

CS 5/7330

File structures

File structures for relational databases

- Relational database
 - A set of tables
 - Each table has a set of tuples (records)
- Assumption:
 - Data are stored in files
 - Files are stored on disk
 - Data on files are divided into fixed size blocks (pages)
 - For now, assume each table corresponds to a file

File structures for relational databases

- Each table has a set of tuples / records
- Each tuple is made up of a fixed set of attributes
- Attributes have types, which require bytes to store
 - E.g. integer – 4 bytes,
- Some types allow variable length
 - E.g. VARCHAR,
 - But VARCHAR(50) can be viewed as fixed size of 50
- Thus tuples can be of fixed or variable length

Page structure: Fixed length records

- Assume each record requires n bytes
 - Store record i starting from byte $n * (i - 1)$, where n is the size of each record.
 - Each record has a number associated with it

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

What other bookkeeping info is needed?

Problem of this approach?

Fixed length records

- Problem 1 : Records may pass page boundaries
- E.g. A page has 100 bytes, each record has 12 bytes
 - Record number 9 will pass two pages
 - What's the problem with that?
- Solution, limit size so no record cross boundaries:
 - E.g. in the previous case, each page can only store $\text{floor}(100/12) = 8$ records
 - Trade-off: internal fragmentation

Fixed length records

- Insert a record
 - Just add a record at the end of the file
- Delete a record
 - 3 options
 - Shift all records up a slot
 - Shift the last record to fill in a page
 - Other solutions?
 - For the first 2 cases, do not shift record across pages (why?)

Move the last



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

Shift everything up 1



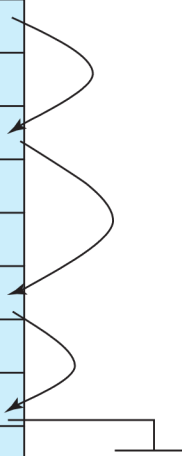
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8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

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

Fixed length records

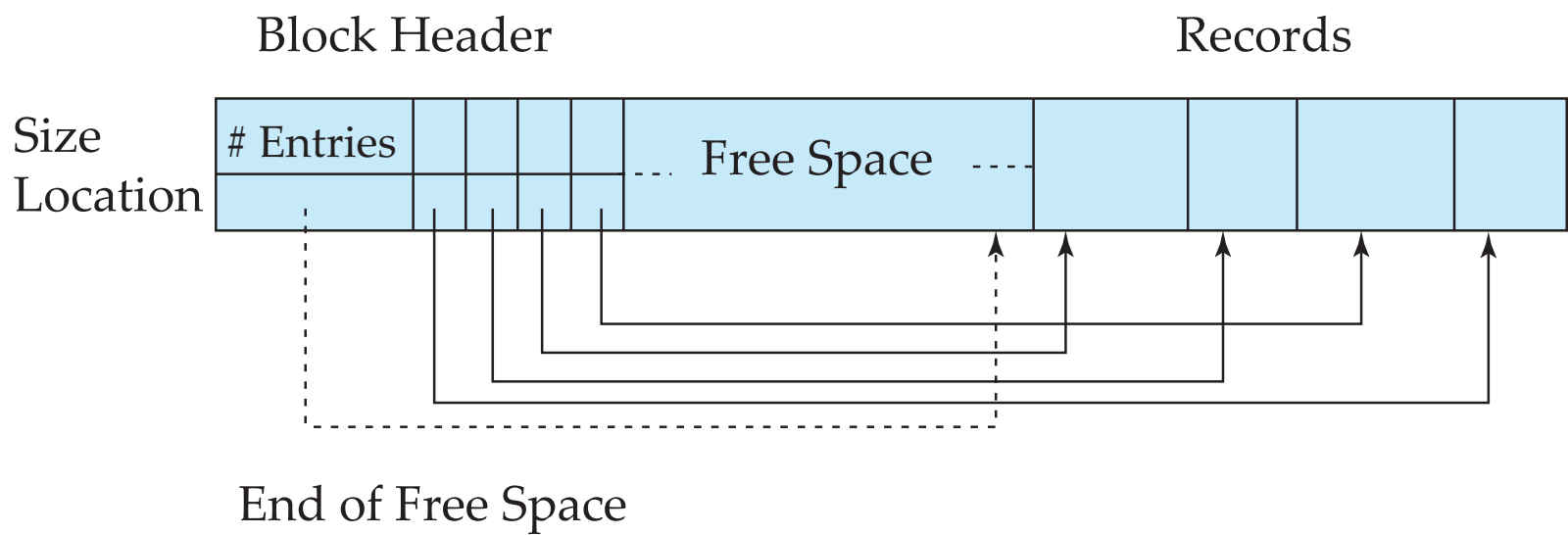
- Third solution
 - Do not move records, but create a free record list
 - Book-keeping info
 - For each record, need a pointer pointing to the first free record
 - Each free block will need to have a designation that it is free
 - And a pointer pointing to the next free record
 - Also a way to denote end of the list
- Advantages / Disadvantages?

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



Page structure: Variable length record

- Cannot predict how many records are there in a page
- And each page can be of variable length
- Solution
 - A slotted page header containing
 - # of records in that page
 - The starting location and size of each record
 - Why do we need size?
 - Notice that typically we put a limit on number of records
 - Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.



Large objects (Blobs)

- Records must be smaller than pages
- Alternatives:
 - Store as files in file systems
 - Store as files managed by database
 - Break into pieces and store in multiple tuples in separate relation
 - PostgreSQL TOAST

Organization (ordering) of records

- How should records in a file be organized (ordered)?
- One file per table:
 - **Heap** – record can be placed anywhere in the file where there is space
 - **Sequential** – store records in sequential order, based on the value of the search key of each record
- With indexing option:
 - **B⁺-tree file organization**
 - Ordered storage even with inserts/deletes
 - **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed
- Multiple tables per file:
 - In a **multitable clustering file organization** records of several different relations can be stored in the same file
 - Motivation: store related records on the same block to minimize I/O

Heap file

- Records can be placed anywhere in the file where there is free space
- Records usually do not move once allocated
- Important to be able to efficiently find free space within file
- **Free-space map**
 - Array with 1 entry per block. Each entry is a few bits to a byte, and records fraction of block that is free
 - In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free

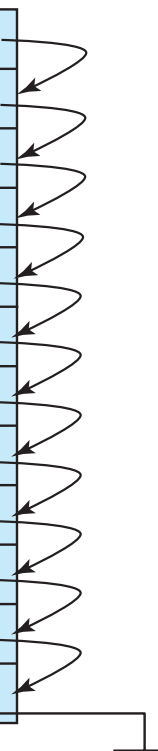
4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---
 - Can have second-level free-space map
 - In example below, each entry stores maximum from 4 entries of first-level free-space map

4	7	2	6
---	---	---	---
- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)

Sequential file

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a **search-key**
- We sometime call this organization clustered via a search-key

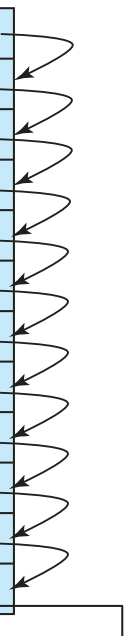
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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



Sequential file

- Deletion – use pointer chains
- Insertion – locate the position where the record is to be inserted
 - if there is free space insert there
 - if no free space, insert the record in an **overflow block**
 - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order

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	



The diagram illustrates a pointer chain mechanism. A series of curved arrows on the right side of the table rows point from the empty fifth column of each row to the row immediately below it. A final arrow from the bottom row points to a ground symbol, represented by a horizontal line with a vertical line and a small triangle at its base.

Heap File vs. Sequential File

- What's the difference?
- Heap file:
 - Relatively little insertion cost
- Sequential file:
 - Cost involved when insertion (to maintain sorted)
 - Or have to sort every time (expensive)
- So why even think about sequential file?
 - It helps with some queries

Heap File vs. Sequential File

- Example: (used for the rest of the slides)
 - Two tables
 - Department(dept_name, building , budget)
 - Assume each tuple is 40 bytes
 - Dept_name is key
 - Instructor(id, name, dept_name, salary)
 - Assume each tuple is 50 bytes
 - Id is key, dept_name is foreign key (referencing Department)
 - Assume data are stored on disk
 - Each page has 1050 bytes
 - Assume 50 bytes are needed for overhead information
 - Assume the pages are fully filled
 - Now assume Department has 100,000 tuples
 - So number of pages = $100,000 * 40 / (1050 - 50) = 4,000$ pages
 - Assume Instructor has 400,000 tuples
 - So number of pages = $400,000 * 50 / (1050 - 50) = 20,000$ pages

Heap File vs. Sequential File

- Now consider the following query

`SELECT * FROM Instructor where id = "1997"`

- Now if you have a heap file
 - You have to look for each tuple
 - You can stop when you find the tuple (why?)
 - Worst case: no instructor has such ID
 - In this case, you have to search the whole file
 - Total cost = 20,000 pages
 - However, if Instructor table is sorted via id:
 - Then can apply binary search
 - Total cost = $\text{ceiling}(\log_2(20000)) = 15$ pages

Heap File vs. Sequential File

- Now consider the following query
`SELECT * FROM Instructor where id = "1997"`
- However, for magnetic disk, we need to worry about seek/rotation
 - For binary search, subsequent searches are not on consecutive pages
 - Thus need rotate (or even seek)
 - Now suppose reading a page take s seconds, and a rotate/seek take $100s$ second
 - Then time for heap file = $100s + 20000s$ (1 seek + 20000 read) = $20100s$
 - Time for sequential file = $100s + \log_2 20000 * (100s + s) = (1 \text{ seek} + 15 \text{ rotate and read}) = 1512s$
 - Still win, but not as big a gap

Heap File vs. Sequential File

- Now consider the following query

`SELECT * FROM Instructor where id = "1997"`

- What is the file is smaller (e.g. 400 pages)
 - Then time for heap file = $100s + 400s = 500s$
 - Time for sequential file = $100s + \text{ceiling}(\log_2(400)) * (100s + s) = 706s$
 - Binary search on sequential file is not worth it

Heap File vs. Sequential File

- Now consider the following query

```
SELECT * FROM Instructor where salary > 1000
```

- Sorting through ID will not help
- In either case, one MUST read the whole file no matter what
- Sequential file doesn't help

Addition: How to sort a file

- Suppose you do want to sort a file on the disk
 - And store the results on the disk
 - E.g. `SELECT * from instructor ORDER BY salary`
- Simple solution
 - Read the whole file into main memory
 - Sort it using your favorite sorting algorithm
 - Write the whole file back to the disk
- What if the file is too large to fit into main memory?

Addition: How to sort a file

- What if the file is too large to fit into main memory?
 - Remember comparing value can only occur when both tuples are in main memory
 - So we cannot just do a one pass sorting
 - Sorting need to be done in stages
 - Merge sort

Sorting a file: Merge sort

- Merge sort
 - Start by breaking tuples into list of one tuple
 - At each iteration, merge pairs of list until half the list remains
 - Repeat until all lists are merged
- We use the same basic algorithm but with some modification

Sorting a file: Merge sort

- Assumptions
 - We have a file with N pages
 - We have B pages of main memory available ($B < N$)
 - Number of pages read/written are the main cost (we assume sorting in main memory is much faster than the time can be ignored)

Sorting a file: Merge sort

- Step 1: Creating the initial list
 - Instead of creating list of one tuples, we can use the amount of main memory available
 - So read B pages from the file at a time
 - In the memory, sort the B pages using any efficient sorting algorithms
 - Write the B pages back to the disk somewhere
 - Repeat until every page is read and written
 - Total page read/written for this step = $2N$
 - Each page is read/written once
 - At the end of this step: $\lceil N/B \rceil$ list is formed, each (possibly except the last one, have length B)

Sorting a file: Merge sort

- Step 2: Merging 2 lists
 - Divide the B pages of main memory into two groups
 - Pick 2 list, read the first part of one list into one group, and the first part of the second list into the other
 - Use the standard merge function in mergesort to merge
 - Need to write the merged page to the disk along the way (unless there is output buffer available)
 - When one list is exhausted, than read the rest of the list
 - Time for an iteration = $2N$ (once again, each page is read and written once)

Sorting a file: Merge sort

- Step 3: Until done
 - The total number of iterations = $\text{ceiling}(\log_2(N/B))$
- So the total page read/written
$$2 * (1 + \text{ceiling}(\log_2(N/B))) * N$$

Sorting a file : Merge Sort (example)

- Suppose you have a file with 12,800 pages
- Assume you have 200 page of memory available.
- First step:
 - Read the first 200 pages of the file into main memory
 - Sort it
 - Write the sorted list somewhere on the disk (on consecutive blocks)
- At the end of the first step, there are a total of $12,800 / 200 = 64$ sorted segments, each of them contains 200 pages

Sorting a file : Merge Sort (example)

- Second step
 - Put the 64 segments into groups of 2 (32 groups)
 - For each group, apply the merge algorithm in merge sort
 - However need to remember merging can be done only when data is in main memory
 - So for each group
 - Divide the main memory equally into 2 part (100 page each)
 - Read the first 100 page of each segment
 - Apply the merge algorithm, until one of the segment (100 page) is used up
 - Then read in the rest of that segment into the main memory and continue
 - Continue merging (if the first 100 of the other segment is used up), bring in the next 100 page
 - Until one of the segments is completely merged
 - Then write the rest of the remaining segment to the disk
 - Now a sorted segment of $200 + 200 = 400$ pages is formed
 - Repeat for all groups, so now we have 32 sorted segments, each with 400 pages

Sorting a file : Merge Sort (example)

- Third step (next iteration)
 - Put the 32 segments into groups of 2 (16 groups)
 - For each group, apply the merge algorithm in merge sort
 - However need to remember merging can be done only when data is in main memory
 - So for each group
 - Divide the main memory equally into 2 part (100 page each)
 - Read the first 100 page of each segment
 - Apply the merge algorithm, until one of the segment (100 page) is used up
 - Then read in the next 100 pages of that segment into the main memory and continue
 - Continue merging (whenever one of the 100 pages part of a segment is used up), bring in the next 100 page
 - Until one of the segments is completely merged
 - Then write the rest of the remaining segment to the disk
 - Now a sorted segment of $400 + 400 = 800$ pages is formed
 - Repeat for all groups, so now we have 16 sorted segments, each with 800 pages

Sorting a file : Merge Sort (example)

- Subsequent steps (continued)
 - Merge 16 segments of 800 pages by pairs , forming 8 segments of 1600 pages
 - Merge 8 segments of 1600 pages by pairs, forming 4 segments of 3200 pages
 - Merge 4 segments of 3200 pages by pairs, forming 2 segments of 6400 pages
 - Merge the 2 remaining segments, forming 1 sorted file

Sorting a file: Merge sort

- Modification:
- Since we have B buffers available
- We can actually merge more than 2 list at a time
- In the extreme, we merge B-1 list at each step
 - Pick B-1 list
 - Assign one page in main memory to each list
 - Read the first page of each list into the corresponding memory
 - Continue the merge process
 - Time taken: $2 * (1 + \text{ceiling}(\log_{(B-1)}(N/B))) * N$

Sorting a file: Merge sort

- In theory, it works well
- But issues
 - We haven't factor in output need
 - One probably should not write a tuple to storage every time a single merge step (comparing 2 (or B) tuples) is taken
 - Why?
 - So need to assign memory pages for output
 - Also, for each read/write operation, there is a potential seek/rotate for hard drive
 - The more segments we merge with one step -> less space for each segment -> there are more reads require to read the segments -> more seeks.

Multitable Clustering File Organization

department

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

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

multitable clustering
of *department* and
instructor

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 File Organization

- good for queries involving *department* ⋈ *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation

Partitioning

- **Table partitioning:** Records in a relation can be partitioned into smaller relations that are stored separately
- E.g., *transaction* relation may be partitioned into *transaction_2018*, *transaction_2019*, etc.
- Queries written on *transaction* must access records in all partitions
 - Unless query has a selection such as *year=2019*, in which case only one partition is needed
- Partitioning
 - Reduces costs of some operations such as free space management
 - Allows different partitions to be stored on different storage devices
 - E.g., *transaction* partition for current year on SSD, for older years on magnetic disk

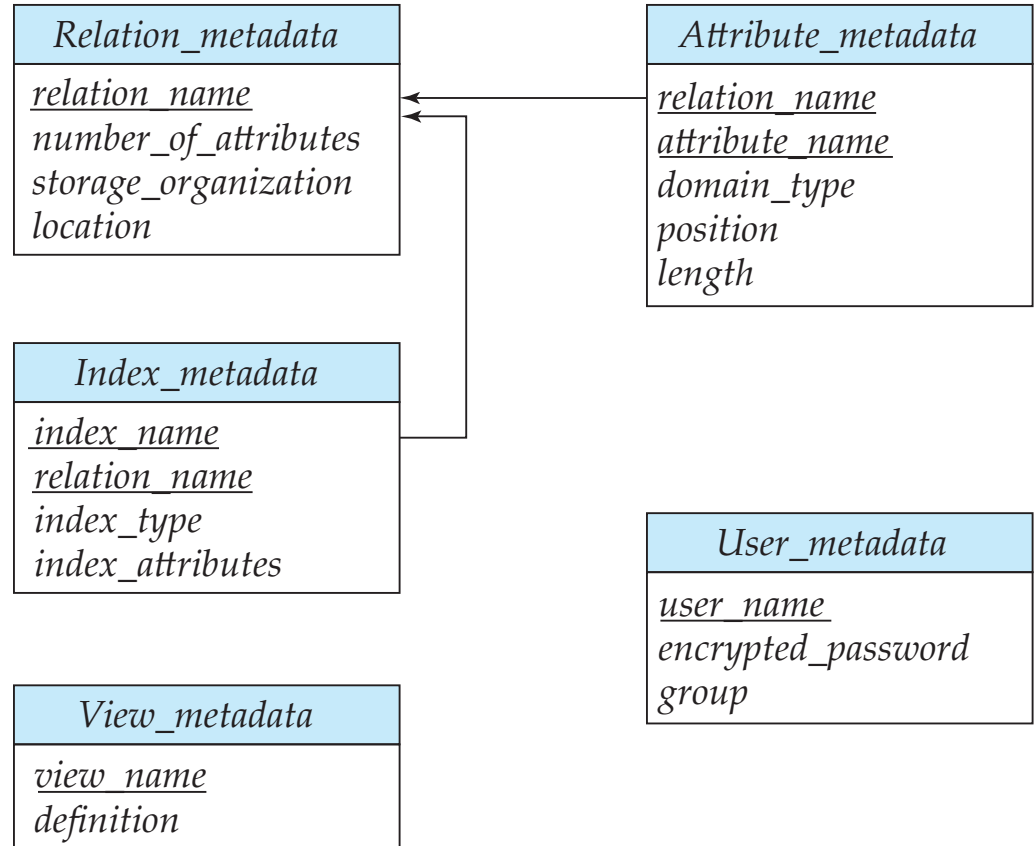
Data Dictionary Storage

The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- Information about relations
 - names of relations
 - names, types and lengths of attributes of each relation
 - names and definitions of views
 - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
 - number of tuples in each relation
- Physical file organization information
 - How relation is stored (sequential/hash/...)
 - Physical location of relation
- Information about indices (Chapter 14)

Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



Column-Oriented Storage

- Also known as **columnar representation**
- Store each attribute of a relation separately
- Example

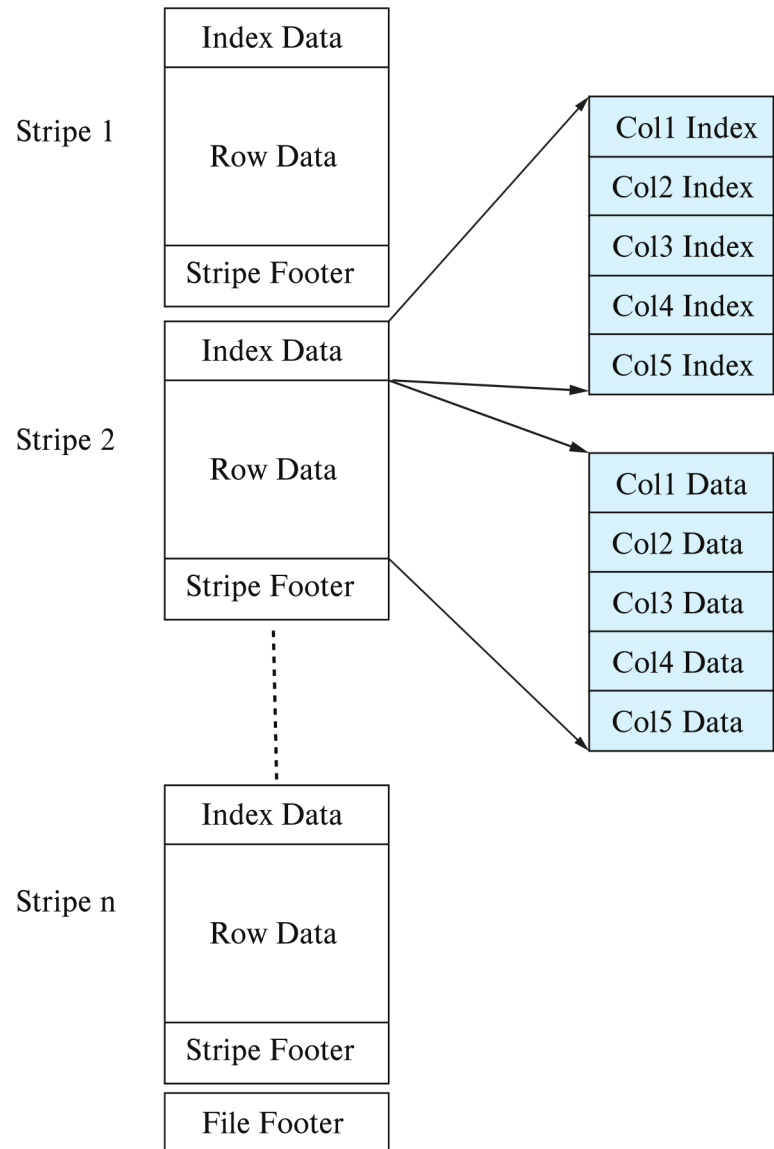
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Columnar Representation

- Benefits:
 - Reduced IO if only some attributes are accessed
 - Improved CPU cache performance
 - Improved compression
 - **Vector processing** on modern CPU architectures
- Drawbacks
 - Cost of tuple reconstruction from columnar representation
 - Cost of tuple deletion and update
 - Cost of decompression
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
 - Called **hybrid row/column stores**

Columnar File Representation

- ORC and Parquet: file formats with columnar storage inside file
- Very popular for big-data applications
- Orc file format shown on right:



Storage Organization in Main-Memory Databases

- Can store records directly in memory without a buffer manager
- Column-oriented storage can be used in-memory for decision support applications
 - Compression reduces memory requirement

