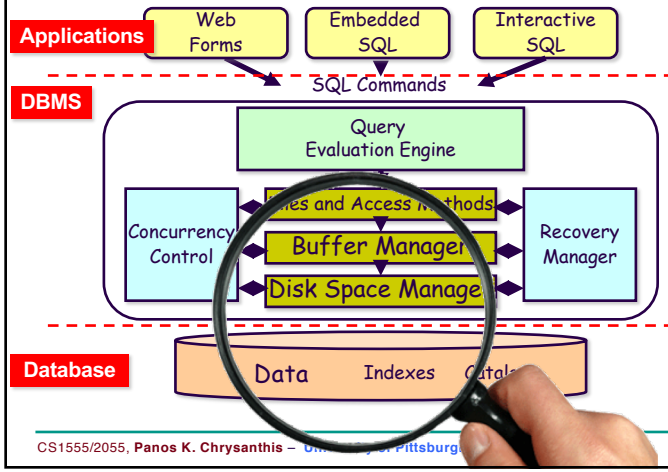
Data Storage

Database Management System (DBMS)



Data Storage

- Two DBMS fundamental questions?

1. How do we store and manage very large volumes of data?

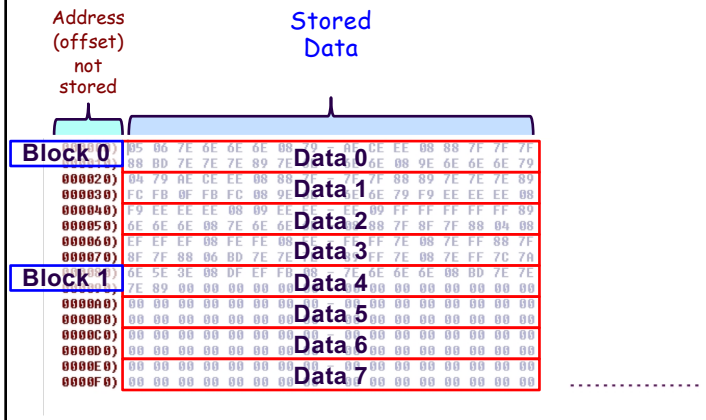
2. What representation and data structures best support efficient manipulation of data?
   - RAM model vs. I/O model of Computation

Storage Hierarchy

Speed; $$

- Primary Storage: random access; volatile
  - *Cache* - on board or level-2 cache
  - *Main memory*
- Secondary storage: random access; non-volatile
  - Flash-based or Solid State Disk
  - *Magnetic disk*
    - Virtual Memory (Main-memory DBS), File System, DBMS
- Tertiary storage: non-volatile
  - *Optical disk* / juke boxes – random access
  - *Magnetic cartridge* / tape silos – seq. access

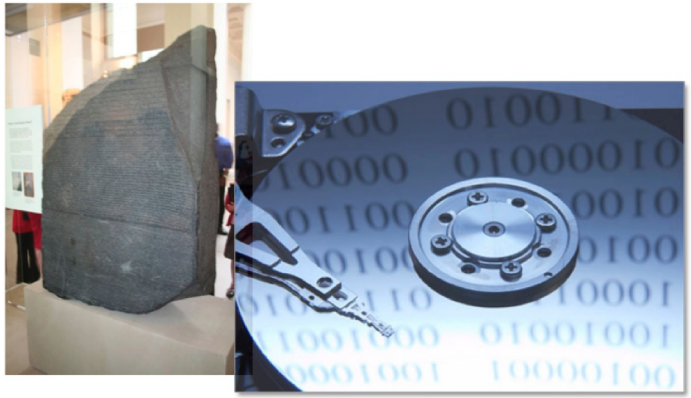
Data (Records/Columns) on Disk



Improving Access Time ?

- What are the costs in I/O requests?
- I/O cost = *secondary storage access* + queuing delays
  - *disk access* = seek time + rotational delay + transfer time
    - Block transfer, Buffering and Prefetching, Multiple/RAID disks
  - Queuing delays are reduced with scheduling
    - Elevator (Scan) Algorithm and its variations
- *Data access cost* is reduced with data organization

But first, how is data stored?



Two Store Approaches

Row Stores

Column Stores

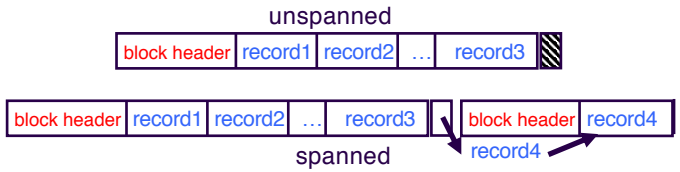
Example Table

```
CREATE TABLE Student (
  SID         INTEGER
  name        CHAR(16),
  address     CHAR(130),
  gender      CHAR(1),
  birthdate   DATE
);
```

- DATE: 10-char string YYYY-MM-DD
  Fixed-length character string char(10)
  - example: 2002-09-15

Row Stores

- Store fields in one record contiguously on disk with word allignment
- Use small (e.g., 4K) disk blocks



Column Stores

SELECT SID, gender
FROM Students;

gender	Birthday	Name	SID	Address

File Types

- Unordered files
- Ordered files
- Clustered files
- Hash files

File header or descriptor includes:
- field names and their data types
- address of the file block on disk

Unordered Files

- The simplest file structure: records are stored in no particular order
- Also called: **Heap**, Pile, or Random File
- New records are inserted at the **end** of file
  1. The last disk block is copied into buffer (i.e., memory)
  2. New record is added
  3. Block is rewritten back to disk
- Record insertion is quite efficient

Example of Heap File

```
CREATE TABLE deposit (
  account_number   CHAR(10),
  branch_name      CHAR(22),
  balance          REAL
)
```

Block 1	Record 0	A-102	Oakland	400
	Record 1	A-305	Shadyside	350
	Record 2	A-101	Downtown	700
	Record 3	A-222	Squirrel Hill	500
Block 2	Record 4	A-217	Shadyside	900
	Record 5	A-110	Waterfront	340
	Record 6	A-257	Oakland	600
	Record 7			

Heap Files: Insert

Insert Record 7

Record 0	A-102	Oakland	400
Record 1	A-305	Shadyside	350
	A-101	Downtown	700
Record 3	A-222	Squirrel Hill	500
Record 4	A-217	Shadyside	900
Record 5	A-110	Waterfront	340
Record 6	A-257	Oakland	600
Record 7			

Record 0	A-102	Oakland	400
Record 1	A-305	Shadyside	350
	A-101	Downtown	700
Record 3	A-222	Squirrel Hill	500
Record 4	A-217	Shadyside	900
Record 5	A-110	Waterfront	340
Record 6	A-257	Oakland	600
Record 7	A-354	Oakland	420

Heap Files: Delete

remove Record 2

Record 0	A-102	Oakland	400
Record 1	A-305	Shadyside	350
Record 2	A-101	Downtown	700
Record 3	A-222	Squirrel Hill	500
Record 4	A-217	Shadyside	900
Record 5	A-110	Waterfront	340
Record 6	A-257	Oakland	600
Record 7	A-403	Downtown	250

Record 0	A-102	Oakland	400
Record 1	A-305	Shadyside	350
⊘			
Record 3	A-222	Squirrel Hill	500
Record 4	A-217	Shadyside	900
Record 5	A-110	Waterfront	340
Record 6	A-257	Oakland	600
Record 7	A-403	Downtown	250

Heap Files: Insert & Delete (optimized)

remove Record 2

Record 0	A-102	Oakland	400
Record 1	A-305	Shadyside	350
⃠			
Record 3	A-222	Squirrel Hill	500
Record 4	A-217	Shadyside	900
Record 5	A-110	Waterfront	340
Record 6	A-257	Oakland	600
Record 7	A-403	Downtown	250

add Record 8

Record 0	A-102	Oakland	400
Record 1	A-305	Shadyside	350
Record 8	A-354	Oakland	420
Record 3	A-222	Squirrel Hill	500
Record 4	A-217	Shadyside	900
Record 5	A-110	Waterfront	340
Record 6	A-257	Oakland	600
Record 7	A-403	Downtown	250

-- **Periodic reorganization:** records are packed by removing deleted records

Properties of Unordered Files

- File records are inserted at the end of the file or in any file block with free space.
- Thus, insertion is efficient.
- To search for a record, a ***linear search*** through the file records is necessary which is quite expensive.
- Reading the records in order of any field requires sorting the file records.

Ordered Files

- Also called ***sequential*** files.
- File records are kept sorted by the value of an ***ordering*** key which has unique value (e.g., primary key)

```
CREATE TABLE deposit (
  account_number   CHAR(10),
  branch_name      CHAR(22),
  balance          REAL
)
```

- Fixed-length records
- 10 + 22 + 8 = 40 bytes

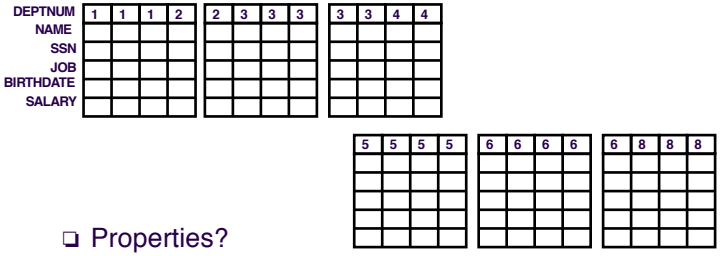
Record 0	A-101	Downtown	700
Record 1	A-102	Oakland	400
Record 2	A-205	Shadyside	350
Record 3	A-217	Shadyside	900
Record 4	A-222	Squirrel Hill	500
Record 5	A-310	Waterfront	340
Record 6	A-357	Oakland	600
Record 7	A-403	Downtown	250

Properties of Ordered Files

- Insertion is expensive: records must be inserted in the correct order.
- Deletion?
- Search for a record on its ordering field value is quite efficient (***binary search algorithm***).
- Search for a record on a non-ordering field?
- Reading the records in order of the ordering field is also quite efficient.
- Reading the records in any order?

Clustered Files

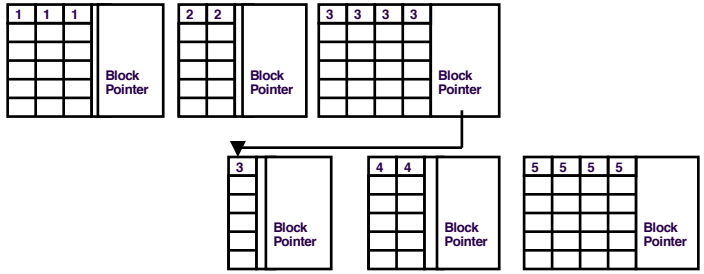
- Order files with ordering field which is **not a key**
- Ordering field has not a unique value



- Properties?
- Purpose ?

Clustered File: Example 2

- For efficient insertion, deletion and search, each group of records with different ordering field is stored on separate block



Hash Files

- Also called **direct** files
- External hashing maps keys to disk blocks
  - Works similar to internal hashing
  - Static hashing
- Dynamic file expansion
  - Linear hashing
  - Extendible hashing

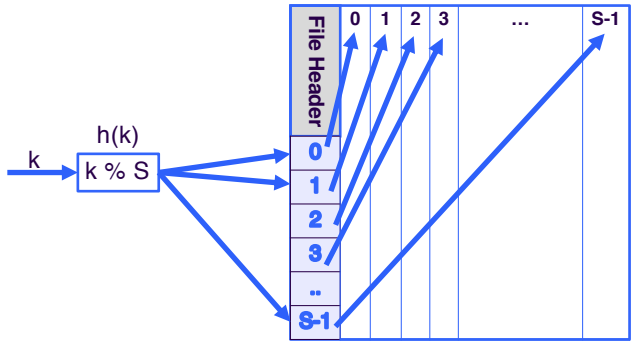
Static Hashing

- Hashing converts the key of a record into an address in which the record is stored.
- An external ***hash*** function maps the key to the relative address of a ***bucket*** in which the record is stored.
  - if a file is allocated ***s*** buckets, the hash function must convert a key ***k*** into the relative address of the block:

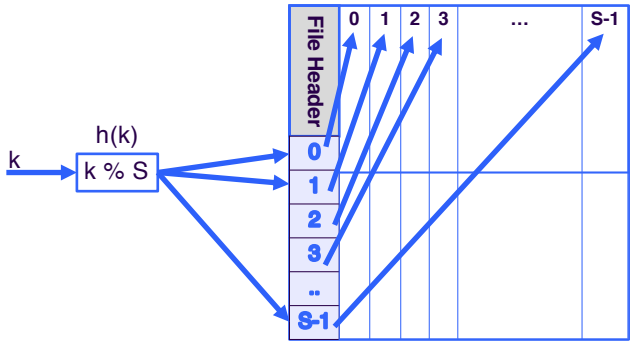
$$h(k) \in \{0, ..., s-1\}$$

- A bucket is either one disk block or a cluster of contiguous blocks
- A table stored in the header of the file maps relative bucket numbers to disk block addresses.

Static Directed File



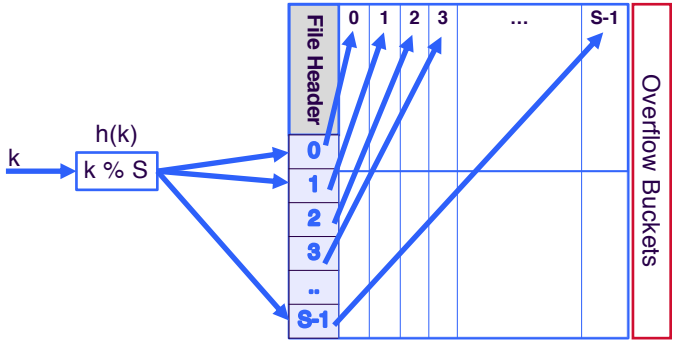
Static Directed File



Collision

- Insertion of a new record may lead to ***collision***.
  - No space in ***B = h(k)***
- *Probing* or *conflict resolution*:
  - *Open Addressing (Rehashing)*: If bucket h(k) is full, use another hash function until a bucket with a free space is found.
    E.g., *linear probing*
    α = ***h(key)*** = ***key mod s***
    rh(α ***)*** = ***h (*** α ***+1)***
    Not a very good technique for databases (Why?)
  - *Chaining*: Use overflow buckets.

Static Directed File



Hashing Functions

- A good hash functions must
  1) be computed efficiently
  2) minimize the number of collisions by spreading keys around the file as evenly and uniform as possible.
- Example of good functions
  - truncation
  - division: h(key) = key mod s
  - Mid-square
  - Folding or partitioning
  - Other ad hoc methods
- Order preserving hash functions:
  - Maintain records in order of hash field values

Pros and Cons of Hashing

- Excellent performance for searching on ***equality on the key*** used for hashing (assuming low density).
- Records are not ordered (heap files).

  ⇒ Any search other than on equality is very expensive (linear search or involves sorting).
- Prediction of total number of buckets is difficult.
  - allocate a large space.
  - estimate a ``reasonable'' size and periodically reorganize.

Dynamic Hashing Methods

- Allow the file size to change as records are added or deleted.
  - Linear Hashing
    - No additional structure
  - Extendible Hashing
  - Binary Hashing