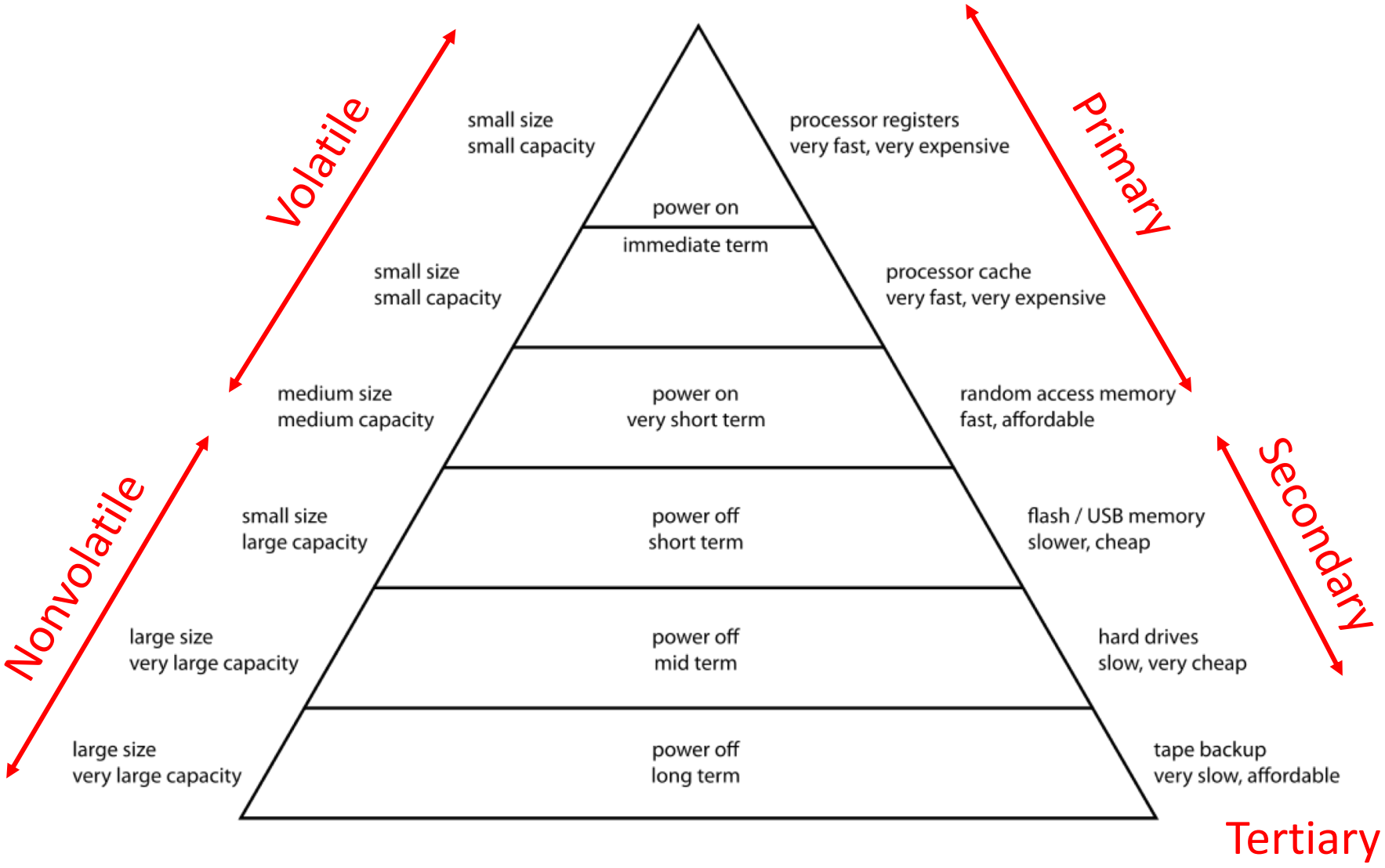
Storing Data

CSC443, Winter 2019

Sayyed Nezhadi

Computer Memory Hierarchy



Levels of Memory Hierarchy

- Cache: Volatile, fastest accessible memory (a few nanoseconds), expensive, typical size in megabytes
- Main Memory: Volatile, most used memory, fast access (around 100 nanoseconds), typical size in order of 10GB
- Secondary Storage: Hard drive or SSD, nonvolatile, very large storage capacity (terabytes), much slower than memory (a few milliseconds for hard drives).
- Tertiary Storage: tape drives or robotic disk arrays, nonvolatile, mostly used for data backup, huge storage capacity (petabytes), very slow access (seconds or minutes).

☛*In this course, we are mainly concerned with the main memory and the secondary storage*

Move Data to & from Secondary Storage

- The mechanical design of the hard drive naturally leads itself to superior sequential access.
- A hard drive consists of an (array of) rotating disk. Each disk has a head driven by a mechanical arm which performs the bitwise read and write. If a sequential data block is accessed, all bits can be read or written during a single rotation.
- Random access of the hard drive data may require multiple rotations for the head to reach all the regions on the disk.

Data Transfer Rate

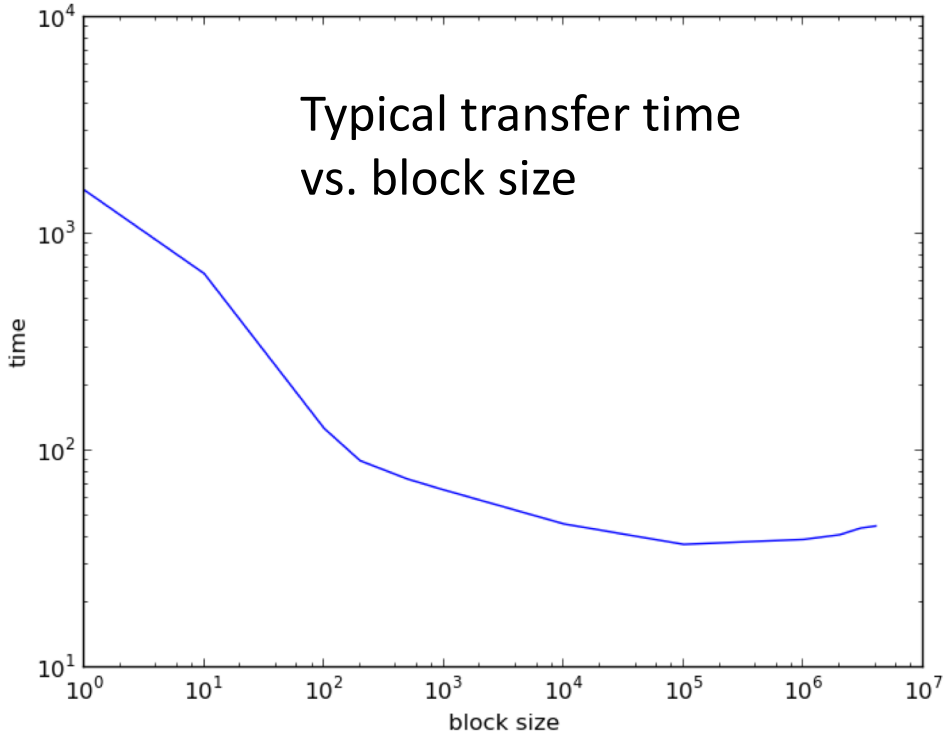
- Seek Time: measures *average* time it takes the head to travel to the region of interest on disk. Typically, the seek time is a few milliseconds.

Roational speed (rpm)	Seek time (ms)
15,000	2
10,000	3
7,200	4.16

- Data Transfer Rate: is the steady state speed the head can read or write data between the disk buffer and the disk. The data transfer rate can reach 1 terabytes / second.

Block Size

- The data transfer speed depends on:
  - contiguousness of the accessed physical locations on the disk
  - the size of the accessed data per request (block)



When data is requested in large enough blocks, we see a drastic improvement in data transfer rate.

Buffer Manager

- Another technique to further combat disk latency.
- All disk I/O issued by the database are done in units of pages.
- A section of the memory is used as the buffer pool.
- The buffer pool is partitioned into frames. Each frame is the size of a page.
- The buffer manager performs lazy disk I/O:
  - Disk read and write are first performed as frame read and write in the buffer pool
  - Data is transferred between the buffer pool and disk only when a requested page is not cached by the buffer pool, or new pages need to be cached but the buffer pool is full.

Relational Data on Disk

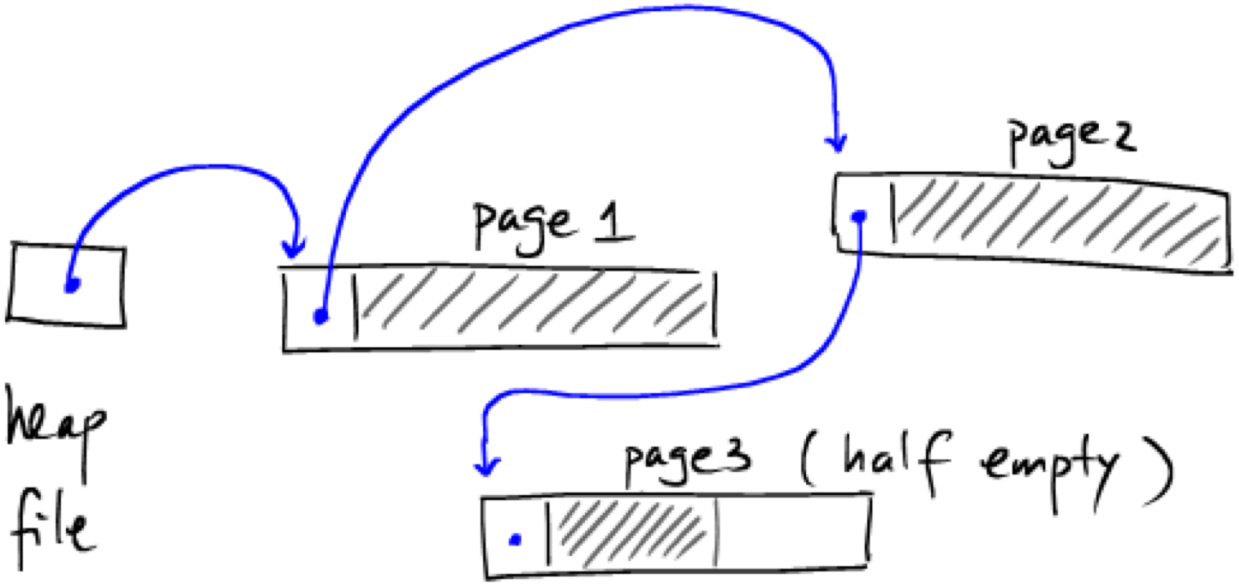
Heap File for Pages

- A heap file is a data structure that holds pages of data.
- Functionally, a heap file is similar to the main memory data structure linked list.
- It serves as a container of generic data items in an unordered fashion
- Similar to linked list, heap file:
  - offers efficient append of new data,
  - supports sequential scanning,
  - does not offer random access.

Properties of Heap File

- Data exists on secondary storage: it's designed to be efficient for large data volume, with capacity only limited by the available secondary storage.
- A heap file can span over multiple disks or machines: heap file can use large addressing schemes to span across multiple disks or ever networks.
- Data is organized into pages.
- Pages can be partially empty.

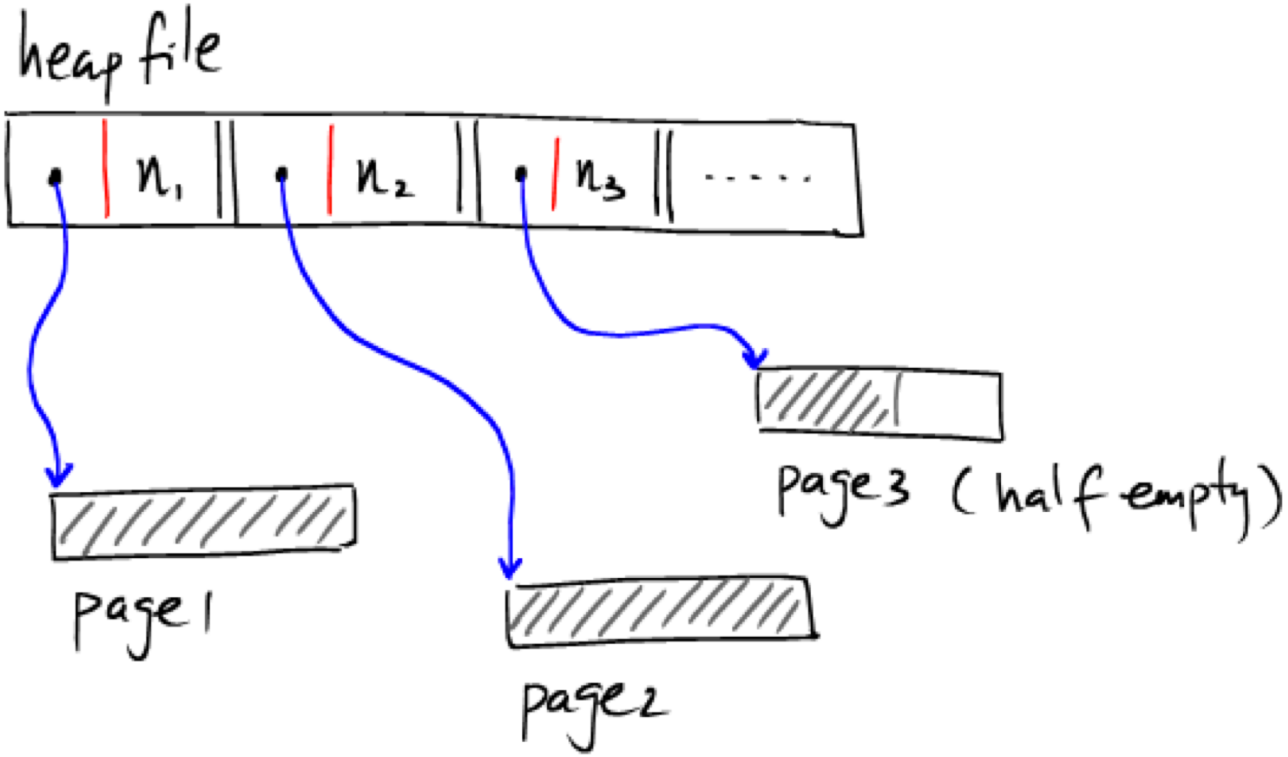
Heap File as a Linked List



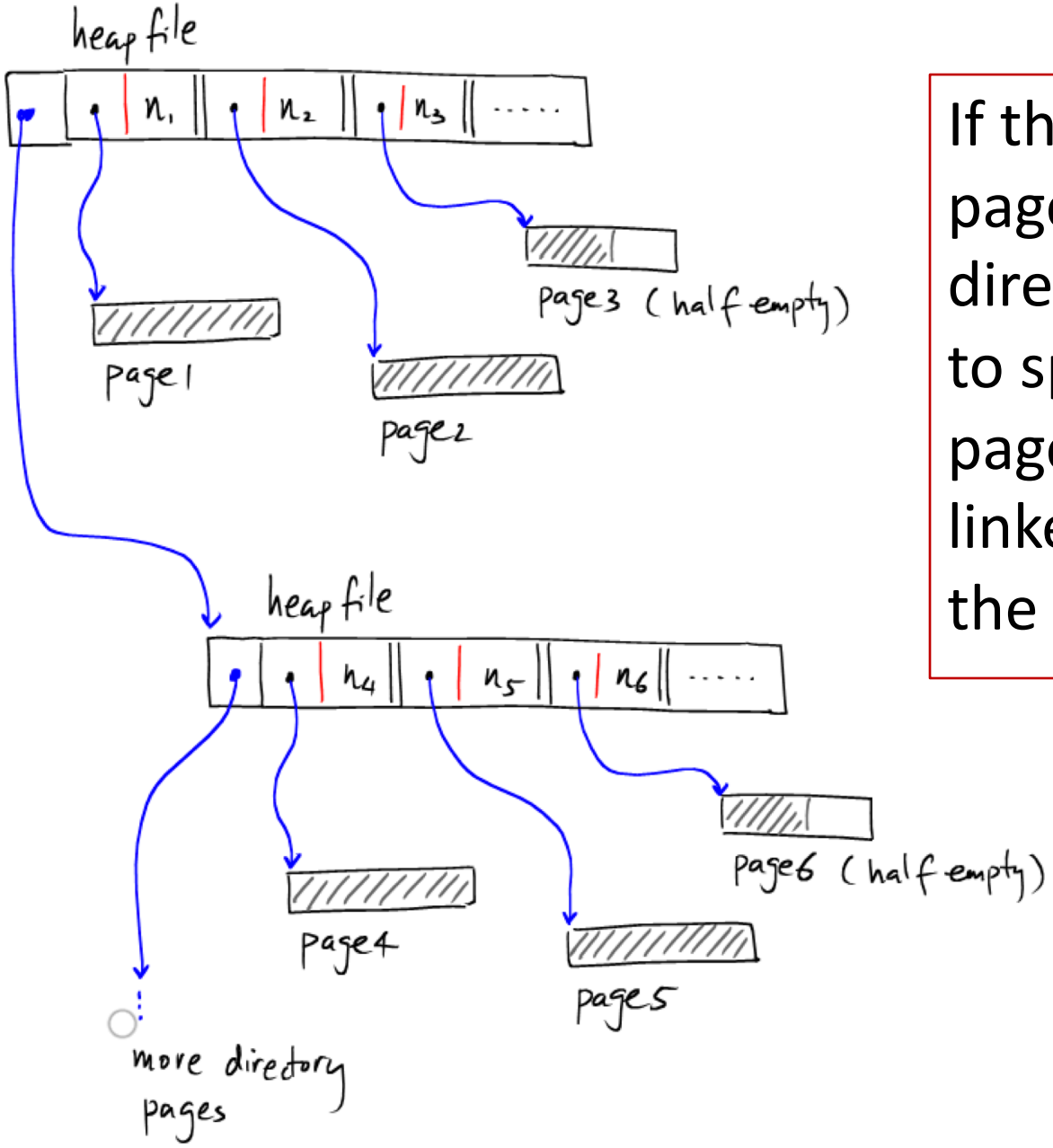
- Problem: finding free spaces is very expensive. To locate a page with free space (*page3*), we have to scan through the pages (*page1* and *page2*).

Directory-based Heap File

- Idea: use one or more blocks to store the directory of all the page addresses and their respective free space.



☛ *Since page1 and page2 are both full, their free space entries n1 and n2 will be both zero*



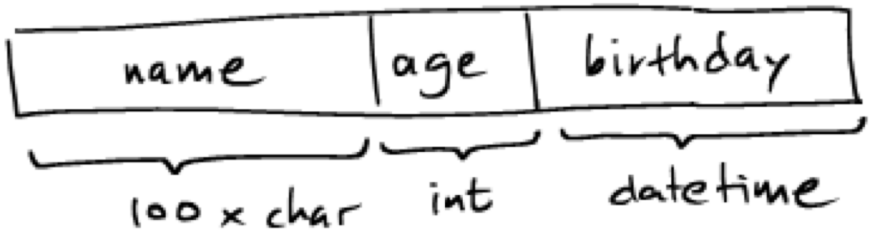
If there are too many pages that the directory itself needs to span over multiple pages, we may use a linked list to store the directory.

Record Serialization

- How to convert tuple records into byte arrays to be stored on disk (inside pages)?
- Serializing fixed length records: all records *must* be exactly the same length.
- Example:

```
CREATE TABLE Person1 (
    name      CHAR(100) NOT NULL,
    age       INTEGER NOT NULL,
    birthdate DATETIME NOT NULL
)
```

- Record Size:

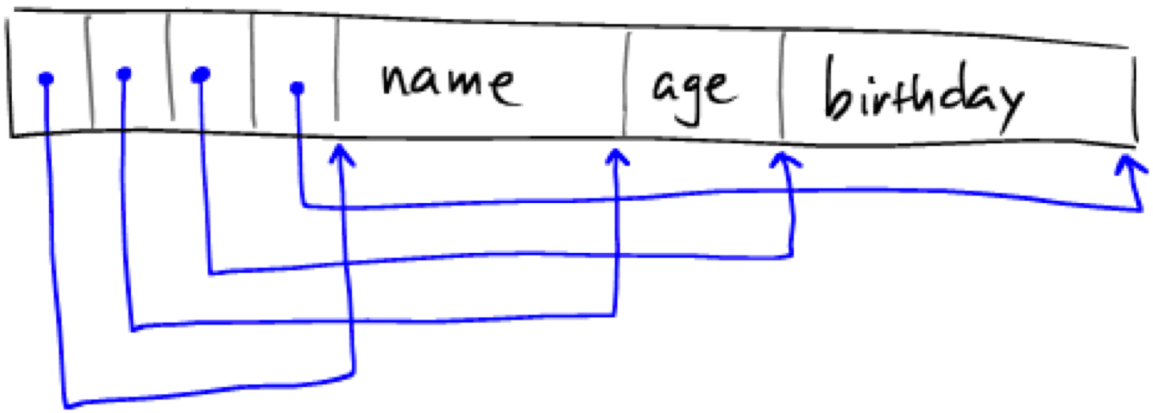



```
100 * sizeof(CHAR) + sizeof(INTEGER) + sizeof(DATETIME)
```

Record Serialization

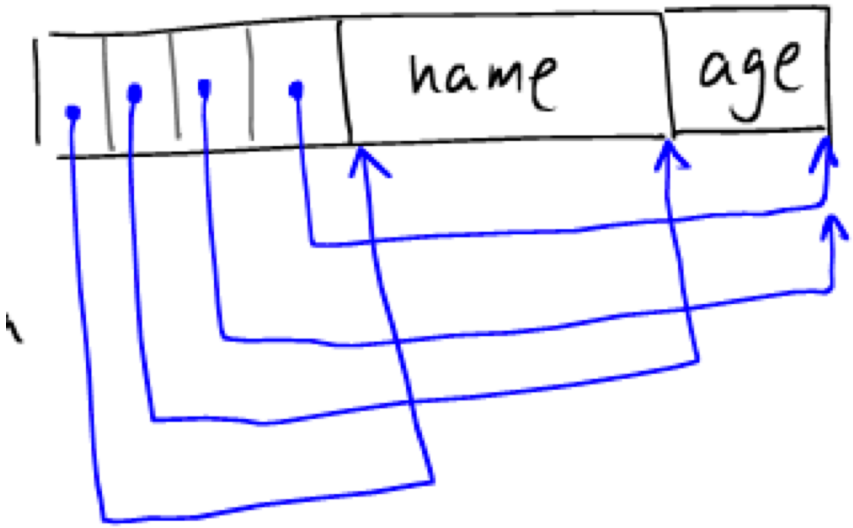
- Serializing variable length records: records have variable size. A popular method is to encode the offsets of the field boundaries in the header.
- Example:

```
CREATE TABLE Person2 (
    name      VARCHAR(1024) NOT NULL,
    age       INTEGER NOT NULL,
    birthdate DATETIME
)
```



- Having these boundary offsets provides a natural way of encoding NULL valued fields -- they can be viewed as fields with zero bytes:

- birthdate = NULL



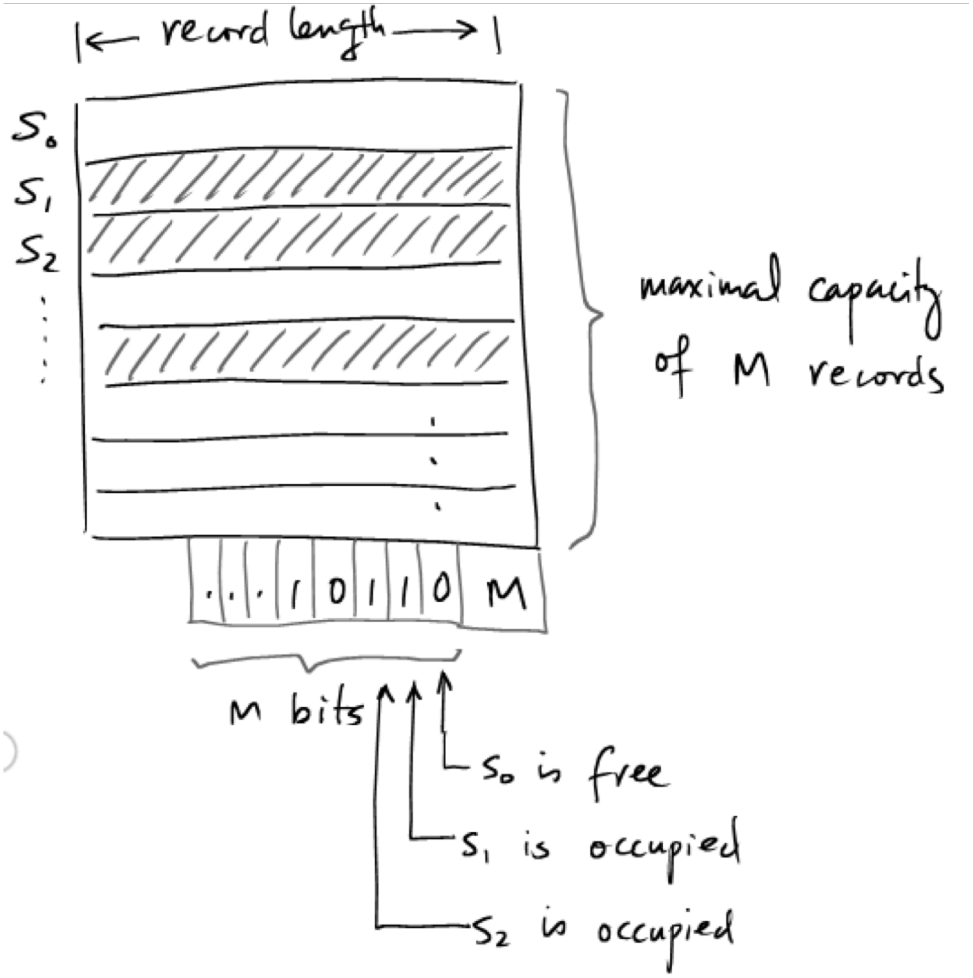
☛ *Here, we have assumed all value in a record are stored together. This is sometimes called row-oriented storage. Alternatively, all values of an attribute can be stored together in what is called column-oriented storage.*

Page Format

- How data is formatted in a each page?
- Simplification: Every page stores tuples with the same schema:
  - Every relation (table) occupies at least one page.
  - Page contains either fixed length tuples or variable length records.
- Important properties of page formats:
  - efficiently utilize disk space.
  - support scan, insert, delete and update with minimal disk I/O.
  - efficient way of assigning unique ID's to records (invariant during insert, delete and update)
  - each record fits into a page (will be relaxed later)

Page Format for Fixed Length Records

- **M** slots of equal size
- Each slot one record
- Last segment:
  - Value of **M**
  - **M** bits to indicate if the respective slot is occupied or free.



Page Format for Fixed Length Records

- Record ID: concat of <page ID, record offset>
- The record ID will never be affected by changes to the database.

INSERT	UPDATE	DELETE
1. Get **M** 2. Scan the M bits backward 3. All the bits are 1 -> no room in the page 4. K-th bit is 0 -> write at offset **nk**	Simply overwrite the old record (at the same location)	Set the corresponding bit in the bit index to 0

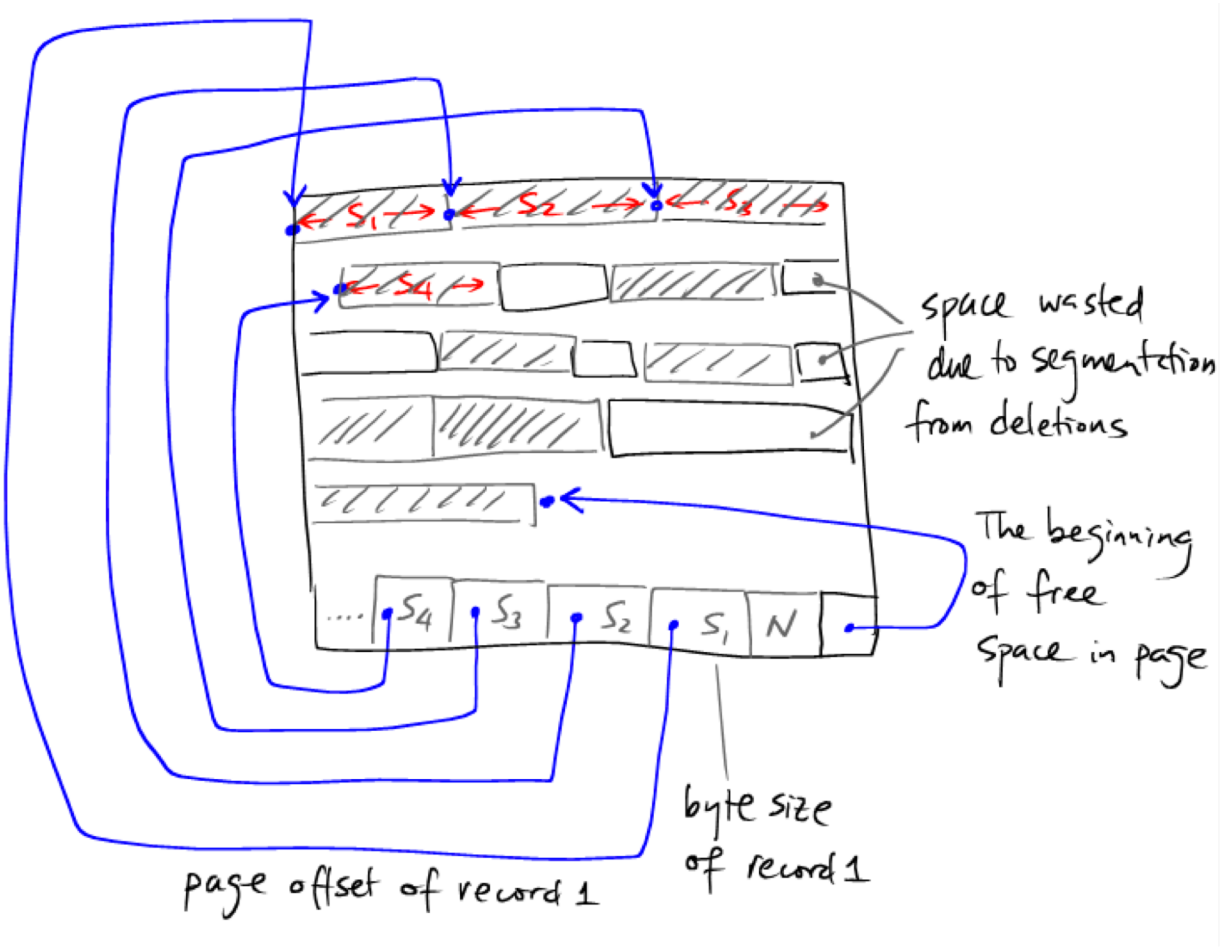
Page Format for Variable Length Records

Variable length records are much more difficult to maintain in a page for the following reasons:

- The record boundaries and field boundaries are dependent on the data content.
- When the data is modified, record size may change, resulting in physical relocation of the record (requires efficient methods to move records in a page or between pages)
- If record ID needs to be maintained, the record ID assignment must be invariant even if the record is to be moved.

Page Format for Variable Length Records

One efficient design is to maintain a page header which tracks the free space and a directory of records.



Page Format for Variable Length Records

- Record ID: concat of <page ID, directory index>
- The record ID is independent of changes to the database.

INSERT	DELETE
1. Check free space can hold the new record and the new entry in the directory 2. Write the record at the start of free space 3. Update directory (offset & size) 4. Update the free space pointer	1. If last entry in the directory, delete the entry and update free space pointer 2. Else, make the offset = -1 in the entry (mark sit as deleted)

☛ *Deletion leaves holes in the page. The space occupied by deleted records cannot be reclaimed unless we reorganize the page such that all space before the free space pointer are contiguously occupied. This is called compacting the page.*

Page Format for Variable Length Records

UPDATE
1. If new record smaller than the old record, update the record and the record size in the directory 2. Else, if enough free space, insert in the free space and update directory (offset & size) 3. Else, try to compact the page 4. If still not enough space, allocate a new page and write the new record in the new page. But record ID needs to be changed. We will make the original record as a forwarding record. The forwarding record only contains header information indicating that it is a forwarding record, and the record ID of the new record.