Data Storage Structures

Outline

- DB organization in files
- File organization in disk
- Database buffer
- Data dictionary
- Column-oriented storage

File Organization

- The **database** is stored as a collection of *files*, maintained by the OS.
- Each **file** is:
  - Sequence of *records* (DB viewpoint)
  - Logically partitioned into *blocks* (storage viewpoint)
- **Records** are a sequence of fields
- **Blocks** are fixed-length units of storage allocation and data transfer

  Block size: usually 4-8 KB
- A block may contain several *records*.
  - "*no record is larger than a block*"
  - "*each record is entirely contained in a single block*"
- Files can have **fixed** or **variable-length** records

Exercise: Record-block relationship

- Why does the allocation of records to blocks affect DB system's performance so bad?

Fixed-Length Records

- All records have always the same length
- Attributes always in the same order
- Advantages:
  - Easy to access
  - How many records/block? Just maths
- Disadvantages:
  - Space wasted

Record:

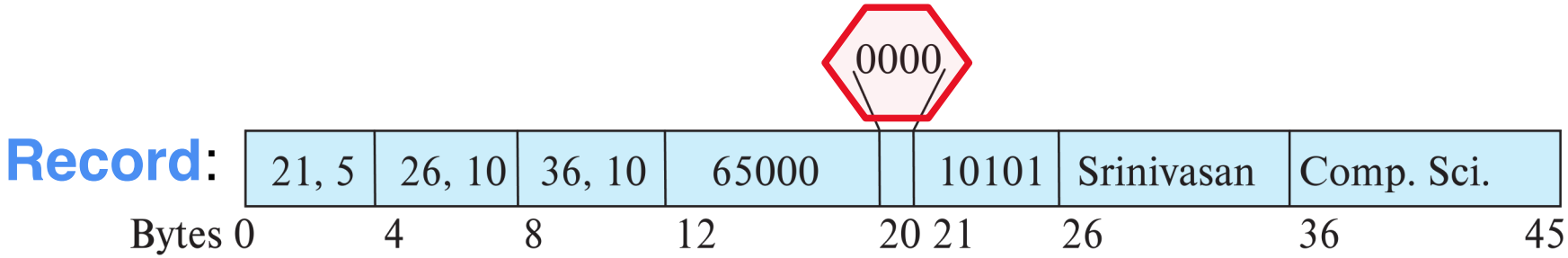
10101	Srinivasan	Comp. Sci.	65000

Variable-Length Records

varchars

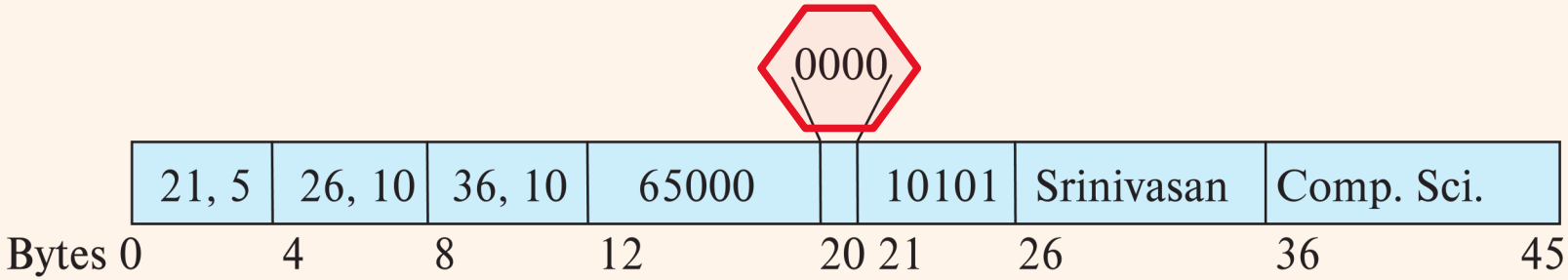
- Challenges:
  - Allow individual attributes to be extracted easily
  - Store variable-length records within a block such that can be easily extracted

- Each record has **2 parts**:
  - Header with fixed-size (offset, length) information about variable-length attributes
  - Attributes (first, fixed length att.; then, variable length att.)
  - **Null-value bitmap**: indicates which attribute is null
    - Save space: no need to save anything else for null-valued attributes



Exercise: Reorganize variable-length records

- Can you modify the record representation such that the only overhead for a null attribute is the single bit in the null bitmap?



Files with Fixed-Length Records

- Simplest approach: A different file for each relation
  Each file only has records of one type
  - Store record *i* starting from byte $r * (i - 1)$, where *r* is the record length (**fixed**).
- Record access is simple

Allocate only $\lfloor b/r \rfloor$ records per block, to avoid having records across blocks
$b \equiv$ block size

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Files with Fixed-Length Records

- Issue: deletion of records is difficult. If record *i* is deleted, we can:
  - move records *i + 1,..., n* one position back (to *i,..., n – 1*)
  - move record *n* to position *i*
  - link all free records on a *free list* (do not move records)

Delete record *i=3*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Files with Fixed-Length Records

- Issue: deletion of records is difficult. If record *i* is deleted, we can:
  - move records *i + 1,..., n* one position back (to *i,..., n – 1*)
  - move record *n* to position *i*
  - link all free records on a *free list* (do not move records)

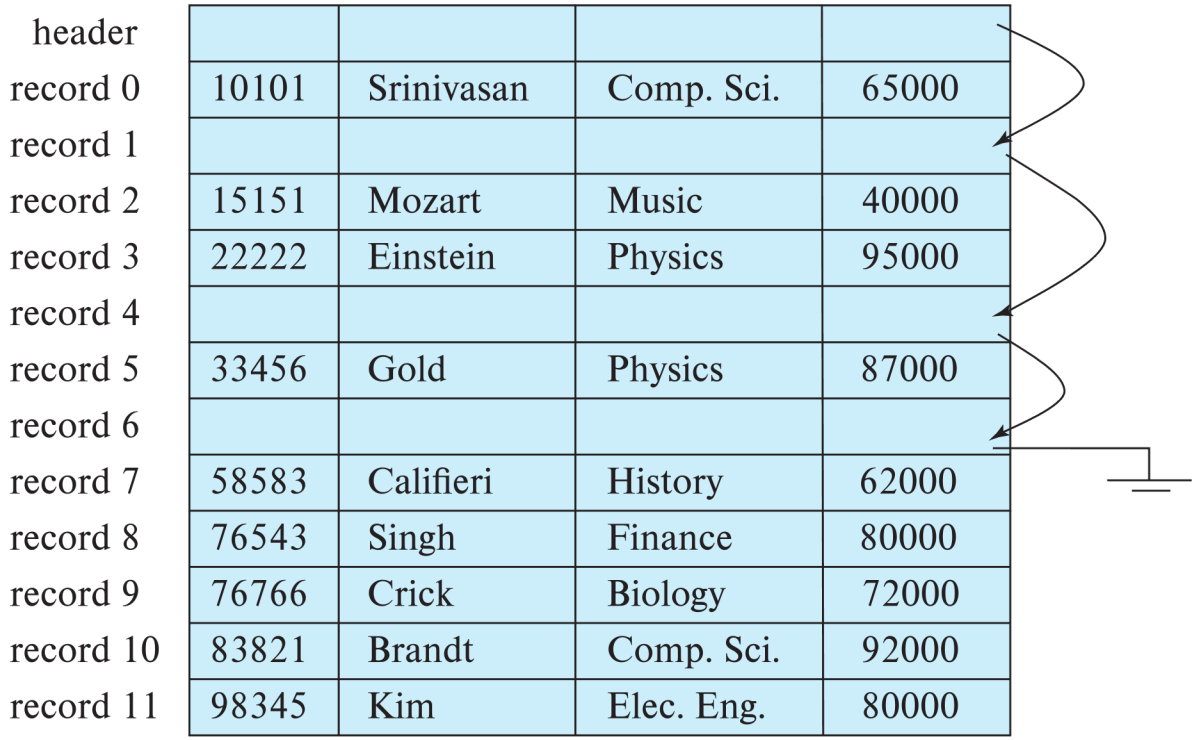
Delete record *i=3,*
Move record *11* to *3*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Files with Fixed-Length Records

- Issue: deletion of records is difficult. If record *i* is deleted, we can:
  - move records *i + 1,..., n* one position back (to *i,..., n – 1*)
  - move record *n* to position *i*
  - link all free records on a *free list* (do not move records)

A **header** to store info. about the file:
- Pointer to first free position

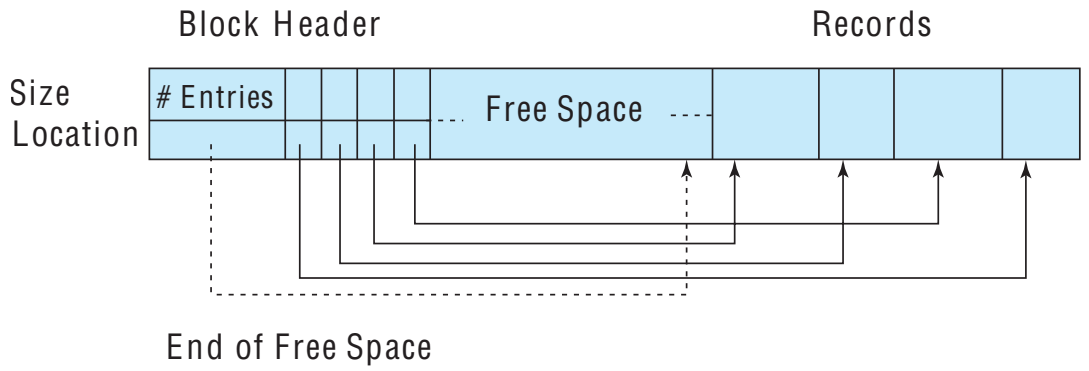


header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Files with Variable-Length Records

- **Slotted Block Structure**
  - Address the problem of storing variable-length records **in a block**
  - Block starts with a **header** that contains:
    - No. record entries
    - End of *free space* in the block
    - Array with **location** and **size** of each record
  - At the end of the block, **records** are saved contiguously.

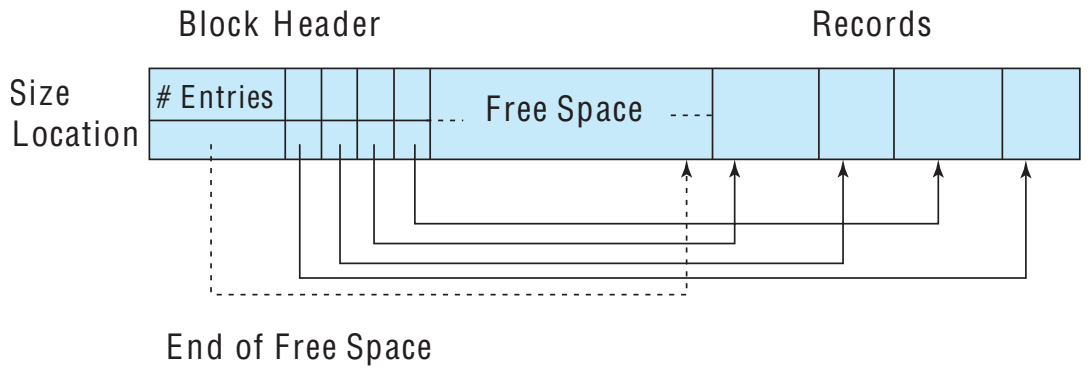
Block:



Files with Variable-Length Records

- **Slotted Block Structure**
  - New records are saved at the end of *free space*, and linked from header
  - Deletes and updates change both header and records.
    - Records are moved so that no *free space* is between records.
    - New pointers might be necessary at header

Block:



Ordering Records in Files

Organization types:

- **Heap**: without order. Records can be placed anywhere in the file where there is free space

- **Sequential**: records are placed sequentially, according to the search-key value
  - **Search-key**: (set of) attributes, not need to be a key

- **Multitable clustering** mixes records of several different relations in the same file
  Store related records on the same block to minimize I/O and common joins

- Others: **B$^+$-tree**, **Hashing**, etc.

Heap File Ordering

- Records can be placed anywhere in the file where there is free space
- Records usually do not move once allocated

- **Free-space map** to track blocks with *free space*
  - Array with 1 entry per block
    - Each entry saves fraction of free space in the corresponding block

4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6

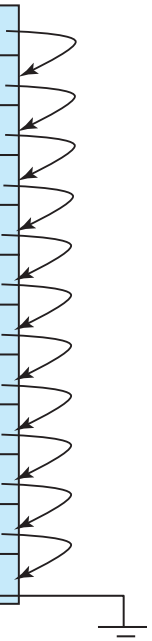
16 blocks, 3 bits/entry (0-7)

  - If a new record comes in and no block has free space, a new block is allocated.
  - Should be updated after inserts/deletes/updates
    Actually, it is only done periodically

Sequential File Ordering

- The records are ordered by a **search key**
- Suitable for apps that process sequentially in order the entire file
- Use pointers in each record

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

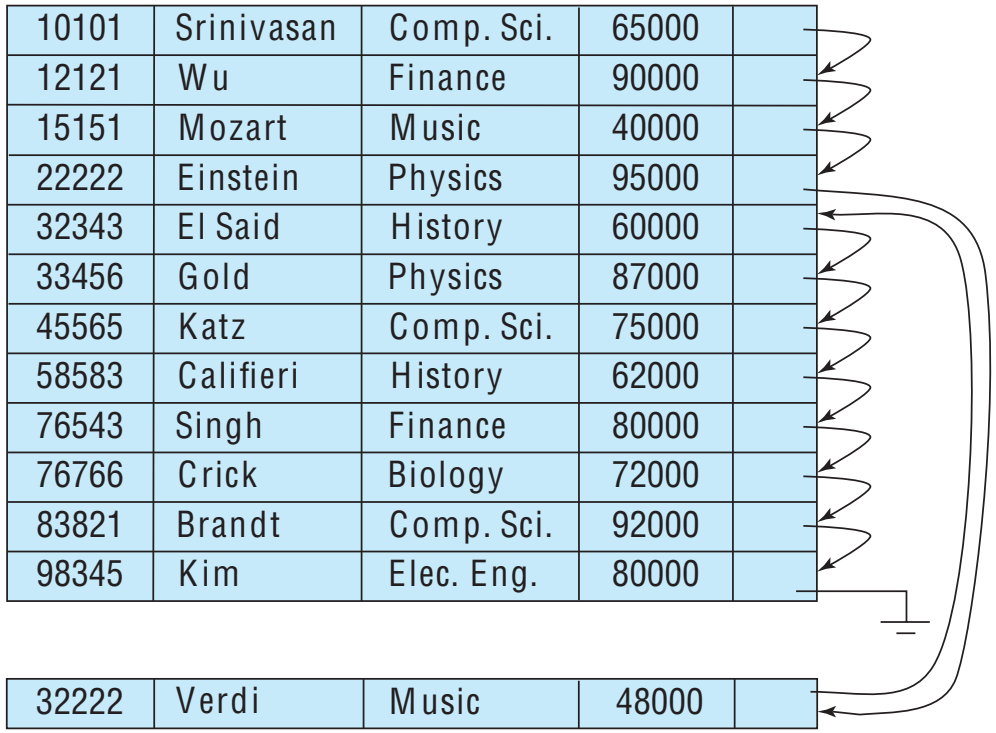


Sequential File Ordering

- **Deletion**: use pointer chains
- **Insertion**:
  1. locate the position where the record is to be inserted
     - if there is free space in the same block, insert it there
     - if not, insert it in an *overflow* block
  2. Update pointer chain

Need to **reorganize** the file from time to time as logical and physical record orderings become too different

Sequential processing, very inefficient



Multitable Clustering File Ordering

Store several relations (multitable) in one file:

Comp. Sci.	Taylor	100000	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000

multitable clustering of *department* and *instructor*

You could actually skip these values

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

department

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

instructor

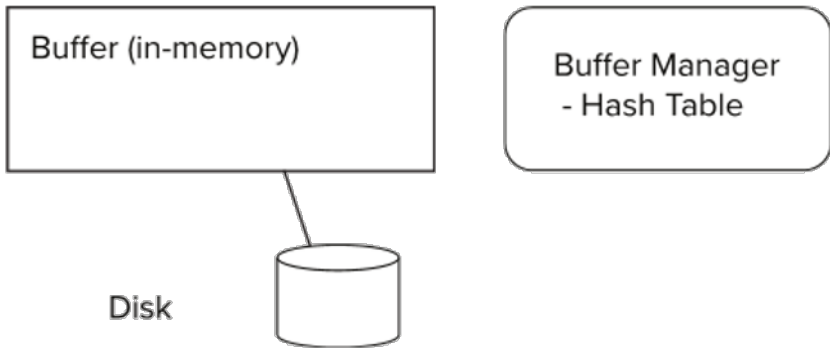
- Two relations clustered on a **cluster key** (*dept_name*).
- Each record contains the identifier of the relation to which it belongs.

Access to disk

- DBMS seek to minimize no. *block transfers* between disk and memory
  - Simplest idea: keep as many blocks as possible in memory

Blocks:
units of storage allocation and data transfer.

- Usually, implements a **buffer**:
  - Portion of main memory available to store copies of disk blocks
  - **Buffer manager**: subsystem that allocates buffer space



Buffer Manager

1. DBMS makes *block requests* to buffer manager
   - If the *block* is already in the buffer, *buffer manager* finds it
   - If the *block* is not in the buffer, the *buffer manager*:
     1. allocates space in the buffer for the block
     2. reads the block from the disk to the buffer
2. The *block* address in main memory is returned

What happens if the buffer is full??
- Remove a block
- Write it to disk (if necessary)

- To anticipate removal, most DBs continually look for updated blocks and write them to disk
  - Locking and pinning are techniques used to identify in-use blocks

Exercise: Finding a record in the buffer

- From previous slide:

  "If the block is already in the buffer, buffer manager finds it"

- Which data structure would you use to quickly find if a block resides in the buffer (and where in it)?

Buffer-Replacement Policies

- Minimizing disk accesses while providing access to blocks
  - **Least recently used** (LRU) strategy: replace the LRU block
    - Acceptable for OS; not so much in DB systems
      E.g., computing a join shows the bad access pattern of LRU

```
for each tuple tr of r do
    for each tuple ts of s do
        if tr[join_attr] = ts[join_attr] then
            ...
```

  - **Toss-immediate** strategy: removes a block as soon as the final tuple of that block is processed
    Records *tr* of *r* in the previous example won't be accessed again
  - **Most recently used** (MRU) strategy: replace the MRU block
    Records *ts* of *s* in the previous example won't be accessed again for a long time

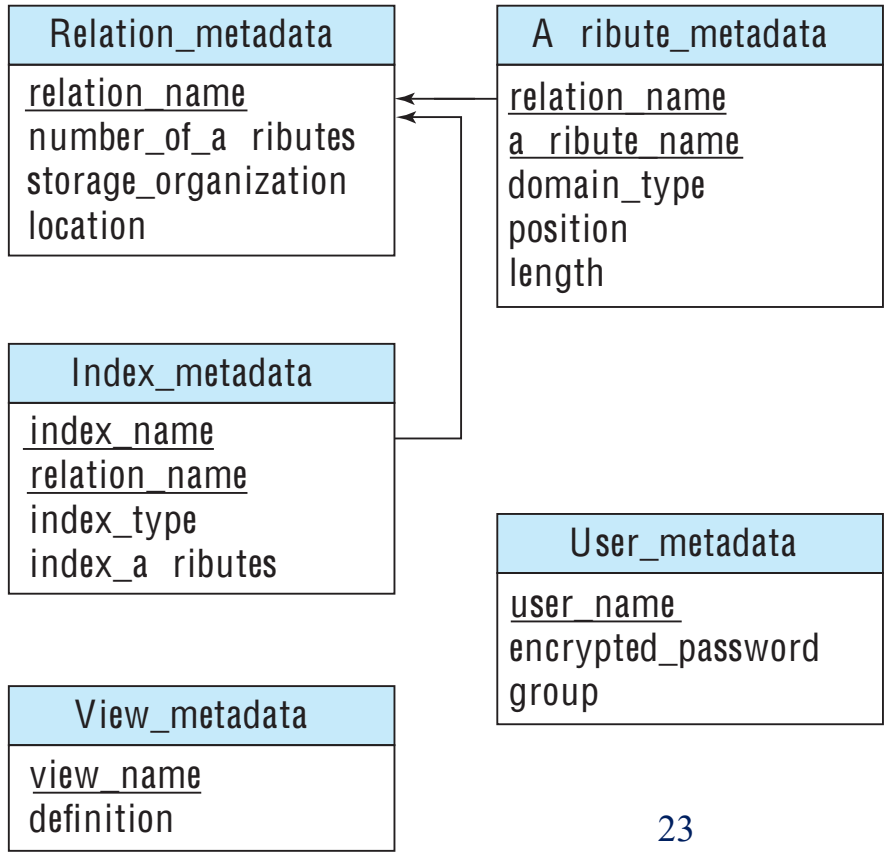
No single strategy handles well all the possible scenarios

OTHER CONCEPTS OF INTEREST

Data Dictionary Storage

- **Data dictionary** (or **system catalog**) stores *metadata* such as…
  - *Relations*
    - names of relations; names, types and lengths of attributes
    - names and definitions of views
    - integrity constraints
  - *Users, passwords* and *authorizations*
  - Statistical and descriptive data
    - No. tuples in each relation
    - No. different values for each attribute
  - *File organization*
    - Physical location of relation (filename, blocks, etc.)
  - *Indices*
    - Name and type
    - Relation and attributes involved

Usually loaded in memory at BD startup



Column-Oriented Storage

- A.k.a., **columnar representation**
  As opposed to classical row-oriented storage
- Store relation attributes separately
- Same order of tuples in all files

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- Increasingly used in data-warehousing apps.
  Queries are primarily data analysis queries, few columns, no deletion/update, …
- Not appropriate for transaction processing

Paradigm change that impacts indices and query processor

Data Storage Structures