does best-effort clustering based on the table’s clustered index (if any) when adding records. (This is similar to what IBM’s DB2 system does.) Thus, the index entries in this system are RID-based. Also assume that the chosen system has both hash-based and B+ tree based indexes, so you have lots of options available. For joins, it supports nested-loops index joins as its preferred algorithm. Finally, assume that the system leaves it to the DBA – that’s you! – to decide what fields to index. It creates no indexes by default, so if you want an index, even on the primary key, it’s up to you to create it.

The following questions give an SQL query template and some suggestions for indexes that might help with the given query. In each case, put an X next to the best suggestion; if the query would run best with several fields being indexed, put an X by each one’s best alternative. Answer each question in isolation; your choices must help with that question’s query. (Also, be careful NOT to pick a clustered index for a given query if clustering won’t help – do not waste clustering!) Lastly, when you see value, that stands for a parameter that the web application will fill in at runtime before sending the query to the DBMS.

(a) (3 pts) SELECT * FROM Patient WHERE pid = value;
- Clustered hashed index on Patient(pid)
- Clustered B+ tree index on Patient(pid)
- No index at all on Patient(pid)

(b) (9 pts) SELECT * FROM Patient p, Doctor d WHERE p.doc = d.did AND d.name LIKE ‘value%’;
- Clustered hashed index on Doctor(name)
- Clustered B+ tree index on Doctor(name)
- No index at all on Doctor(name)
- Clustered hashed index on Patient(doc)
- Clustered B+ tree index on Patient (doc)
- No index at all on Patient (doc)
- Clustered hashed index on Doctor(did)
- Clustered B+ tree index on Doctor(did)
- No index at all on Doctor(did)
Question 3: Physical Database Design (cont’d.)

<table>
<thead>
<tr>
<th>Patient</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Doctor</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>pid</em></td>
<td>lastname</td>
<td>firstname</td>
<td>age</td>
<td>spouse</td>
<td>symptoms</td>
<td>doc</td>
<td><em>did</em></td>
<td>name</td>
<td>specialty</td>
<td>certyear</td>
</tr>
<tr>
<td>1</td>
<td>Doe</td>
<td>Jane</td>
<td>30</td>
<td>null</td>
<td>none</td>
<td>1002</td>
<td>1000</td>
<td>Frankenstein</td>
<td>surgery</td>
<td>1950</td>
</tr>
<tr>
<td>2</td>
<td>Smith</td>
<td>Laura</td>
<td>25</td>
<td>3</td>
<td>chills</td>
<td>1000</td>
<td>1001</td>
<td>Jekyll</td>
<td>dermatology</td>
<td>1994</td>
</tr>
<tr>
<td>3</td>
<td>Smith</td>
<td>David</td>
<td>24</td>
<td>2</td>
<td>fever</td>
<td>1001</td>
<td>1002</td>
<td>Dre</td>
<td>audiology</td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>Lee</td>
<td>Richard</td>
<td>40</td>
<td>null</td>
<td>none</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) (3 pts) **SELECT * FROM Doctor** **WHERE** **name LIKE ‘%value%’**;
- □ Clustered hashed index on Doctor(name)
- □ Unclustered hashed index on Doctor(name)
- □ Clustered B+ tree index on Doctor(name)
- □ Unclustered B+ tree index on Doctor(name)
- □ No index at all on Doctor(name)

(d) (3 pts) **SELECT certyear, COUNT(*) FROM Doctor GROUP BY certyear HAVING certyear > value**;
- □ Clustered hashed index on Doctor(certyear)
- □ Unclustered hashed index on Doctor(certyear)
- □ Clustered B+ tree index on Doctor(certyear)
- □ Unclustered B+ tree index on Doctor(certyear)
- □ No index at all on Doctor(certyear)

(c) (7 pts) **SELECT age, symptoms FROM Patient WHERE symptoms = value**;
- □ Clustered hashed index on Patient(symptoms)
- □ Unclustered hashed index on Patient (symptoms)
- □ Clustered B+ tree index on Patient (symptoms)
- □ Unclustered B+ tree index on Patient (symptoms)
- □ Clustered B+ tree index on Patient (symptoms, age)
- □ Unclustered B+ tree index on Patient (symptoms, age)
- □ Clustered hashed index on Patient (symptoms, age)
- □ Unclustered hashed index on Patient (symptoms, age)
- □ No index at all on Patient
Question 4: NoSQL Databases (25 points)

Suppose that you have a dataverse containing the following datatypes and datasets in Apache AsterixDB.
(The new medical network Don’tCare wants to switch from relational to AsterixDB to increase application
flexibility and also cut costs on software.)

```
CREATE TYPE DoctorType AS {
    did: int,
    name: string,
    specialty: string,
    certyear: int
};

CREATE TYPE PatientType AS {
    pid: int,
    name: {last: string, first: string},
    age: int,
    spouse: int?,
    symptoms: [string],
    doc: int?
};

CREATE DATASET Doctor(DoctorType)
PRIMARY KEY did;

CREATE DATASET Patient(PatientType)
PRIMARY KEY pid;
```

The following SQL++ statements have been successfully run to insert data into the two datasets above:

```
INSERT INTO Doctor ([
    {"did": 1000, "name": "Frankenstein", "specialty": "surgery", "certyear": 1950},
    {"did": 1001, "name": "Jekyll", "specialty": "dermatology", "certyear": 1994},
    {"did": 1002, "name": "Dre", "specialty": "audiology", "certyear": 2000}
]);

INSERT INTO Patient ([
    {"pid": 1, "name": {"last": "Doe", "first": "Jane"}, "age": 30, "symptoms": [], "doc": 1002},
    {"pid": 2, "name": {"last": "Smith", "first": "Laura"}, "age": 25, "spouse": 3, "symptoms": {"chills"},
    "doc": 1000},
    {"pid": 3, "name": {"last": "Smith", "first": "David"}, "age": 24, "spouse": 2, "symptoms": {"fever",
    "rash"}, "doc": 1001},
    {"pid": 4, "name": {"last": "Lee", "first": "Richard"}, "age": 40, "symptoms": [], "doc": 1000}
]);
```

(a) (5 pts) Below are some potential additional Patient records. Indicate for each one, by circling the
appropriate letter to its left, whether an attempt to INSERT that new record into the Patient dataset would
succeed (S) or fail (F) based on the AsterixDB datatype and dataset definitions above.

```
S F {"did": 1003, "name": "Love", "specialty": "cardiology", "certyear": 1995}

S F {"pid": 5, "name": {"last": "Jones", "first": "Joe"}, "age": 35, "symptoms": {"cough"}}

S F {"pid": 6, "name": "Jerry Jones", "age": 37, "spouse": 1, "symptoms": {"sneezing"}, "doc": 1002}

S F {"pid": 7, "name": {"last": "Smith", "first": "Sam"}, "age": 54, "symptoms": {"fever", "nausea"},
    "doc": 1001, "notes": "Having a fever and nausea together kinda sucks"}

S F {"pid": 8, "name": {"last": "Lee", "first": "Leopold"}, "symptoms": {"nausea"}, "doc": 1000}
```
Question 4: NoSQL Databases (cont.)

\[
\begin{array}{|c|c|}
\hline
\text{CREATE TYPE} & \text{CREATE TYPE} \\
\text{DoctorType AS} & \text{PatientType AS} \\
\text{did: int,} & \text{pid: int,} \\
\text{name: string,} & \text{name: \{last: string, first: string\},} \\
\text{specialty: string,} & \text{age: int,} \\
\text{certyear: int} & \text{spouse: int?,} \\
\text{)} & \text{symptoms: \{string\},} \\
\hline
\text{CREATE DATASET} & \text{CREATE DATASET} \\
\text{Doctor(DoctorType)} & \text{Patient(PatientType)} \\
\text{PRIMARY KEY} & \text{PRIMARY KEY} \\
\text{did;} & \text{pid;} \\
\hline
\end{array}
\]

(b) (4 pts) In defining the AsterixDB version of the DonCare schema, the designer used some of the more flexible goodies that are missing in the traditional relational model. Some of their choices have given the new schema a richer “information capacity” – i.e., the AsterixDB design can handle some situations that couldn’t be handled using the relational schema defined elsewhere in the exam. For each of the following situations, indicate whether or not this is additional modeling power (NEW) relative to the earlier SQL design (NOT NEW) due to the AsterixDB schema.

(i) Patients can have several symptoms.
   
   NEW     NOT NEW

(ii) Patients can have both a last name and a first name.
   
   NEW     NOT NEW

(c) (6 pts) App developer Flora Flad data knows SQL and she wants to write a SQL++ query that in DonCare’s older SQL-based relational database system would be written as:

\[
\text{SELECT p.firstname FROM Patient p WHERE p.symptoms = "chills";}
\]

Identify the SQL++ query (or queries) below that will return the same answer once DonCare has migrated to their new AsterixDB database. Circle their query number(s) at the bottom of the list of suggested queries.

\[
\begin{align*}
Q1: & \quad \text{SELECT p.firstname FROM Patient p WHERE p.symptoms = "chills";}
Q2: & \quad \text{SELECT p.name.first FROM Patient p WHERE p.symptoms LIKE "%chills%";}
Q3: & \quad \text{SELECT p.name.first FROM Patient p WHERE "chills" IN p.symptoms;}
Q4: & \quad \text{SELECT p.name.first FROM Patient p WHERE SOME s IN p.symptoms SATISFIES s = "chills";}
Q5: & \quad \text{SELECT p.name.first FROM Patient p WHERE "chills" IN p.symptoms;}
\end{align*}
\]

Equivalent SQL++ query (or queries): \( Q1 \quad Q2 \quad Q3 \quad Q4 \quad Q5 \)
Question 4: NoSQL Databases (cont.)

| CREATE TYPE DoctorType AS {                      | CREATE TYPE PatientType AS {                      |
|       did: int,                                    |       pid: int,                                    |
|       name: string,                               |       name: {last: string, first: string},        |
|       specialty: string,                          |       age: int,                                   |
|       certyear: int                               |       spouse: int?,                               |
| };                                                |       symptoms: [string],                         |
|                                                  |       doc: int?                                   |
|                                                 | };                                               |
| CREATE DATASET Doctor(DoctorType) PRIMARY KEY did; | CREATE DATASET Patient(PatientType) PRIMARY KEY pid; |

(d) (5 pts) Indicate whether each of the following statements are true (T) or false (F) by circling the appropriate letter.

T  F  SQL++ Tutorial author Don Chamberlin co-invented the SQL language in the 1970’s.
T  F  Apache AsterixDB was primarily designed for the purpose of running on mobile devices.
T  F  JSON has essentially “run over” XML because JSON is much more complex and powerful.
T  F  NoSQL databases are named as such because they do not support declarative queries.

(e) (5 pts) App developer Flora Flatdata knows SQL, and she wants to write a query that in Don’tCare’s current SQL-based relational database system would be written as:

```
SELECT * FROM Doctor d
WHERE d.certyear = (SELECT MAX(certyear) FROM Doctor);
```

Flora’s struggling a bit with the flexibility and generality of SQL++. Help her by identifying the SQL++ query that will correctly return the same answer once they’ve migrated to their new AsterixDB database. *Only one of the SQL++ queries listed below will do what Flora wants.* Identify the equivalent SQL++ query and then circle that query’s number at the bottom of the list of suggestions.

Q1: `SELECT * FROM Doctor d
WHERE d.certyear = (SELECT MAX(certyear) FROM Doctor);`

Q2: `SELECT VALUE d FROM Doctor d
WHERE d.certyear = (SELECT VALUE MAX(certyear) FROM Doctor);`

Q3: `SELECT VALUE d FROM Doctor d
WHERE d.certyear = (SELECT VALUE MAX(certyear) FROM Doctor)[0];`

Q4: `SELECT * FROM Doctor d
WHERE d.certyear = (SELECT MAX(certyear) FROM Doctor)[0];`

Q5: `SELECT VALUE d FROM Doctor d
WHERE d.certyear = (SELECT MAX(certyear) FROM Doctor)[0];`

Equivalent SQL++ query (or queries):  Q1   Q2   Q3   Q4   Q5