DTS207TC Database Development and Design

Lab 9 Tutorial: Data Storage

Instructor: Xiaowu Sun
School of AI and Advanced Computing (AIAC), XJTLU

November 2025

Contents

1 Lab 9.Part 1: Fixed-Length Record

Theoretical Background

A **fixed-length record** file stores all tuples with the same byte size. Each field is allocated its maximum length, so every record has an identical structure and predictable offset. This design simplifies addressing and I/O operations because the i-th record starts exactly at:

$$\text{offset}(i) = i \times \text{RECORD\_SIZE}.$$

Example Schema

Consider an instructor record with the following schema:

`(ID: varchar(5), name: varchar(20), dept_name: varchar(20), salary: numeric(8,2))`

Instructor Record Structure

Assuming 1 byte per character and 8 bytes for the numeric field, the maximum record length is $5 + 20 + 20 + 8 = 53$ bytes.
In this lab, the schema is simplified to fixed-size ASCII fields and one integer salary field:

- **ID** – 5 bytes (ASCII)
- **Name** – 20 bytes (ASCII)
- **Department** – 20 bytes (ASCII)
- **Salary** – 4 bytes (little-endian integer, in cents)

Each record therefore occupies **49 bytes** in total.

Record Size and Page Capacity

Page Capacity Calculation

A single database page is 4KB (4096 bytes). The number of records that fit entirely within one page is:

$$\text{CAPACITY} = \left\lfloor \frac{\text{PAGE\_SIZE}}{\text{RECORD\_SIZE}} \right\rfloor = \left\lfloor \frac{4096}{49} \right\rfloor = 83.$$

Hence, each 4KB page can hold up to **83 complete records** without crossing page boundaries.

Code Implementation

```python
PAGE_SIZE   = 4096  # 4KB page

# Structured dtype with fixed-length fields:
#   id:   5 bytes ASCII
#   name: 20 bytes ASCII
#   dept: 20 bytes ASCII
#   salary: 4-byte little-endian int (we store cents to avoid float issues)
dtype       = np.dtype([
    ('id','S5'), ('name','S20'), ('dept','S20'), ('salary','<i4')
])
RECORD_SIZE = dtype.itemsize   # 5 + 20 + 20 + 4 = 49 bytes
CAPACITY    = PAGE_SIZE // RECORD_SIZE  # number of records the page can hold
```

Listing 1: Definition of fixed-length record structure

The free-list method keeps a header pointer to the first free slot; each deleted slot stores the index of the next free one. On insertion, the head slot is reused first. This structure exactly matches the “free-list file” described in database textbooks (Figure 13.4).

Field Validation and Encoding Utilities

```python
def _check_len(self, s: str, n: int, field: str)
def _encode(self, id5: str, name20: str, dept20: str, salary_float: float)
def _decode_row(self, rec)
```

Listing 2: Utility function definitions

Explanation of Utility Functions

__check__len(s, n, field) Ensures that a string value does not exceed the schema-defined maximum byte length. For example, the `id` field must be at most 5 bytes, while `name` and `dept` are limited to 20 bytes each. This keeps all records aligned to a fixed length and prevents overflow.

__encode(id5, name20, dept20, salary__float) Converts input values from Python objects into the binary form stored in the page. Strings are encoded as ASCII byte arrays, and the salary is scaled to integer cents and stored as a 4-byte integer. The function returns a tuple of encoded values ready for insertion into the NumPy structured array.

__decode__row(rec) Transforms a stored binary record back into a Python-friendly dictionary. It removes padding zeros from ASCII strings and divides the stored integer salary by 100 to recover the original floating-point value. This allows readable inspection of data when printing or debugging the page content.

Exercise 1.1 — Implement `insert()` (Dense Prefix)

Run: `python lab9_1_fixed_record_lab.py`

Exercise 1.1 — Implement `insert()` (dense-prefix append)

Goal: Append a new record at the end of the dense prefix $[0..\texttt{count} - 1]$ with no gaps.

```python
def insert(self, id5: str, name20: str, dept20: str, salary_float: float):
    """
␣␣␣␣Append␣a␣new␣record␣at␣the␣end␣of␣the␣dense␣prefix␣[0␣..␣count-1].
␣␣␣␣Assumes␣there␣are␣no␣holes␣in␣the␣current␣prefix.
␣␣␣␣"""

    # TODO:
    # 1) If full (self.count >= CAPACITY), return None.
    # 2) i = self.count
    # 3) rid, rname, rdept, rcents = self._encode(id5, name20, dept20,
       salary_float)
    # 4) Write fields into self.data at index i
    # 5) self.count += 1
    # 6) self.next_idx = max(self.next_idx, self.count)
    # 7) return i
```

Listing 3: Function to implement: insert into dense prefix

Hints / Checkpoint

- Use `_encode(...)` to get fixed-size binary fields.
- After success, `count` increases by 1; `next_idx` stays $\geq$ `count`.
- Return the index `i` where the new record was written.

Exercise 1.2 — Implement `delete_swap()`

Exercise 1.2 — Implement `delete_swap()` (O(1), changes order)

Goal: Delete index `i` by moving the last record into position `i`.

```
def delete_swap(self, i: int) -> bool:
    """
␣␣␣␣O(1)␣deletion␣by␣swapping␣the␣last␣record␣into␣position␣i.
␣␣␣␣This␣operation␣changes␣record␣order.
␣␣␣␣"""

    # TODO:
    # 1) Validate 0 <= i < self.count; else return False
    # 2) last = self.count - 1
    # 3) If i != last: self.data[i] = self.data[last]
    # 4) self.count -= 1
    # 5) self.next_idx = max(self.next_idx, self.count)
    # 6) return True
```

Listing 4: Function to implement: O(1) deletion using swap

Concept: Swap Deletion

- Complexity: **O(1)** — only overwrites one record.
- Changes logical order — record at the end moves into the deleted slot.
- Fast and suitable when record ordering is not important.

Exercise 1.3 — Implement `delete_compact()`

Exercise 1.3 — Implement `delete_compact()` (stable, O(n))

Goal: Delete index `i` and preserve order by shifting subsequent records left.

```
def delete_compact(self, i: int) -> bool:
    """
␣␣␣␣Stable␣deletion␣by␣shifting␣subsequent␣records␣left␣by␣one␣position.
␣␣␣␣Preserves␣order␣but␣requires␣more␣data␣movement.
␣␣␣␣"""

    # TODO:
    # 1) Validate 0 <= i < self.count; else return False
    # 2) If i < self.count - 1:
    #        self.data[i:self.count - 1] = self.data[i + 1:self.count]
    # 3) self.count -= 1
    # 4) self.next_idx = max(self.next_idx, self.count)
    # 5) return True
```

Listing 5: Function to implement: stable deletion by compaction

Concept: Compact Deletion

- Complexity: **O(n)** — must shift all following records.
- Order-preserving, suitable for sequential scans.
- More expensive for large pages or frequent deletions.

Exercise 1.4 — Run and Observe (Practical Demo)

Exercise 1.4.1 — Insert Records (Dense Prefix)

Goal: Insert all instructor records and inspect the page state.

```
page.insert("10101","Srinivasan","Comp. Sci.",65000.00)
page.insert("12121","Wu","Finance",90000.00)
page.insert("15151","Mozart","Music",40000.00)
page.insert("22222","Einstein","Physics",95000.00)
page.insert("32343","El Said","History",60000.00)
page.insert("33456","Gold","Physics",87000.00)
page.insert("45565","Katz","Comp. Sci.",75000.00)
page.insert("58583","Califeri","History",62000.00)

print("stats:", page.stats())
page.dump_visible()
```

Checkpoint

Expect:

- `count = 8`, `next_idx = 8`, `free_head = -1`
- `free_chain = []` (no deleted slots)
- Visible indices 0–7 in insertion order

Exercise 1.4.2 — O(1) Deletion by Swap

Goal: Delete index 0 and observe reordering.

```
# page.delete_swap(0)
# print("after delete_swap(0), read(0):", page.read(0))
# print("stats:", page.stats())
# page.dump_visible()
```

Observe

The record at the last index overwrites slot 0. `count` decreases by 1; order changes. `next_idx` remains $\geq$ `count`.

Exercise 1.4.3 — Stable Deletion (Compaction)

Goal: Delete index 0 again but preserve order.

```
# page.delete_compact(0)
# page.dump_visible()
# print("stats:", page.stats())
```

Observe

Records from indices [1..7] shift left; `count` decreases by 1, order preserved. Higher cost than swap deletion.

Exercise 1.4.4 — Free-list Deletion (Pointer-based)

Goal: Use a linked free-list for deletions without moving data.

```
# page.delete_to_freelist(1)
# page.delete_to_freelist(3)
# print("after delete_to_freelist(1,3):", page.stats())
# page.dump_visible()
```

Observe

Expect `free_head = 3`, `free_chain = [3, 1]`. No data movement — faster, but uses pointer links.

Exercise 1.4.5 — Free-list Insertion (Slot Reuse)

Goal: Reuse deleted slots via `insert_using_freelist()`.

```
# posA = page.insert_using_freelist("A0001","Alice","X",11111.00)
# posB = page.insert_using_freelist("A0002","Bob","Y",22222.00)
# posC = page.insert_using_freelist("A0003","Carol","Z",33333.00)
# print("reused/append positions:", posA, posB, posC)

# page.dump_visible()
# print("final stats:", page.stats())
```

Observe

Slots are reused in **LIFO** order: first 3, then 1, then append to end. After reuse, `free_head = -1`, and `next_idx` advances only on append.

2 Lab 9-Part 2: Variable-Length Record

Theoretical Background

Unlike fixed-length records, variable-length records allow some fields (such as `name` or `dept`) to take different sizes in each record. A special header area at the beginning of each record stores metadata — including offsets, lengths, and a null bitmap — describing how to locate each field's data within the payload area.

Record Structure Example

Schema:

`(ID: varchar(5), Name: varchar(20), Dept: varchar(20), Salary: numeric(8,2))`

Offsets in the header specify where each field begins and ends inside the record's payload.

```python
# Header format:
#  - B:   null bitmap (1 byte; bit0=id, bit1=name, bit2=dept, bit3=salary)
#  - 6H:  (off,len) pairs for id, name, dept (2 bytes each, unsigned short)
#  - Q:   salary (8 bytes unsigned integer; or use i32 if storing cents)

HDR_FMT = "<BHHHHHHQ"   # little-endian layout
HDR_SIZE = struct.calcsize(HDR_FMT)
print("HDR_SIZE", HDR_SIZE)
```

Listing 6: Header format and size calculation

Explanation of Header Fields

- **Null Bitmap (1 byte)** – each bit marks whether a field is NULL.
- **Offsets and Lengths (6 × 2 bytes)** – record the start position and length for each text field (ID, Name, Dept).
- **Salary (8 bytes)** – numeric value stored as an unsigned 64-bit integer (or 4-byte int for cents).
- **Total header size:** 1 + 12 + 8 = 21 bytes.

The header serves as a miniature directory for the record, allowing the DBMS to locate each field efficiently, even when variable-length or NULL.

Exercise 2.1 — Encoding a Variable-Length Record

```python
def encode_record(id_s, name_s, dept_s, salary, nulls=()):
    """
␣␣␣␣Encode␣a␣variable-length␣instructor␣record␣into␣binary␣form.
␣␣␣␣"""
    # Step 1. Build a 1-byte "null bitmap"
    nb = 0
    if 'id' in nulls:     nb |= (1 << 0)
    if 'name' in nulls:   nb |= (1 << 1)
    if 'dept' in nulls:   nb |= (1 << 2)
    if 'salary' in nulls: nb |= (1 << 3)

    # Step 2. Convert non-null fields into ASCII bytes
    b_id   = b"" if (nb & (1 << 0)) else _b(id_s)
    b_name = b"" if (nb & (1 << 1)) else _b(name_s)
    b_dept = b"" if (nb & (1 << 2)) else _b(dept_s)

    # Step 3. Concatenate all variable-length text fields
    payload = b_id + b_name + b_dept

    # Step 4. Compute field offsets relative to start-of-record
    # TODO:
    # off_id   = ??
    # off_name = ??
    # off_dept = ??

    ## TODO: Print offsets, lengths, and values for debugging
    print("----- Record Layout (Header Summary) -----")
    print("ID     -> offset=?, length=?, value=?")
    print("Name   -> offset=?, length=?, value=?")
    print("Dept   -> offset=?, length=?, value=?")

    # Step 5. Numeric salary (0 if NULL)
    sal = 0 if (nb & (1 << 3)) else int(salary)

    # Step 6. Pack the header
    header = struct.pack(
        HDR_FMT,
        nb,
        off_id,   len(b_id),
        off_name, len(b_name),
        off_dept, len(b_dept),
        sal
    )

    # Step 7. Return concatenation of header and payload
    return header + payload
```

Listing 7: Function to implement: encode a variable-length record

Hints for Students

- Step 4 computes offsets relative to the record start. The first payload byte begins at `HDR_SIZE`.
- The `print(...)` lines are used to check correctness: verify offsets and lengths match each field's byte size.
- Step 6 packs all metadata into the fixed-size header.
- The final record is `header + payload`.

Exercise 2.2 — Handling NULL Fields

Test NULL Field Encoding

Goal: Encode records with some fields set to NULL using the `nulls` argument.

```
# Example 1: No NULLs
r1 = encode_record("10101", "Srinivasan", "Comp.␣Sci.", 6500000)

# Example 2: Only salary is NULL
r2 = encode_record("10101", "Srinivasan", "Comp.␣Sci.", 0, nulls={"salary"
    })

# Example 3: Multiple NULLs (name and dept)
r3 = encode_record("10101", "", "", 7000000, nulls={"name", "dept"})
```

Checkpoint

- Observe how the null bitmap (`nb`) changes:
  - All non-NULL then `0000` (0)
  - Salary NULL then `1000` (8)
  - Name + Dept NULL then `0110` (6)
- Note that payload size shrinks when more fields are NULL.

Try Modifying a Field to NULL

Goal: Modify an existing record to make one or more fields NULL and observe the change in offsets.

```
# Original record (no NULL)
r_orig = encode_record("12121", "Wu", "Finance", 9000000)

# Modified record (dept becomes NULL)
r_null = encode_record("12121", "Wu", "", 9000000, nulls={"dept"})
```

Observation

Compare the printed offsets of `r_orig` and `r_null`. The missing field shortens the payload and shifts subsequent offsets.

Concept Summary

- Variable-length records combine a fixed-size header and a flexible payload.
- Offsets and lengths allow direct access to each field's data.
- A 1-byte null bitmap efficiently marks missing attributes.
- This structure is the foundation for the slotted-page layout used in modern DBMSs.

3 Lab 9-Part 3: Slotted Page for Variable-Length Records

Theoretical Background

A **slotted page** manages variable-length records within a fixed-size page. Two regions grow toward each other:

- The slot directory (array of record pointers) grows upward.
- The data region (actual records) grows downward.

Free space lies between them. Each slot stores (offset, length) for its record.

Core Code

```python
def insert_record(self, rec_bytes):
    size = len(rec_bytes)
    free_start = (len(self.slots)+1) * 8
    if size + 8 > self.free_end - free_start: return None
    self.free_end -= size
    offset = self.free_end
    self.data[offset:offset+size] = rec_bytes
    self.slots.append((offset, size))
```

Listing 8: Insert into slotted page

Explanation

Slots provide logical record identifiers. Deleting a record invalidates its slot rather than moving data. Free space fragmentation can be reclaimed by compaction when necessary.

Demo

Run: `python lab9_3_slotted_page_lab.py`

Expected: records inserted with varying lengths, slot table updated dynamically.

4 Lab 9-Part 4: Free-Space Map (FSM)

Theoretical Background

A Free-Space Map (FSM) tracks the amount of unused space per data page. A two-level FSM reduces search cost:

- Level-0: exact free-space buckets per page,
- Level-1: max free fraction per group of pages.

Core Code

```python
def find_page(self, min_free_frac=0.0):
    need = self.bucketize(min_free_frac)
    for i, mx in enumerate(self.l1):
        if mx >= need:
            gidx = i; break
    s, e = gidx*self.G, min(len(self.l0), gidx*self.G+self.G)
    for pid in range(s, e):
        if self.l0[pid] >= need: return pid
```

Listing 9: Two-level FSM search algorithm

Explanation

By summarizing groups, FSM reduces the expected scan time from $O(N)$ to $O(\sqrt{N})$. This hierarchical structure is conceptually similar to a multi-level index.

Demo

Run: `python lab9_4_fsm_lab.py`

Observe how FSM efficiently locates pages with sufficient free space.

5 Lab 9-Part 5: Buffer Replacement Policies

Theoretical Background

A **database buffer pool** is a fixed-size memory area used to cache recently accessed disk pages. When the system needs to access a page, it first checks whether the page is already in the buffer:

- **HIT**: the page is found in memory, so no disk I/O is needed.
- **FAULT**: the page is not in memory and must be loaded from disk.

Because memory capacity is limited, some page must be **evicted** when the buffer is full and a new page arrives. A **replacement policy** determines which page to remove.

Page Replacement Concept

Each buffer access produces either a HIT or a FAULT. If a FAULT occurs and the buffer is full, one page is selected for eviction:

```
disk then buffer (if full then evict one page then insert new page)
```

Common replacement policies:

- **FIFO (First-In-First-Out)** — evict the page that has been in memory the longest.
- **LRU (Least Recently Used)** — evict the page that has not been accessed for the longest time.
- **Random** — evict a random page (simple baseline for comparison).

Exercise 5.1 — Implement `FIFOBuffer.access()`

```python
from collections import deque

class FIFOBuffer:
    """
␣␣␣␣FIFO␣(First-In-First-Out)␣page␣replacement.
␣␣␣␣-␣Keep␣pages␣in␣a␣queue␣(left␣=␣oldest,␣right␣=␣newest).
␣␣␣␣-␣On␣FAULT␣when␣full:␣evict␣the␣oldest␣page␣(front␣of␣queue).
␣␣␣␣-␣On␣HIT:␣order␣does␣not␣change.
␣␣␣␣"""
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.q = deque()           # queue: leftmost is oldest
        self.in_mem = set()        # O(1) membership
        self.last_evicted = None   # record last evicted page for printing

    def access(self, page_id: int) -> bool:
        """
␣␣␣␣␣␣␣␣Return␣True␣if␣FAULT,␣False␣if␣HIT.
␣␣␣␣␣␣␣␣Update␣self.last_evicted␣when␣eviction␣happens.
␣␣␣␣␣␣␣␣"""
        # TODO:
        # 1) If page_id in self.in_mem:   # HIT

        #
        # 2) Else (FAULT) and self.capacity > 0:

        pass
```

Listing 10: TODO: FIFO (queue-based) replacement

Hints

- Evict from the left of the queue (`popleft()`).
- Insert new pages at the right (`append()`).
- Always keep `in_mem` in sync with the queue.

Exercise 5.2 — Implement `LRUBuffer.access()`

```python
from collections import OrderedDict

class LRUBuffer:
    """
␣␣␣␣LRU␣(Least␣Recently␣Used)␣page␣replacement.
␣␣␣␣-␣Use␣an␣OrderedDict␣to␣maintain␣recency.
␣␣␣␣-␣On␣HIT:␣move␣page␣to␣end␣(most␣recent).
␣␣␣␣-␣On␣FAULT␣when␣full:␣evict␣from␣front␣(least␣recent).
␣␣␣␣"""
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.od = OrderedDict()   # keys in recency order: left=LRU, right=MRU
        self.last_evicted = None

    def access(self, page_id: int) -> bool:
        """
␣␣␣␣␣␣␣␣Return␣True␣if␣FAULT,␣False␣if␣HIT.
␣␣␣␣␣␣␣␣Update␣self.last_evicted␣when␣eviction␣happens.
␣␣␣␣␣␣␣␣"""
        # TODO:
        # 1) If page_id in self.od:          # HIT

        #
        # 2) Else (FAULT) and self.capacity > 0:

        #
        # 3) return True
        pass
```

Listing 11: TODO: LRU (recency-based) replacement

Hints

- `move_to_end(page_id)` marks a page as most recently used.
- `popitem(last=False)` evicts the least recently used page.

Exercise 5.3 — Implement `RandomBuffer.access()`

```python
import random

class RandomBuffer:
    """
␣␣␣␣Random␣page␣replacement␣(baseline).
␣␣␣␣-␣On␣FAULT␣when␣full:␣evict␣one␣random␣page.
␣␣␣␣-␣Does␣not␣use␣recency␣or␣frequency.
␣␣␣␣"""
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.pages = set()          # pages in memory (unordered)
        self.last_evicted = None

    def access(self, page_id: int) -> bool:
        """
```

```
␣␣␣␣␣␣␣␣Return␣True␣if␣FAULT,␣False␣if␣HIT.
␣␣␣␣␣␣␣␣Update␣self.last_evicted␣when␣eviction␣happens.
␣␣␣␣␣␣␣␣"""
        # TODO:
        # 1) If page_id in self.pages:   # HIT

        # 2) Else (FAULT) and self.capacity > 0:

        pass
```

Listing 12: TODO: Random (baseline) replacement

Hints

- Convert the set to a list for `random.choice`.
- Keep `last_evicted` as `None` when no eviction occurs.

Exercise 5.4 — Step-by-step Simulation (Same Trace, Same Capacity)

Goal

Run all three policies on the *same* short access sequence with the *same* capacity. Print each step: HIT/FAULT, evicted page, and the buffer state. This makes the behavioral differences obvious.

```
def buffer_order(buf):
    """Pretty-print␣buffer␣state␣by␣policy␣type."""
    if hasattr(buf, "q"):       # FIFO: oldest -> newest
        return list(buf.q)
    if hasattr(buf, "od"):      # LRU:  LRU -> MRU
        return list(buf.od.keys())
    if hasattr(buf, "pages"):  # Random: unordered set (sorted for stable view)
        return sorted(buf.pages)
    return []

def run_verbose(trace, capacity, policy_cls, policy_name):
    """
␣␣␣␣Print␣per-step:
␣␣␣␣␣␣-␣accessed␣page
␣␣␣␣␣␣-␣HIT/FAULT
␣␣␣␣␣␣-␣evicted␣page␣(if␣any)
␣␣␣␣␣␣-␣buffer␣state␣after␣the␣access
␣␣␣␣"""
    buf = policy_cls(capacity)
    print(f"\n===␣{policy_name}␣(capacity={capacity})␣===")
    print(f"{'Step':>4}␣|␣{'Access':>6}␣|␣{'Result':>6}␣|␣{'Evicted':>8}␣|␣State")
    print("-" * 65)
    faults = 0
    for step, p in enumerate(trace, start=1):
        result = buf.access(p)
        evicted = getattr(buf, "last_evicted", None)
        if result:
            faults += 1
        print(f"{step:4d}␣|␣{p:6d}␣|␣{('FAULT'␣if␣result␣else␣'HIT'):>6}␣|␣"
              f"{str(evicted):>8}␣|␣{buffer_order(buf)}")
    print(f"Total␣faults:␣{faults}/{len(trace)}␣(Fault␣rate␣=␣{faults/len(trace):.3f})")

# Demo trace and capacity (short and readable)
trace = [1, 2, 3, 1, 4, 5, 2, 1, 2, 3, 4, 5]
```

```python
capacity = 3

# Run all three policies
run_verbose(trace, capacity, FIFOBuffer,  "FIFO")
run_verbose(trace, capacity, LRUBuffer,   "LRU")
run_verbose(trace, capacity, RandomBuffer,"Random")
```

Listing 13: Step-by-step runner with a short trace

What to Observe

- **FIFO**: queue order is oldest to newest. Notice how old but still-useful pages can be evicted.
- **LRU**: recently used pages are retained; least recent is evicted.
- **Random**: victims change unpredictably; fault rate fluctuates.