Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.830 Database Systems: Fall 2017 Quiz I

There are 15 questions and 12 pages in this quiz booklet. To receive credit for a question, answer it according to the instructions given. *You can receive partial credit on questions.* You have **80 minutes** to answer the questions.
xxx

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
LAPTOPS MAY BE USED; NO PHONES OR INTERNET ALLOWED.

Do not write in the boxes below

1-6 (xx/36)	7-9 (xx/20)	10-13 (xx/26)	14-15 (xx/18)	Total (xx/100)

Name:

I Column Stores

Suppose you have a table T with 100 columns, c_1 through c_{100}, each of which contains random integers, distributed uniformly between 1 and 100. Initially the table is unsorted and uncompressed. Your disk can scan 100 MB/sec, and data is divided into 10 KB pages.

This table is 1000 MB, and you run the query:

```
SELECT c1, c2, c3
FROM T
WHERE c4 > 90
```

1. [4 points]: Ignoring seek times and CPU costs, how long do you expect it would take to run this query in a row store such as SimpleDB:
(Choose the best answer.)

[✓ A.] 10.00 seconds

B. 4.00 seconds

C. 0.40 seconds

D. 0.31 seconds

E. 0.04 seconds

F. None of the above

Answer: A is the correct answer. We have to read the whole table. 1000MB at 100MB/sec is 10 sec.

2. [4 points]: Ignoring seek times and CPU costs, how long do you expect it would take to run this query in a column store using an early materialization design as studied in class:
(Choose the best answer.)

A. 10.00 second

B. 4.00 seconds

[✓ C.] 0.40 seconds

D. 0.31 seconds

E. 0.04 seconds

F. None of the above

Answer: C is the correct answer. We have to read only 4 of the 100 columns. So now the scan time is .4 seconds.

Name:

3. [6 points]: Ignoring seek times and CPU costs, how long do you expect it would take to run this query in a column store using a late materialization design as studied in class (assume that the position bitmaps can fit into memory):

(Choose the best answer.)

A. 10.00 seconds

B. 4.00 seconds

[✓ C.] 0.40 seconds

D. 0.31 seconds

E. 0.04 seconds

F. None of the above

Answer: C is the correct answer. Late materialization doesn't help here because the non-zero positions are scattered throughout the table, so all 4 columns need to be scanned in their entirety. So the scan time is still .4 seconds.

4. [6 points]: Suppose you sort the table on column c_4, and create an index on the column so you can directly offset to the first record in c_4 satisfying the predicate. Ignoring seek times and CPU costs, how long do you expect it would take to run this query in a column store using a late materialization design as studied in class (assume the position bitmaps and index are in memory):

(Choose the best answer.)

A. 10.00 seconds

B. 4.00 seconds

C. 0.40 seconds

D. 0.31 seconds

[✓ E.] 0.04 seconds

F. None of the above

Answer: E is the correct answer. We read 4 of the 100 columns, but we can offset directly to the first record satisfying the predicate in c_4 , and the satisfying positions will be grouped together in the remaining columns. Hence, we only need to scan 10% of what we scanned before, so the scan time is .04 seconds.

Name:

II Joins and Join Ordering

5. [6 points]: Consider joining three tables, A, B, and C on a database system with the following join algorithms: nested loops, in-memory hash and grace hash, in-memory sort-merge and external sort merge, with join predicates between all three pairs of tables. There are no indexes, and no selection predicates, and the tables are not sorted on any of the join attributes.

You find that the cost of running $(A \bowtie B) \bowtie C$ is much less than the cost of running $(A \bowtie C) \bowtie B$, even though B occupies more bytes on disk and has more records than C.

Give one possible explanation for how this could be the case:

(Write your answer in the space below.)

Answer: 1. The cardinality of $(A \bowtie B)$ could be less than $(A \bowtie C)$.
2. The join of $(A \bowtie B)$ could produce an interesting ordering.

6. [10 points]: Consider the Selinger optimizer's algorithm for ordering joins, running on four tables, A, B, C and D. Here there all joins are nested loops joins, and there are no interesting orders. Suppose the Selinger cost model estimates that the optimal way to join A, B and C is $(A \bowtie B) \bowtie C$, and that the optimal way to join A, B, and D is $(A \bowtie D) \bowtie B$. Assume all tables are large and different sizes.

Which of the following plans could the Selinger optimizer choose as the overall optimal ordering?

(Circle 'T' or 'F' for each choice.)

T F $((A \bowtie D) \bowtie B) \bowtie C$

T F $((A \bowtie B) \bowtie D) \bowtie C$

T F $((A \bowtie D) \bowtie C) \bowtie B$

T F $(A \bowtie B) \bowtie (C \bowtie D)$

T F $((B \bowtie C) \bowtie D) \bowtie A$

Answer: T $(A \bowtie D) \bowtie B)$ was shown to be optimal.
F False for the reason above was True. $(A \bowtie D) \bowtie B)$ better than $(A \bowtie B) \bowtie D)$
T This possibility is not ruled out by the optimal joins of the subsets given so it is possible.
F The Selinger optimizer does not consider "bushy" plans.
T This possibility is not ruled out by the optimal joins of the subsets given so it is possible.

Name:

III Entity and Schema Design

You are developing a note-taking application (like Evernote / Onenote). Your database needs to store the following information:

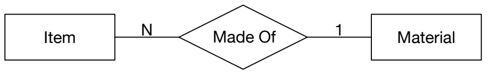
- For each user, their id, name, and email address.
- For each note, its id, title, content, last updated date.
- For each notebook, its id, title.

In addition, your database should represent the following relationships:

- Each user can have multiple notebooks.
- Each note is part of a notebook.
- Each notebook belongs to a unique user.

Although users write notes, the authorship of a note is implicit from the ownership of the notebook in this design.

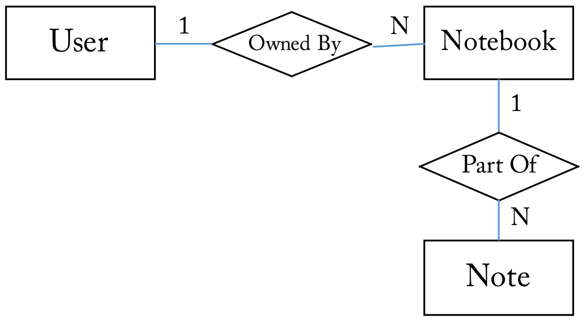
7. [8 points]: Draw an entity relationship diagram for this database. Please draw entities as squares, attributes as ovals, and denote relationships as diamonds between pairs of entities. Label each edge with a "1" or an "N" to indicate whether the entity on the other side of the relationship connects to 1 or N entities on this side of the relationship. For example, the following would indicate that each item is made of a single material (e.g. wood, metal), and multiple items can be made of the same material:



Give each entity, relationship, and attribute a name.

(Draw your diagram in the space below)

.



Name:

8. [6 points]: Use your ER diagram to determine a relational schema for this database. For each table, use the form:

TablenameN (field1-name, ..., fieldn-name)

to denote its schema. If you wish, you can create an additional field for each table to serve as a unique identifier. Underline the primary keys and use the "... references ..." syntax for foreign keys in your schema.

(Write your answer in the space below.)

User (id, name, email)
Notebook (id, title, owner_id references (user.id))
Note (id, title, content, last_updated, notebook_id references (notebook.id))

9. [6 points]: As your application grows in popularity, users want to be able to share their notes and notebooks with their friends/colleagues. What changes do you need to make to your relational schema support this feature?

(Write your modified relational schema below.)

User (id, name, email)
Notebook (id, title)
OwnedBy (user_id references (user.id), notebook_id references (notebook.id))
Note (id, title, content, last_updated)
PartOf (notebook_id references (notebook.id), note_id references (note.id))

For the sharing feature to work, each notebook can belong to multiple users and each note can belong to multiple notebooks.

IV Postgres Plans

Here we use the IMDB dataset from PS2 and study a few query plans. The schemas relevant for this question are listed below for you.

```
     Table "public.movies"
 Column  |  Type   | Modifiers
---------+---------+-----------
 id      | text    | not null
 title   | text    |
 year    | integer |
 runtime | integer |
Indexes:
    "movies_pkey" PRIMARY KEY, btree (id)
    "movies_title" btree (title)
Referenced by:
    TABLE "cast_members"
        CONSTRAINT "cast_members_movie_id_fkey"
        FOREIGN KEY (movie_id) REFERENCES movies(id)
    TABLE "directors"
        CONSTRAINT "directors_movie_id_fkey"
        FOREIGN KEY (movie_id) REFERENCES movies(id)
    TABLE "ratings"
        CONSTRAINT "ratings_movie_id_fkey"
        FOREIGN KEY (movie_id) REFERENCES movies(id)


      Table "public.people"
   Column   |  Type   | Modifiers
------------+---------+-----------
 id         | text    | not null
 name       | text    |
 birth_year | integer |
 death_year | integer |
Indexes:
    "people_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "cast_members"
        CONSTRAINT "cast_members_person_id_fkey"
        FOREIGN KEY (person_id) REFERENCES people(id)
    TABLE "directors"
        CONSTRAINT "directors_person_id_fkey"
        FOREIGN KEY (person_id) REFERENCES people(id)
```

Name:

Suppose that we're interested in finding all the pairs of movie titles and people's names for which the person was born in the same year as the movie was released, for movies starting with 'y' or 'z' in the alphabet and people whose names come after "Zeke" in the alphabet. Consider the following query and its query plan from Postgres:

```
EXPLAIN ANALYZE SELECT movies.year, movies.title, people.name
    FROM movies
    JOIN people
        ON movies.year=people.birth_year
    WHERE people.name > 'zeke' AND
        movies.title > 'y';

                                   QUERY PLAN
-------------------------------------------------------------------------------------
 Merge Join  (cost=31843.55..32194.92 rows=30215 width=36)
             (actual time=496.720..510.071 rows=38381 loops=1)
   Merge Cond: (movies.year = people.birth_year)
   ->  Sort  (cost=9905.45..9918.62 rows=5268 width=22)
             (actual time=151.781..152.690 rows=5634 loops=1)
         Sort Key: movies.year
         Sort Method: quicksort  Memory: 729kB
         ->  Seq Scan on movies  (cost=0.00..9579.81 rows=5268 width=22)
                                 (actual time=145.826..149.340 rows=7640 loops=1)
               Filter: (title > 'y'::text)
               Rows Removed by Filter: 456425
   ->  Sort  (cost=21936.87..21953.89 rows=6808 width=18)
             (actual time=344.918..347.980 rows=38465 loops=1)
         Sort Key: people.birth_year
         Sort Method: quicksort  Memory: 423kB
         ->  Seq Scan on people  (cost=0.00..21503.44 rows=6808 width=18)
                                 (actual time=341.883..343.847 rows=4151 loops=1)
               Filter: (name > 'zeke'::text)
               Rows Removed by Filter: 1099324
 Planning time: 0.450 ms
 Execution time: 511.988 ms
```

Note that the working memory size used for these queries is 200MB. Also note the following relevant stats:

- `movies` has a cardinality of 464065

- `people` has a cardinality of 1103475

10. [4 points]: What selectivity does the planner estimate for the predicate `movies.title > 'y'`? **(Write your answer in the space below.)**

5268/464065 (or equivalent)

This is the estimated number of rows after the filter on `movies.title > 'y'`, divided by the cardinality of the movies table.

Name:

11. [8 points]: Why do you think the query optimizer chose a sort-merge join?
(Write your answer in the space below.)

First, we know that the filtered results fit in-memory, so in-memory sort-merge join will be better than any external join.

Second, given the sizes of the filtered tables are relatively similar, sort-merge join will be faster than nested loop join.

Lastly, the query planner must think that sort-merge join is better than hash join because the movies.year and people.birth_year columns will have lots of duplicates, so a hash on either of those columns would result in long chains (and thus would not achieve the usual efficiency of a hash table).

12. [8 points]: For the following questions, mark true if the following operation would likely significantly improve the performance of the above query, and false if it would not.
(Circle 'T' or 'F' for each choice.)

T F Increase the working memory size (i.e., memory available for intermediate hash tables and join buffers.)

T F Cluster the `movies` on the index `movies_title`

T F Create a clustered index on `people.name`

T F Create a clustered index on `movies_year`

F - The filtered tables already fit in memory (which you can see by the memory size used by sorting), so adding more memory won't improve performance.

T - The predicate is highly selective, but the index is not being used because it is unclustered, so clustering on the index would allow an efficient index scan rather than needing to use a seq scan.

T - The predicate is highly selective, and therefore a clustered index would allow an efficient index scan rather than a seq scan as the access method.

T - A clustered index on `movies_year` would mean that the movies table was already sorted for the sort-merge join, thus skipping the expensive sort step.

Name:

Now suppose that instead we are only interested in people with names that come after 'Zeke' in the alphabet and who were born in the same year that 'The Dark Knight' was released.

```
EXPLAIN ANALYZE SELECT movies.year, movies.title, people.name
    FROM movies
    JOIN people
        ON movies.year=people.birth_year
    WHERE people.name > 'zeke' AND
        movies.title = 'The Dark Knight';
                            QUERY PLAN
---------------------------------------------------------------
 Nested Loop  (cost=0.42..21596.98 rows=6 width=36)
              (actual time=346.798..348.031 rows=1 loops=1)
   Join Filter: (movies.year = people.birth_year)
   Rows Removed by Join Filter: 4150
   ->  Index Scan using movies_title on movies
             (cost=0.42..8.44 rows=1 width=22)
             (actual time=0.045..0.054 rows=1 loops=1)
         Index Cond: (title = 'The Dark Knight'::text)
   ->  Seq Scan on people  (cost=0.00..21503.44 rows=6808 width=18)
                           (actual time=345.728..347.495 rows=4151 loops=1)
         Filter: (name > 'zeke'::text)
         Rows Removed by Filter: 1099324
 Planning time: 0.309 ms
 Execution time: 348.080 ms
```

13. [6 points]: Note that the optimizer switched from a sort-merge join to a nested loops join. Why do you think the query optimizer switched join methods?

(Write your answer in the space below.)

Since the estimated cardinality of the filtered result of the predicate on movies is 1, the nested loop join would only require one sequential pass through the filtered people result, which would therefore have a faster complexity than the $O(nlog(n))$ sorting required for sort-merge join.

Name:

V Indexing

You have a database of US Patents consisting of the following tables:

```
Inventors(inventor_id, first_name, last_name, alma_mater, birth_state, birth_year)
Patents(patent_id, inventor_id, patent_name, patent_year)
```

Assume the following:

- There are 2,000,000 patents and 2,000,000 inventors; there is only 1 inventor for each patent, and each inventor has 1 patent.
- Patents are uniformly distributed across all years from 1900–2017
- 50% of all inventors have MIT as their alma mater. .0001% of inventors have Harvard as their alma mater.
- Inventors birth years are evenly distributed from 1895–2014 (some precocious youth).
- There are 100,000 different first names evenly distributed across all inventors.
- Disk seeks take 1 ms, and scanning either table takes 100 secs.
- For B+Trees, only leaf pages are not cached, and leaf pages contain pointers to heap file tuples, with 1000 pointers per leaf page. Each heap page also contains 1000 records.

You may only create 1 clustered index for each table, but can create as many unclustered indexes as you like. Clustering implies sorting the pages of the heap file according to the index.

Consider the query:

```
SELECT inventors.first_name,
       inventors.alma_mater,
       patents.patent_name
FROM patents
JOIN inventors ON patents.inventor_id = inventors.inventor_id
WHERE inventor.alma_mater = 'MIT'
  AND inventor.birth_year > 2013
  AND patents.patent_year > 1905
```

14. [10 points]: Would you create any indexes to speed up this query? Which and why?
(Write your answer in the space below.)

Answer: Note that the answer to this question was updated 10/27/2017.

This question is tricky. Clearly, a clustered index on inventor.birth_year will dramatically reduce the number of records that need to be scanned from inventor, since just 1/120 records will satisfy this. Adding alma_mater as a 2ndardy index key will help since it will eliminate scanning 50% of the remaining records.

Creating an unclustered index on inventors.inventor_id will be best for the join, since there are just 2M/120x2=8,333 records that satisfy the predicates on patents, so the index lookups will take about

16 seconds, which is better than building a clustered index on patents.inventor_id or which will require nearly a full sequential scan.

Finally, we can do slightly better by doing an index-only join by including first_name in the inventors index and patent_name and patent_year in the patents indexes, allowing us to avoid any I/Os from the B+tree to the heap file.

Now suppose you want to run the three following queries with equal frequency.

```
SELECT patent_name
FROM patents
JOIN inventors ON inventors.inventor_id = patents.inventor_id
WHERE patents.patent_year > '2015'
  AND inventors.birth_year > '2012'
ORDER BY inventors.birth_year

SELECT patent_name,
       patent_year
FROM patents
WHERE patent_name =
       "System and apparatus for computing the
      Ultimate Question of Life,The Universe, and Everything"

SELECT patent_name,
         patent_year
FROM patents
JOIN inventors ON inventors.inventor_id = patents.inventor_id
WHERE inventors.alma_mater = 'Harvard'
```

15. [8 points]: What indexes, if any, would you create? Would they be clustered or unclustered? Why? **(Write your answer in the space below.)**

Answer: There are specific indexes that are good for each query.

For Q1, because both predicates are relatively selective, a clustered index on patent_year and birth_year will be effective.

For Q2, an unclustered index on patent_name will work well, because presumably there is only one patent with that name. Adding patent_year to that index will allow this query to be run as an index-only query.

For Q3, an unclustered index on alma_mater will work well, because there are very few patents from Harvard (just 2). Also, an unclustered index on patents.inventor_id will eliminate 2 sequential scans of patents, replacing them with 2 index lookups.

End of Quiz I!

Name: