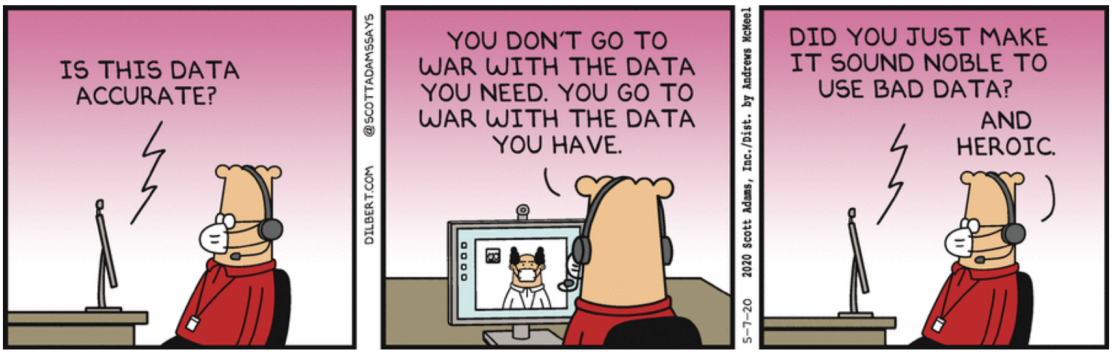
Q1 Preliminaries

1 Point



Instructions

The allowed time for the exam this time is 90 minutes (75 minutes plus a 15-minute technology buffer). Be sure to pay attention to time and budget your time accordingly! (Hopefully you will find this exam to be much less of a race than Midterm 2 was!)

The exam is open pre-prepared cheat sheet, open book, open notes, open web browser, and even open MySQL and open AsterixDB. You are just not allowed to communicate with or otherwise interact with other students (or friends) during the course of the exam, and this includes your HW brainstorming buddy. This exam is to be a solo effort!

Read each question carefully, in its entirety, and then answer each part of the question.

If you don't understand something, make your best guess; if you find ambiguities in a question, note the interpretation that you are taking.

Acknowledgement: I certify that I am taking this exam myself, on my own, with honesty and integrity, without interaction with others during the exam, and without having obtained any information about the exam's content from others prior to taking it.

- (●) True
- ( ) False

A Running Example

Consider the following relational schema, where primary keys are indicated in bold (and some are composite primary keys). Assume that the data is going to be stored in a new open source DBMS called **SQLThis!**. You will recall from Quiz 10 that **SQLThis!** supports both clustered and unclustered B+ tree indexes. It stores its data records in a separate heap file and uses RIDs in index leaf page entries to refer to them. It creates *no* indexes by default - not even on primary keys! - so users must tell it *everything* they want indexed. Here are the tables (with primary keys in bold):

Dept (**dno** integer, dname varchar, budget integer, mgrno integer)
Emp (**eno** integer, ename varchar, age integer, salary integer, dno integer)
Kids (**eno** integer, **kname** string, gender string, allowance integer)

The **SQLThis!** syntax for creating indexes is:

```
CREATE CLUSTERED INDEX ixdname ON tblname(fieldlist);
CREATE UNCLUSTERED INDEX idxname ON tblname (fieldlist);
```

Here is some example relational data (for **SQLThis!**):

Dept(dno, dname, budget, mgrno):
(101, 'Accounting', 75000, 1)
(102, 'Sales', 50000, 3)
(103, 'Legal', 125000, 7)

Emp(eno, ename, age, salary, dno):
(1, 'Dustin', 50, 60000, 101)
(3, 'Emily', 30, 55000, 103)
(5, 'John', 20, 20000, 102)
(7, 'Win', NULL, 45000, 101)
(9, 'Arthur', 30, 40000, 102)

Kids(eno, kname, gender, allowance):
(1, 'Bustin', 'M', 360)
(3, 'Chelsea', 'F', 240)
(9, 'Arthur Jr', 'M', 5000)
(9, 'Othur', 'F', 1000)

Here is the example data in JSON form (for **AsterixDB**):

Dept:

```
{"dno": 101, "dname": "Accounting", "budget": 75000, "mgrno": 1}
{"dno": 102, "dname": "Sales", "budget": 50000, "mgrno": 3}
{"dno": 103, "dname": "Legal", "budget": 125000, "mgrno": 7}
```

Emp:

```
{"eno": 1, "ename": "Dustin", "age": 50, "salary": 60000, "dno": 101
    "kids": [ {"kname": "Bustin", "gender": "M", "allowance": 360} ] }
{"eno": 3, "ename": "Emily", "age": 30, "salary": 55000, "dno": 103
    "kids": [ {"kname": "Chelsea", "gender": "F", "allowance": 240} ] }
{"eno": 5, "ename": "John", "age": 20, "salary": 20000, "dno": 102}
{"eno": 7, "ename": "Win", "age": NULL, "salary": 45000, "dno": 101}
{"eno": 9, "ename": "Arthur", "age": 30, "salary": 40000, "dno": 102
    "kids": [ {"kname": "Arthur Jr", "gender": "M", "allowance": 5000},
              {"kname": "Othur", "gender": "F", "allowance": 1000} ] }
```

NOTE:: You needn't try to load any of this data into anything for this exam. You should be able to look at the questions and data and then answer based on your experience and knowledge. (Plus you would have to download and install **SQLThis!**... :-))

You will find a PDF version of this exam at the end of the attachments section of the CS122a wiki page (https://grape.ics.uci.edu/wiki/asterix/attachment/wiki/cs122a-2020-spring/endterm.pdf) in case you'd like to open a copy in a separate window (to reduce in-exam scrolling).

Q2 Indexing Truth or Consequences

15 Points

For each of the following statements, you should indicate whether the given statement is TRUE or FALSE.

Q2.1

1 Point

If every node in a B+ tree of order d=200 is as empty as the rules for B+ trees permit, a 3-level B+ tree index will have 100 x 100 x 100 = 1,000,000 leaf nodes.

- ○ TRUE
- ◉ FALSE

Q2.2

1 Point

For some queries it can be useful to have a clustered index on a non-key field.

- ◉ TRUE
- ○ FALSE

Q2.3

1 Point

Something that ISAM and B+ tree indexes have in common is the property that the number of page I/Os needed to lookup a key in an index is the same for any key (i.e., all keys are equidistant from the root in terms of the number of pages to be examined).

- ○ TRUE
- ◉ FALSE

Q2.4

1 Point

Something that ISAM and Static Hashed indexes have in common is that both support I/O-efficient exact-match key lookups that are much cheaper than a full scan of the data. (Assume that the key value distribution is well-behaved so that overflow pages are rare for both.)

- [x] TRUE
- [ ] FALSE

Q2.5

1 Point

Consider a large indexed table whose data records occupy N disk pages in a heap file and an **unclustered** B+ tree index whose leaf pages occupy M disk pages (M << N). Ignoring a few non-leaf index page accesses, the maximum I/O cost for a range query that uses this index will be approximately **M + N**.

- [ ] TRUE
- [x] FALSE

Q2.6

1 Point

Consider a large indexed table whose data records occupy N disk pages in a heap file and a **clustered** B+ tree index whose leaf pages occupy M disk pages (M << N). Ignoring a few non-leaf index page accesses, if **f** is the fraction of the range of values covered by a range query, the approximate I/O cost for a range query that uses this index will be **f** * (**M** + **N**).

- [x] TRUE
- [ ] FALSE

Q2.7

1 Point

It is possible to lookup a record in a single I/O via an exact-match query if the table is stored using a primary Static Hashed index whose entries contain the actual data records.

- [x] TRUE
- [ ] FALSE

Q2.8

1 Point

Suppose a workload for our running example contains range queries on both Emp.age and Emp.salary. In this case, we should choose the following physical DB design:

```
CREATE CLUSTERED INDEX cidx1 ON Emp(age);
CREATE CLUSTERED INDEX cidx2 ON Emp(salary);
```

- [ ] TRUE
- [x] FALSE

Q2.9

1 Point

The performance of an index-only query plan will be better if the relevant index is clustered.

- [ ] TRUE
- [x] FALSE

Q2.10

1 Point

The fastest way to build a large B+ tree index with a total of N key entries is to perform a series of N individual B+ tree inserts.

○ TRUE

◉ FALSE

Q2.11

1 Point

Making sure that I/O requests are sequential whenever possible is **more** important when the database storage is based on hard disk (HDD) technology than when it is based on solid-state disk technology (SSD).

◉ TRUE

○ FALSE

Q2.12

1 Point

All operations that a DBMS performs on its data goes through the DBMS's in-memory buffer pool.

◉ TRUE

○ FALSE

Q2.13

1 Point

It is up to the query optimizer to decide which index (if any) to use to process a given query.

◉ TRUE

○ FALSE

Q2.14
1 Point

In most situations involving a real DBMS that is managing real data, the order of its B+ indexes will typically be in the range d < 10.

- ○ TRUE
- ◉ FALSE

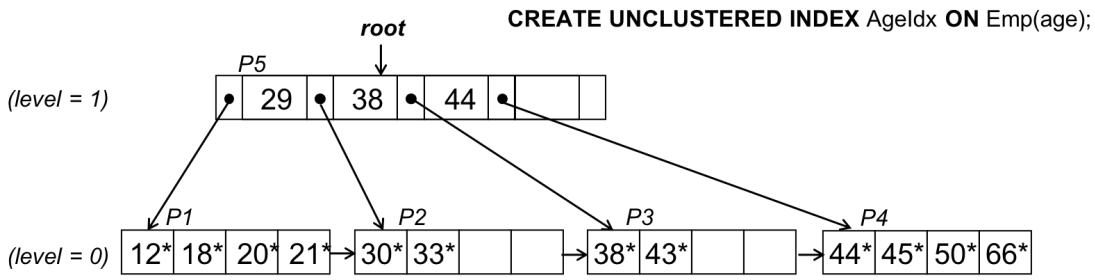
Q2.15
1 Point

An index on a composite key is always built on a set of fields whose original E-R model involved a composite attribute.

- ○ TRUE
- ◉ FALSE

Q3 Save the Trees
24 Points

Below is an example of a B+ tree index. The pages are numbered for reference in the questions that follow. If new pages are needed for an index operation, assume they will come from the end of the file (so their page numbers will continue in sequence). Answer each question as best you can, making use of the indicated notation for index node content since you can't draw pictures on the exam. (You can draw pictures for yourself and then translate the relevant nodes into this notation to give your answer.)



Example index node notation:

```
P5: [P1|29|P2|38|P3|44|P4|--|--]
P1: [12*|18*|20*|21*]
P2: [30*|33*|--|--]
P3: [38*|43*|--|--]
P4: [44*|45*|50*|66*]
```

Q3.1

8 Points

Consider the SQL query

```
SELECT * FROM Emp WHERE age >= 32 AND age < 36
```

Which pages in which order will the DBMS read if it utilizes the B+ tree index shown above? (Answer in list form, e.g., say P4, P2 if your answer is page P4 and then P2.)

P5, P2, P3

Q3.2

8 Points

Show the results of performing the SQL insert operation

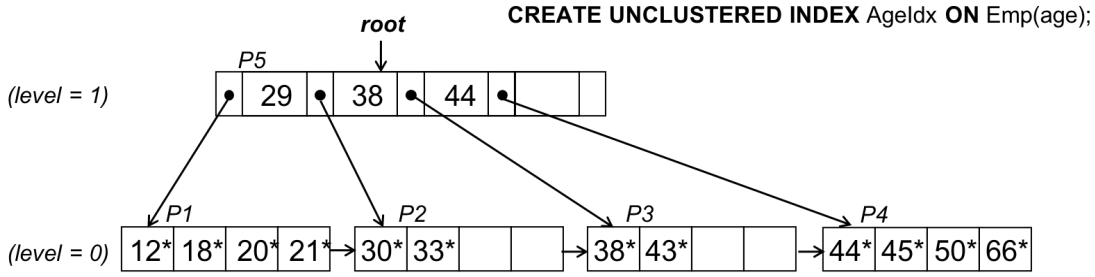
```
INSERT INTO Emp(eno, ename, age, salary, dno)
          VALUES(11, 'Suzi', 69, 60000, 103)
```

using the node notation shown in the figure. Assume the insertion strategy (from lecture) that favors **splitting over redistribution**. You only need to show the nodes whose contents are new or different as a result of the insert.

P5: [P1|29|P2|38|P3|44|P4|66|P6]
P4: [44*|45*|50*|--]
P6: [66*|69*|--|--]
-or-
P5: [P1|29|P2|38|P3|44|P4|50|P6]
P4: [44*|45*|--|--]
P6: [50*|66*|69*|--]

Q3.3

8 Points



Example index node notation:

```
P5: [P1|29|P2|38|P3|44|P4|--|--]
P1: [12*|18*|20*|21*]
P2: [30*|33*|--|--]
P3: [38*|43*|--|--]
P4: [44*|45*|50*|66*]
```

Show the results of performing the SQL delete operation

```
DELETE FROM Emp WHERE age = 38
```

using the node notation shown in the figure. Start from the original tree again (i.e., not from your answer to question 3.2). Assume the deletion strategy (from lecture) that favors **redistribution over merging**. You only need to show the nodes whose contents are new or different as a result of the delete.

P5: [P1|29|P2|38|P3|50|P4|--|--]
P3: [43*|44*|45*|--]
P4: [50*|66*|--|--]
-or-
P5: [P1|29|P2|38|P3|45|P4|--|--]
P3: [43*|44*|--|--]
P4: [45*|50*|66*|--]

Q4 Let's Get Physical

25 Points

SQLThis! supports both clustered and unclustered B+ tree indexes. It stores the data in a heap file and uses RIDs to refer to it. It creates *no* indexes by default - not even on primary keys! - so users must tell it *everything* they want indexed. Here are the tables again for easy reference - assume that they are much bigger than the sample database at the start of the exam :-):

Dept (**dno** integer, dname varchar, budget integer, mgrno integer)
Emp (**eno** integer, ename varchar, age integer, salary integer, dno integer)
Kids (**eno** integer, **kname** string, gender string, allowance integer)

And again, the **SQLThis!** syntax for creating indexes is:

```
CREATE CLUSTERED INDEX ixdname ON tblname(fieldlist);
CREATE UNCLUSTERED INDEX idxname ON tblname (fieldlist);
```

Consider each of the following problems individually - i.e., create the best index (or indexes) for its given SQL query. Express your answer by writing `CREATE ... INDEX` statements. If the given query doesn't warrant creation of an index, answer "No index" in the box (so we can distinguish such answers from questions left unanswered). The **?**'s in the queries represent query parameters that an application will fill in at runtime.

Q4.1

4 Points

```
SELECT * FROM Dept WHERE dno = ?;
```

CREATE UNCLUSTERED INDEX dnoIdx ON Dept(dno);

Q4.2

4 Points

```
SELECT * FROM Dept WHERE budget >= ? AND budget < ?;
```

CREATE CLUSTERED INDEX dnameIdx ON Dept(budget);

Q4.3

8 Points

```
SELECT e.eno, e.ename, e.age
FROM Emp e JOIN Dept d ON e.dno = d.dno
WHERE e.age = ?
  AND d.dname LIKE '%ing';
```

CREATE CLUSTERED INDEX eageIdx ON Emp(age);
CREATE UNCLUSTERED INDEX dnoIdx ON Dept(dno);

Q4.4

8 Points

```
SELECT e.age, MIN(e.salary), AVG(e.salary), MAX(e.salary)
FROM Emp e
GROUP BY e.age;
```

CREATE UNCLUSTERED INDEX covrIdx ON Emp(age, salary);

Q4.5

1 Point

```
SELECT * FROM Kids;
```

No index

Q5 The Sequel to SQL

25 Points

Time to consider your new favorite DBMS, **Apache AsterixDB** of course, and your new favorite query language, **SQL++**. Here again is the NoSQL (JSON) version of the data:

Dept:

```
{"dno": 101, "dname": "Accounting", "budget": 75000, "mgrno": 1}
{"dno": 102, "dname": "Sales", "budget": 50000, "mgrno": 3}
{"dno": 103, "dname": "Legal", "budget": 125000, "mgrno": 7}
```

Emp:

```
{"eno": 1, "ename": "Dustin", "age": 50, "salary": 60000, "dno": 101
    "kids": [ {"kname": "Bustin", "gender": "M", "allowance": 360} ] }
{"eno": 3, "ename": "Emily", "age": 30, "salary": 55000, "dno": 103
    "kids": [ {"kname": "Chelsea", "gender": "F", "allowance": 240} ] }
{"eno": 5, "ename": "John", "age": 20, "salary": 20000, "dno": 102}
{"eno": 7, "ename": "Win", "age": NULL, "salary": 45000, "dno": 101}
{"eno": 9, "ename": "Arthur", "age": 30, "salary": 40000, "dno": 102
    "kids": [ {"kname": "Arthur Jr", "gender": "M", "allowance": 5000},
              {"kname": "Othur", "gender": "F", "allowance": 1000} ] }
```

Q5.1

5 Points

What will the following SQL++ query print given the data above?
(Recall that every SQL++ SELECT query returns an *array of JSON data model instances*, and be sure to include the right fields and field names in any JSON objects in your answer.)

```
SELECT d
FROM Dept d
WHERE d.dname LIKE '%ing;
```

{"d": {"dno": 101, "dname": "Accounting", "budget": 75000, "mgrno": 1}}

Q5.2

8 Points

What will the following SQL++ query print given the data above?
(Recall that every SQL++ SELECT query returns an *array of JSON data model instances*, and be sure to include the right field names in any JSON objects in your answer.)

```
SELECT *
FROM Emp e, Dept d
WHERE d.dno = e.dno
ORDER BY e.salary ASC
LIMIT 1;
```

{"e": {"eno": 5, "ename": "John", "age": 20, "salary": 20000, "dno": 102},
"d": {"dno": 102, "dname": "Sales", "budget": 50000, "mgrno": 3}}

Q5.3

8 Points

Write a SQL++ query to return the names and genders of Arthur's kids:

SELECT k.kname, k.gender
FROM Emp e UNNEST e.kids k
WHERE e.ename = "Arthur";

Q5.4

4 Points

Consider (carefully, due to logic's treatment of empty sets) the following SQL++ query:

```
SELECT e.ename
FROM EMP e
WHERE EVERY k IN e.kids SATISFIES k.allowance >= 1000;
```

Which employee names will be returned?

- [ ] Dustin
- [ ] Emily
- [x] John
- [x] Win
- [x] Arthur

Q6 Would ACID Kill Coronavirus?

8 Points

You've heard that **SQLThis!** supports the full gamut of SQL transaction options, which is attractive to you. Here once again is the relational version of our data:

Dept(dno, dname, budget, mgrno):
(101, 'Accounting', 75000, 1)
(102, 'Sales', 50000, 3)
(103, 'Legal', 125000, 7)

Emp(eno, ename, age, salary, dno):
(1, 'Dustin', 50, 60000, 101)
(3, 'Emily', 30, 55000, 103)
(5, 'John', 20, 20000, 102)
(7, 'Win', NULL, 45000, 101)
(9, 'Arthur', 30, 40000, 102)

Kids(eno, kname, gender, allowance):
(1, 'Bustin', 'M', 360)
(3, 'Chelsea', 'F', 240)
(9, 'Arthur Jr', 'M', 5000)
(9, 'Othur', 'F', 1000)

Q6.1

2 Points

Transaction T1 contains the following SQL statements:

```
BEGIN;
   UPDATE Emp SET salary = salary + 5000 WHERE eno = 9;
   UPDATE Kids SET kname = 'Artie' WHERE gender = 'M';
COMMIT;
```

Transaction T2 contains the following SQL statements:

```
BEGIN;
   SELECT e.ename, e.salary, k.kname
   FROM Emp e JOIN Kids k ON e.eno = k.eno;
COMMIT;
```

Can T2 produce ('Arthur', 45000, 'Arthur Jr') as a result if these two transactions are executed concurrently and T2's SQL isolation level is SERIALIZABLE?

- ○ Yes
- ◉ No

Q6.2

2 Points

Can T2 produce ('Arthur', 45000, 'Arthur Jr') as a result if these two transactions are executed concurrently and T2's SQL isolation level is READ UNCOMMITTED?

- ◉ Yes
- ○ No

Q6.3

2 Points

The ACID transaction concept and acronym is the single most important take-away from the transactions lecture in CS122a. A is for Atomicity (*a.k.a.* all or nothing), C is for Consistency, I is for Isolation, and D is for Durability. For which

one of these letters is two-phase locking the most important implementation mechanism?

- ○ A(tomicity)
- ○ C(onsistency)
- ◉ I(solation)
- ○ D(urability)

Q6.4
2 Points

For which two of the ACID letters is write-ahead logging the most important implementation mechanism?

- ☑ A(tomicity)
- ☐ C(onsistency)
- ☐ I(solation)
- ☑ D(urability)

Q7 You Made It!
2 Points

Hey, I made it to the end of the CS122a Endterm: **TGIF** of Week 10...!

- ◉ True that!

Endterm

GRADED

1 DAY, 21 HOURS LATE

STUDENT

Unknown Student (removed from roster?)

TOTAL POINTS

100 / 100 pts

QUESTION 1

Preliminaries — **1** / 1 pt

QUESTION 2

Indexing Truth or Consequences — **15** / 15 pts

2.1	(no title)	**1** / 1 pt
2.2	(no title)	**1** / 1 pt
2.3	(no title)	**1** / 1 pt
2.4	(no title)	**1** / 1 pt
2.5	(no title)	**1** / 1 pt
2.6	(no title)	**1** / 1 pt
2.7	(no title)	**1** / 1 pt
2.8	(no title)	**1** / 1 pt
2.9	(no title)	**1** / 1 pt
2.10	(no title)	**1** / 1 pt
2.11	(no title)	**1** / 1 pt
2.12	(no title)	**1** / 1 pt
2.13	(no title)	**1** / 1 pt
2.14	(no title)	**1** / 1 pt
2.15	(no title)	**1** / 1 pt

QUESTION 3

Save the Trees — **24** / 24 pts

3.1	(no title)	**8** / 8 pts
3.2	(no title)	**8** / 8 pts
3.3	(no title)	**8** / 8 pts

QUESTION 4

Let's Get Physical — **25** / 25 pts

4.1	(no title)	**4** / 4 pts
4.2	(no title)	**4** / 4 pts
4.3	(no title)	**8** / 8 pts
4.4	(no title)	**8** / 8 pts
4.5	(no title)	**1** / 1 pt

QUESTION 5

The Sequel to SQL — **25** / 25 pts

5.1	(no title)	**5** / 5 pts
5.2	(no title)	**8** / 8 pts
5.3	(no title)	**8** / 8 pts
5.4	(no title)	**4** / 4 pts

QUESTION 6

Would ACID Kill Coronavirus? — **8** / 8 pts

6.1	(no title)	**2** / 2 pts
6.2	(no title)	**2** / 2 pts
6.3	(no title)	**2** / 2 pts
6.4	(no title)	**2** / 2 pts

QUESTION 7

You Made It! — **2** / 2 pts