

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

Homework #2 (by Saransh) – Solutions  
Due: **Sunday Feb 8, 2026 @ 11:59pm**

**IMPORTANT:**

- Enter all of your answers into **Gradescope by 11:59pm on Sunday Feb 8, 2026.**
- **Plagiarism:** Homework may be discussed with other students, but all homework is to be completed **individually**.

For your information:

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

*Revision : 2026/02/18 16:48*

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 can cause write amplification**

Log structured storage requires the DBMS to check previous records on update operations

Log structured storage can cause fragmentation

■ **In leveled compaction, SSTables within a level (except Level 0) are 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.

(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 cause write amplification

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

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

Slotted-page storage does not allow variable length tuples

None of the above

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

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

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.

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

(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
- For an append-only workload, both achieve comparable performance**
- Slotted-page requires storing schema metadata with the tuple, while log-storage does not
- Both designs may require background maintenance to reclaim space after numerous inserts/updates/deletes**
- Log-structured storage requires less disk space than slotted-page storage
- 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.

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

Neither format requires storing schema metadata with the tuple

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.

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

**Question 2: Storage Models.....[35 points]****Graded by:**

Consider a database with a single table  $G(\text{game\_id}, \text{dev\_id}, \text{total\_sales}, \text{player\_count})$ , where  $\text{game\_id}$  is the *primary key*, and all attributes are the same fixed width. Suppose  $G$  has 10,000 tuples that fit into 400 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.
- $G$  does *not* have any indexes (including for primary key  $\text{game\_id}$ ).
- None of  $G$ 's pages are already in memory. The DBMS can store an infinite number of pages in memory.
- Content-wise, the tuples of  $G$  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  $G$  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(player_count) FROM G
WHERE dev_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  $G$ .*
- 1-100    101-200    201-300    301-400    > 400    Not possible to determine

**Solution:** 400 pages. To find  $\text{player\_count}$  and  $\text{dev\_id}$  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  $G$ .*
- 1-100    101-200    201-300    301-400    > 400    Not possible to determine

**Solution:** 200 pages. There are 100 pages per attribute. 100 pages to find  $\text{dev\_id}$  for all tuples. In the worst-case scenario for  $G$ 's content,  $\text{player\_count}$  for all tuples must be accessed as well. Hence, another 100 pages must be read.

(b) Now consider the following query:

```
SELECT player_count, dev_id FROM G
WHERE game_id = 1 OR game_id = 999
```

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

$\alpha$ ) [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     10-100     101-200     201-300     301-400  
 > 400     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.

$\beta$ ) [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     10-100     101-200     201-300     301-400  
 > 400     Not possible to determine

**Solution:** 400 pages. At least one tuple with matching primary key is located on the last page. We must thus scan through every page.

ii. Suppose the DBMS uses the decomposition storage model (DSM) with implicit off-sets.

$\alpha$ ) [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     10-100     101-200     201-300     301-400  
 > 400     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 `game_id=1` and `game_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 `player_count` and `dev_id` at their corresponding offsets.

$\beta$ ) [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     10-100     101-200     201-300     301-400  
 > 400     Not possible to determine

**Solution:** 104 pages. There are 100 pages per attribute. In the worst case, we scan through all 100 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 `game_id=1` and `game_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 `dev_id` and `player_count`.

(c) Finally consider the following query:

```
SELECT MIN(game_id) FROM G
WHERE player_count = (SELECT MIN(player_count) FROM G);
```

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?

- 1    2-9    10-100    **101-200**    201-300    301-400    > 400  
 Not possible to determine

**Solution:** 200 pages. 100 pages for the inner select, and 100 pages to get the `game_id` since the buffer pool will have the `player_count` 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:

[Gates-Hillman Complex, Porter Hall, Doherty Hall, Wean Hall, Hunt Library]

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.

- [1, 2, 3, 4, 5]  
 [2, 4, 1, 5, 3]  
 [0, 12, 32, 33, 31]  
 [50, 40, 20, 10, 30]  
 [10, 34, 8, 67, 17]  
 [49, 9, 29, 19, 39]

**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 compress 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.*

- Bit-packing Encoding  
 **Run-length 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.
- **Run-length Encoding is effective for compressing a low cardinality integer column.**
  - Delta Encoding is good at compressing large text values.
  - For *point lookup-only* workload, order-preserving dictionary encoding is required.
  - **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.**
  - Run Length Encoding is most effective on unsorted uniformly random data.
  - None of the above.

**Solution:**

- 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
- Delta Encoding is good at compressing large text values: **F**. Large text values in theory have large differences; hence delta encoding may not be an effective choice.
- For *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.
- 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.