CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2026)
PROF. JIGNESH PATEL

Homework #2 (by Drew Ripberger) – Solutions

Due: **Sunday Sept 20, 2026 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday Sept 20, 2026.**
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.
- No AI may be used to complete homework.

For your information:

- Graded out of **100** points; **3** questions total
- Rough time estimate: ≈4-6 hours (1-1.5 hours for each question)

Revision : 2026/09/20 23:58

Question	Points	Score
Slotted Pages and Log-Structured	30	
Storage Models	35	
Database Compression	35	
Total:	100	

Question 1: Slotted Pages and Log-Structured . . . . . [30 points]

Graded by:

(a) **[10 points]** Which of the following statements are true for database systems using *log-structured storage*? Select all that apply.

□ Log structured storage doesn't experience write amplification

■ **Log structured storage allows for update operations that don't require accessing previous log records**

■ **Log structured storage typically doesn't experience fragmentation**

□ In leveled compaction, SSTables within a level are always non-overlapping on key ranges

□ None of the above

Solution: Log-structure storage incurs write amplification due to compaction.

Log-structured storage does not require the DBMS to check previous log records on updates, instead the DBMS simply appends log records to the end of the file.

Fragmentation is a problem typically associated with slotted-page storage due to tuple deletion in a page leaving holes. This is not associated with log-structured storage.

Levelled compaction has non-overlapping key-ranges for SSTables on each level below the 0th level. The 0th level may have overlapping key ranges.

(b) **[10 points]** Which of the following statements are true for database systems using *slotted-page storage*? Select all that apply.

■ **Slotted-page storage can increase random reads and writes**

□ Applications should treat RIDs in slotted-page storage systems as stable identifiers and may rely on them

□ Slotted-page storage does not allow variable length tuples

□ Slotted-page storage can cause write amplification

□ None of the above

Solution: Since tuples can be stored across separate pages, it may increase the amount of random I/O that the DBMS has to incur when both reading data and when writing out dirty pages.

RIDs should never be relied on as a stable identifier, since they are simply used by the DBMS to locate a tuple on disk.

Slotted page storage does allow for variable length tuples, by simply storing the start byte of a tuple in the slot array.

Write amplification is a problem typically associated with log-structured storage, not slotted-page storage.

(c) **[10 points]** You are asked to compare *log-structured storage* to *slotted-page storage* for

a new system. Ignore any indexes and overhead from metadata. Select all true statements.

□ Both storage systems update tuples in-place to avoid extra I/O

□ Slotted-page requires storing schema metadata with the tuple, while log-storage does not

■ **For an append-only workload, both achieve comparable performance**

□ Log-structured storage requires less disk space than slotted-page storage

■ **Both designs may require background maintenance to reclaim space after numerous inserts/updates/deletes**

□ None of the above are true

Solution: Log-structured storage does not update tuples in-place, and always inserts a new tuple to the log.

Neither format requires storing schema metadata with the tuple

In absence of indexes + metadata, then slotted-page for append-only workload becomes the log-structured storage architecture. They achieve similar performance.

Log structured storage requires more disk space in workloads that are not append-only.

After lots of inserts/updates/deletes, slotted-page storage may benefit from maintenance to reclaim any empty space or compact partially empty pages. Log-structured storage may benefit from compaction.

Question 2: Storage Models . . . . . [35 points]

Graded by:

Consider a database with a single table `F(player_id,team_id,position,salary)`, where `player_id` is the *primary key*, and all attributes are the same fixed width. Suppose `F` has 15,000 tuples that fit into 200 pages. You should ignore any additional storage overhead for the table (e.g., page headers, tuple headers). Additionally, you should make the following assumptions:

- The DBMS does *not* have any additional meta-data.
- `F` does *not* have any indexes (including for primary key `player_id`).
- None of `F`'s pages are already in memory. The DBMS can store an infinite number of pages in memory.
- Content-wise, the tuples of `F` will always make each query run the longest possible and do the most page accesses. Always consider what the worst-case *content/values* for each column can be.
- Order-wise, the tuples of `F` can be in any order. Keep this in mind when computing *minimum* versus *maximum* number of pages that the DBMS will potentially have to read. Think of all possible orderings, but always for the worst-case content of data

(a) Consider the following query:

```
SELECT MAX(salary) FROM F
    WHERE team_id = 15445 ;
```

i. **[5 points]** Suppose the DBMS uses the N-ary storage model (NSM). How many pages will the DBMS potentially have to read from disk to answer this query?
Be sure to keep in mind the assumption about the contents of `F`.
☐ 1-50 ☐ 51-100 ☐ 101-150 ■ **151-200** ☐ > 200 ☐ Not possible to determine

Solution: 200 pages. To find `salary` and `team_id` for all tuples, all pages must be accessed.

ii. **[5 points]** Suppose the DBMS uses the decomposition storage model (DSM) with implicit offsets. How many pages will the DBMS potentially have to read from disk to answer this query?
Be sure to keep in mind the assumption about the contents of `F`.
☐ 1-50 ■ **51-100** ☐ 101-150 ☐ 151-200 ☐ > 200 ☐ Not possible to determine

Solution: 100 pages. There are 50 pages per attribute. 50 pages to find `team_id` for all tuples. In the worst-case scenario for `F`'s content, `salary` for all tuples must be accessed as well. Hence, another 50 pages must be read.

(b) Now consider the following query:

```
SELECT salary, team_id FROM F
   WHERE player_id = 1 OR player_id = 999
```

i. Suppose the DBMS uses the N-ary storage model (NSM).

α) **[5 points]** What is the *minimum* number of pages that the DBMS will potentially have to read from disk to answer this query?

■ **1** ☐ 2-9 ☐ 10-50 ☐ 51-100 ☐ 101-150 ☐ 151-200
☐ > 200 ☐ Not possible to determine

Solution: We find the tuples of all two primary keys on the first page. No need to look in other pages since all attributes are stored together.

β) **[5 points]** What is the *maximum* number of pages that the DBMS will potentially have to read from disk to answer this query?

☐ 1 ☐ 2-9 ☐ 10-50 ☐ 51-100 ☐ 101-150 ■ **151-200**
☐ > 200 ☐ Not possible to determine

Solution: 200 pages. At least one tuple with matching primary key is either located on the last page or it can't be found in any of the pages. We must thus scan through every page.

ii. Suppose the DBMS uses the decomposition storage model (DSM) with implicit offsets.

α) **[5 points]** What is the *minimum* number of pages that the DBMS will potentially have to read from disk to answer this query?

☐ 1 ■ **2-9** ☐ 10-50 ☐ 51-100 ☐ 101-150 ☐ 151-200
☐ > 200 ☐ Not possible to determine

Solution: 3 pages. Suppose all two primary keys appear on the first page. Since all attributes are of the same fixed width, each attribute of `player_id=1` and `player_id=999` will also appear on the same page. We'll thus need to read 1 page to find both primary keys and read 2 pages to access `salary` and `team_id` at their corresponding offsets.

β) **[5 points]** What is the *maximum* number of pages that the DBMS will potentially have to read from disk to answer this query?

☐ 1 ☐ 2-9 ☐ 10-50 ■ **51-100** ☐ 101-150 ☐ 151-200
☐ > 200 ☐ Not possible to determine

Solution: 54 pages. There are 50 pages per attribute. In the worst case, we scan through all 50 pages to find both primary keys. In the worst case, the primary keys will be located on different pages. Since all attributes are of the

same fixed width, each attribute of `player_id=1` and `player_id=999` will also appear on different pages. Hence we must read 2 pages to access each attribute at their corresponding offsets. Thus, we read 4 pages in total to access `team_id` and `salary`.

(c) Finally consider the following query:

```
SELECT MIN(player_id) FROM F
    WHERE salary = (SELECT MIN(salary) FROM F);
```

Suppose the DBMS uses the decomposition storage model (DSM) with implicit offsets.

i. **[5 points]** What is the *minimum* number of pages that the DBMS will potentially have to **read from disk** to answer this query? Keep in mind the assumptions we make about our DBMS.

☐ 1 ☐ 2-9 ☐ 10-50 ■ **51-100** ☐ 101-150 ☐ 151-200 ☐ > 200
☐ Not possible to determine

Solution: 100 pages. 50 pages for the inner select, and 50 pages to get the `player_id` since the buffer pool will have the `salary` pages from the inner select. Remember content-wise the tuples make the queries always run for the longest time, you can only consider different orderings of the tuples.

Question 3: Database Compression . . . . . [35 points]

Graded by:

(a) **[5 points]** Suppose that the DBMS has a `VARCHAR` column storing the following values:

```
[Banana, Apple, Orange, Pawpaw, Cherry]
```

Which of the following are valid encodings (`uint32`) for this column **under dictionary** compression as discussed in lecture that will support both point queries and range queries? Select **all** the valid encodings.

- ☐ `[2,1,3,5,0]`
- ☐ `[0,12,32,33,31]`
- ■ `[8,3,15,9891,11]`
- ☐ `[50,40,20,10,30]`
- ■ `[12,5,29,30,19]`
- ☐ `[1,2,3,4,5]`

Solution: To support range queries, the DBMS must use an order-preserving encoding scheme. The values of the dictionary codes do not matter as long as they preserve the same ordering of the original data.

(b) **[15 points]** Suppose the DBMS wants to compresses a table `R(a)` using columnar compression. Which of the following compression schemes **will benefit** (when considering space efficiency) from sorting the table before compressing column `a`? Select **all** that apply.

Hint: "Benefit" means that the efficacy of the compression scheme may improve on sorted data. You should not make any assumptions about the column type or its distribution of values.

- ■ **Run-length Encoding**
- ☐ Bit-packing Encoding
- ☐ Mostly Encoding
- ☐ Bitmap Encoding
- ☐ Dictionary Encoding
- ■ **Delta Encoding**
- ☐ None of the above will benefit.

Solution: Sorting only benefits Run-length encoding and Delta encoding for the below reasons. All other encodings do not benefit from sorting the table first.

- **Run-length Encoding:** Sorting improves the potential compression ratio for RLE because there could potentially be more consecutive values in a column.
- **Delta Encoding:** For numeric data types with a small range of values, the difference between consecutive values in the column after sorting could be smaller than the original value. Therefore, the compression ratio could improve.

(c) **[15 points]** A colleague approaches with a list of true and false statements about run-length encoding, delta encoding, bitmap encoding, and dictionary encoding. The colleague wants your assistance in identifying the true statements. Select **all** that apply.

■ **Delta Encoding is good at compressing predicatable time series data.**

□ For a *point lookup-only* workload, order-preserving dictionary encoding is required.

□ Run Length Encoding is most effective on unsorted uniformly random data.

■ **Run-length Encoding is effective for compressing a low cardinality integer column.**

■ **If dictionary codes are order-preserving and you insert a new distinct value that falls between two existing values, you may need to reassign codes.**

□ None of the above.

Solution:

- Delta Encoding is good at compressing predicatable time series data: **T**. A predictable time series is likely to have larger runs of deltas. For instance, a column that stores timestamps at 5 minute intervals can be collapsed into a large run of +5 minutes.
- If dictionary codes are order-preserving and you insert a new distinct value that falls between two existing values, you may need to reassign codes: **T**. Since we would need the new value between the two existing values, if the two existing values are contiguous, they will need to be reassigned.
- Run Length Encoding is most effective on unsorted uniformly random data: **F**. RLE relies on repeated values in the column, and randomness reduces its efficacy.
- Run-length Encoding is effective for compressing a low cardinality integer column: **T**. With low cardinality columns, we will often have runs of data with the same value, which may benefit from RLE
- For a *point lookup-only* workload, order-preserving dictionary encoding is required: **F**. Since we are only looking up an exact value, we only require the dictionary's hash properties.