CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2025)
PROF. ANDY PAVLO

Homework #2 (by Will) – Solutions

Due: **Sunday Sept 21, 2025 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday Sept 21, 2025.**
- **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: ≈4-6 hours (1-1.5 hours for each question)
- Each part is all or nothing. There is no partial credit.

Revision : 2025/09/26 19:30

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 problems are associated with the *log-structured storage* in a database system? Select all that apply.

- ■ **Write Amplification**
- □ Increased Random Writes
- □ Increased Random Reads
- □ Fragmentation
- □ None of the above

Solution: Log-structure storage is particularly beneficial for write-intensive workloads, such as append-only data. But it incurs write amplification due to compaction.

Although a log-structured DBMS may have to read multiple pages to find a tuple, these will be sequential I/Os and not random reads.

(b) **[10 points]** Which problems are associated with the *slotted-page storage* in a database system? Select all that apply.

- □ Write Amplification
- ■ **Increased Random Reads**
- ■ **Increased Random Writes**
- ■ **Fragmentation**
- □ None of the above

Solution: The Slotted-Page Design often leads to fragmentation, as deletion of tuples can leave gaps in the pages, making them not fully utilized.

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.

(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 support variable length tuples.**
- □ Log-structured storage requires less disk space.
- ■ **For an append-only workload, both achieve comparable performance.**
- □ Slotted-page requires storing schema metadata with the tuple.
- □ After lots of insert/update/deletes, only log-structured benefits from maintenance.
- □ None of the above are true.

Solution: In absence of indexes + metadata, then slotted-page for append-only workload becomes the log-structured storage architecture.

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

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

Graded by:

Consider a database with a single table P(player_id,team_id,G_all,HR), where player_id is the *primary key*, and all attributes are the same fixed width. Suppose P has 20,000 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.
- P does *not* have any indexes (including for primary key player_id).
- None of P's pages are already in memory. The DBMS can store an infinite number of pages in memory.
- Content-wise, the tuples of P will always make each query run the longest possible and do the most page accesses.
- The tuples of P 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 and think of all possible orderings)

(a) Consider the following query:

```
SELECT MAX(HR) FROM P
    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 P.
☐ 1-50 ☐ 51-100 ☐ 101-150 ■ **151-200** ☐ ≥ 201 ☐ Not possible to determine

Solution: 200 pages. To find HR 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 P.
☐ 1-50 ■ **51-100** ☐ 101-150 ☐ 151-200 ☐ ≥ 201 ☐ 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 P's content, HR for all tuples must be accessed as well. Hence, another 50 pages must be read.

(b) Now consider the following query:

```
SELECT HR, team_id FROM P
   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** ☐ 2-3 ☐ 4-6 ☐ 7-9 ☐ 10-100 ☐ 101-200 ☐ $\geq$ 201 ☐ 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 ☐ 2-3 ☐ 4-6 ☐ 7-9 ☐ 10-100 ■ **101-200** ☐ $\geq$ 201 ☐ Not possible to determine

Solution: 200 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 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?

■ **1-3** ☐ 4-6 ☐ 7-9 ☐ 10-100 ☐ 101-200 ☐ $\geq$ 201 ☐ 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 `HR` 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?

☐ 1-20 ☐ 21-40 ■ **41-60** ☐ 61-80 ☐ 81-100 ☐ $\geq$ 101 ☐ 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 `HR`.

(c) Finally consider the following query:

```
SELECT MIN(player_id) FROM P
    WHERE HR = (SELECT MIN(HR) FROM P);
```

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?
☐ 1-50 ■ **51-100** ☐ 101-150 ☐ 151-200 ☐ $\geq$ 201 ☐ 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 `HR` 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:

```
[Milwaukee Braves, Baltimore Orioles, Buffalo Blues,
Brooklyn Gladiators, Chicago Cubs]
```

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]`
- ■ `[4,0,2,1,3]`
- ☐ `[79,12,32,33,31]`
- ☐ `[50,40,20,10,30]`
- ☐ `[79,12,33,15,32]`
- ■ `[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 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 improves 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.

☐ Delta Encoding is good at compressing large text values.

☐ Inserts and updates on high cardinality columns with bitmap encoding has no overhead.

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

☐ For a heavy update workload, dictionary encoding and delta encoding always require re-encoding.

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

☐ None of the above.

Solution:

- 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.
- Inserts and updates on high cardinality columns with bitmap encoding has no overhead: **F**. On high cardinality columns with bitmap encoding, each insert/update would need to edit all the bitmaps which would generate extra writes.
- 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.
- For a heavy update workload, dictionary encoding and delta encoding always require re-encoding: **F**. If update values are already in the dictionary, no re-encoding is needed.
- Run-length Encoding is effective for compressing low cardinality integer column: **T**.