Chapter 13: Data Storage Structures

Database System Concepts, 7^{th} Ed.

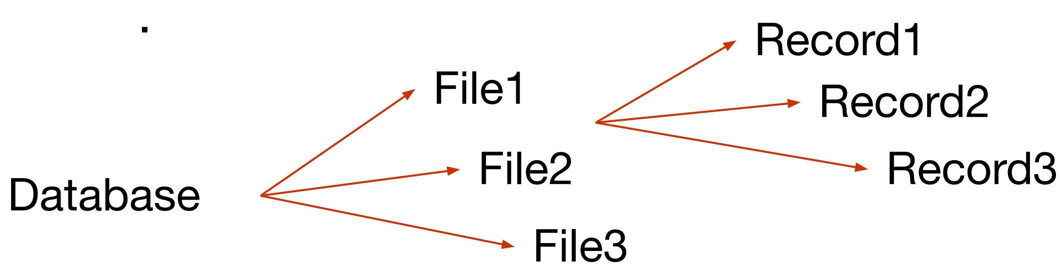
File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
- One approach
  - Assume record size is fixed
  - Each file has records of one particular type only
  - Different files are used for different relations

  This case is easiest to implement; will consider variable length records later

- **We assume that records are smaller than a disk block => In a disk block, there can be multiple records**

.



Sector->track

page->disk block -> records->Files-> DB

Fixed-Length Records

- Simple approach:
  - Store record *i* starting from byte *n* * (*i* – 1), where *n* is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries

Block shifting would be costly.

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

Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - **option 1: move records *i* + 1, . . ., *n* to *i, . . . , n* – 1**
  - **option 2:** move record *n* to *i*
  - **option 3:** do not move records, but link all free records on a *free list*

Record 3 deleted

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

- Deletion of record *i:* alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - **move record *n* to *i***
  - do not move records, but link all free records on a *free list*

Record 3 deleted and replaced by record 11

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

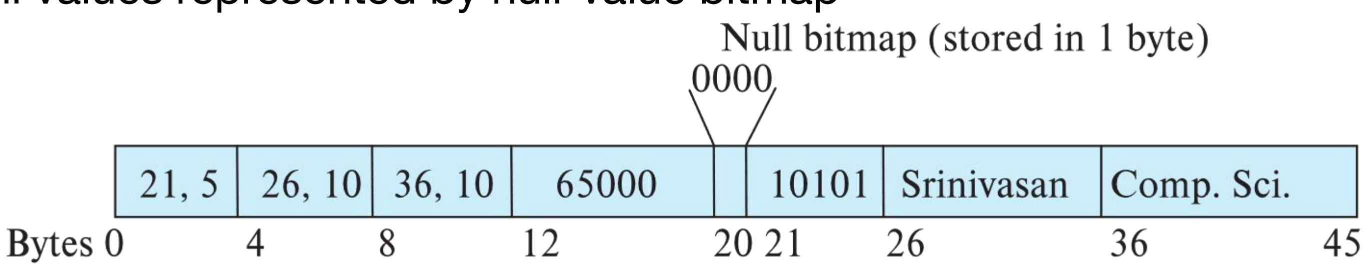
Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record n to i
  - **do not move records, but link all free records on a *free list***

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

Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- ***Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes***
- Null values represented by null-value bitmap



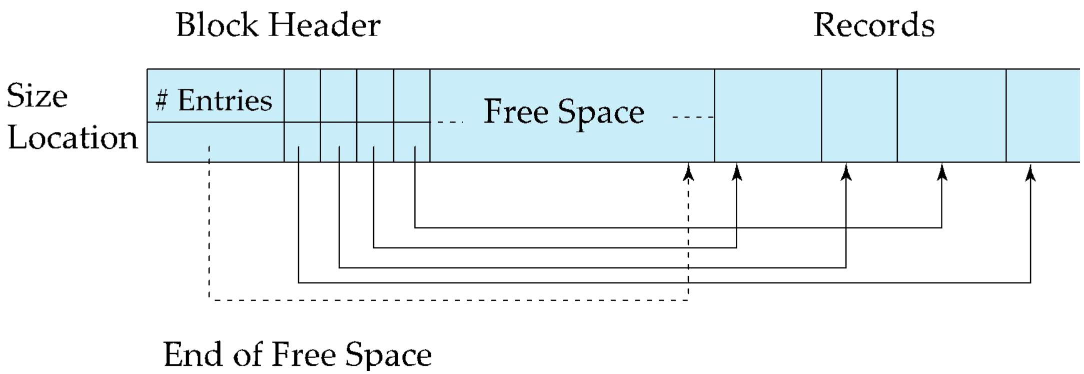
Example of Repeating Fields

Student ID	Name	Courses
101	Alice	Algorithm, Data structure

Same field, multiple, elements

Student ID	Name	Courses
101	Alice	Algorithm
101	Alice	Data structure

Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record

Slotted Page Structure Format

- **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.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.

Storing Large Objects

BLOB = Binary large object CLOB = Character large object

- E.g., blob/clob types
- **Constraint: 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

```
CREATE TABLE Files (
    id INT PRIMARY KEY,
    filename VARCHAR(255),
    filepath VARCHAR(512) --
Stores file location
);
```

Files are stored in **special tables** (or BLOB/CLOB fields) managed by the database.
The database **controls file storage**, including access permissions, indexing, and backups.

```
CREATE TABLE LargeObjects (
    object_id SERIAL PRIMARY KEY,
    object_name VARCHAR(255)
);

CREATE TABLE ObjectChunks (
    chunk_id SERIAL PRIMARY KEY,
    object_id INT REFERENCES LargeObjects(object_id),
    chunk_number INT, -- Order of the chunk
    data_chunk BYTEA -- Stores part of the BLOB
);
```

Organization of Records in Files

- **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
- 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
- **B$^+$-tree file organization**
  - **Ordered storage** even with **inserts/deletes**
  - More on this in Chapter 14
- **Hashing** – a **hash function** computed on search key; the result specifies in **which block of the file** the record should be placed
  - More on this in Chapter 14

Heap File Organization

- **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, e.g: 4/8 => 50%, 2/8 => 25% empty for the ith block.

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/groups of first-level free-space map

4	7	2	6

For each block of 4, the biggest one

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

Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a **search-key**

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 Organization (Cont.)

- 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	

32222	Verdi	Music	48000	

Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization

department

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

instructor

ID	*name*	*dept_name*	*salary*
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 (cont.)

- good for queries involving *department* $\bowtie$ *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 in 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

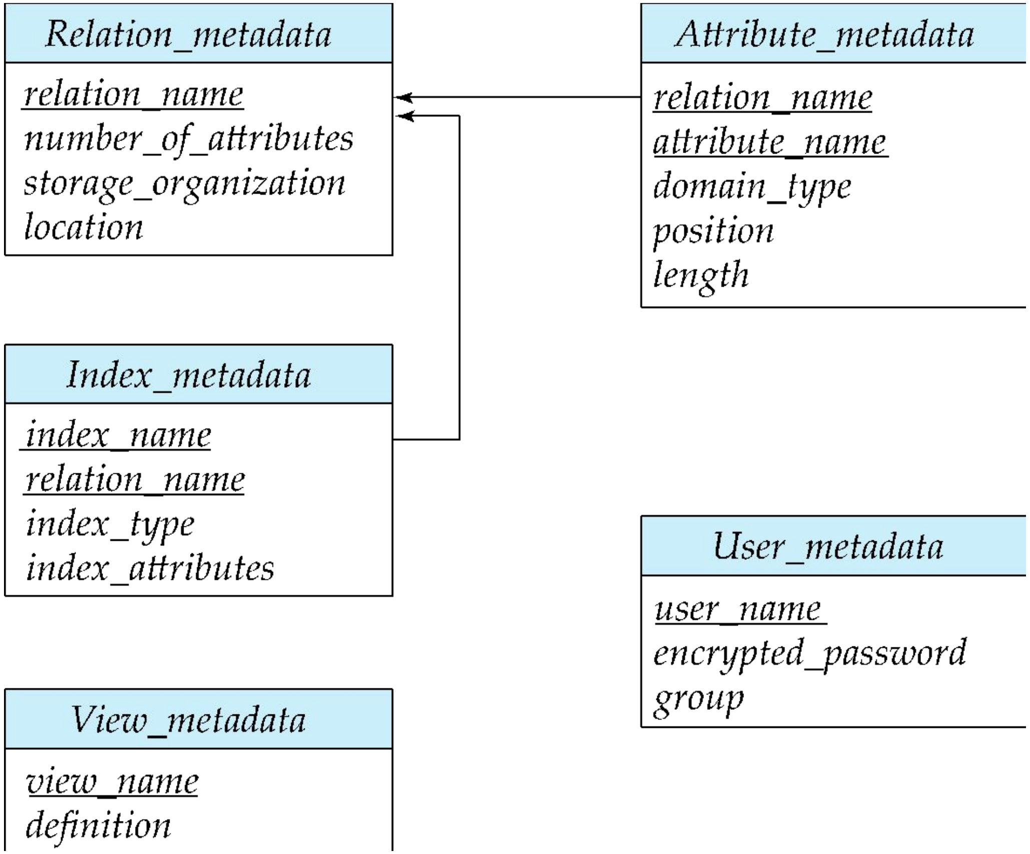
Data Dictionary Storage

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)

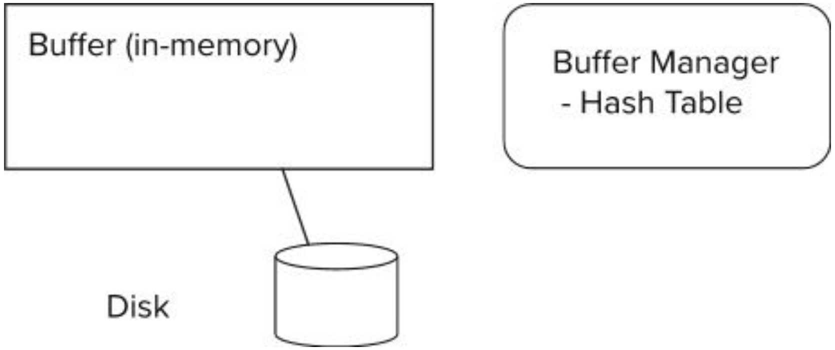
Relational Representation of System Metadata

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



Storage Access

- **Blocks are units of both storage allocation and data transfer**.
- Database system **seeks to minimize the number of block transfers** between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



Buffer Manager

- **Programs call on the buffer manager when they need a block from disk.**
  - If the block is already in the buffer, buffer manager returns the address of the block in main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer for the block
      - Replacing (throwing out) some other block, if required, to make space for the new block.
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
    - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

Buffer Manager

- **Buffer replacement strategy** (details coming up!)
- **Pinned block:** memory block that is not allowed to be written back to disk
  - **Pin** done before reading/writing data from a block
  - **Unpin** done when read /write is complete
  - Multiple concurrent pin/unpin operations possible
    - Keep a pin count, buffer block can be evicted only if pin count = 0
- **Shared and exclusive locks on buffer**
  - Needed to prevent concurrent operations from reading page contents as they are moved/reorganized, and to ensure only one move/reorganize at a time
  - Readers get shared lock, updates to a block require exclusive lock
  - **Locking rules:**
    - Only one process can get exclusive lock at a time
    - Shared lock cannot be concurrently with exclusive lock
    - Multiple processes may be given shared lock concurrently

Buffer-Replacement Policies

- [Strategy-I] Most operating systems replace the block **least recently used** (LRU strategy) [Algorithm/Policy]
  - Idea behind LRU – use past pattern of block references as a **predictor** of future references
  - LRU can be bad for some queries
- Queries have **well-defined access patterns** (such as sequential scans), and **a database system can use the information in a user's query to predict future references [prediction preferences/pattern]**
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- Example of bad access pattern for LRU: when computing the join of 2 relations r and s by a nested loops

for each tuple *tr* of *r* do
  for each tuple *ts* of *s* do
    if the tuples *tr* and *ts* match …

- Thrashing and fetching s blocks of data repeatedly in successive iterations

Buffer-Replacement Policies (Cont.)

- [Strategy-II] **Toss-immediate** strategy – frees the space occupied by a block **as soon as the final tuple of that block** has been processed *[Automatic Erase]*
- [Strategy-II] **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block. [***Detected, but if required then erased, not auto***]
- Buffer manager **can use statistical information regarding the probability** that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Operating system or buffer manager **may reorder writes**
  - **Can lead to corruption of data structures** on disk
    - E.g., linked list of blocks with missing block on disk [B1->B2->B3, missed ordering problem]
    - File systems **perform consistency check** to detect such situations
  - Careful ordering of writes can avoid many such problems

Optimization of Disk Block Access (Cont.) (Reducing the overall access time)

- Buffer managers support **forced output** of blocks for the purpose of recovery (more in Chapter 19)
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM or flash buffer immediately
  - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – *a disk devoted to writing a sequential log of block updates*
  - Used exactly like nonvolatile RAM
    - Write to log disk is very fast since no seeks are required
- **Journaling file systems** write data in-order to NV-RAM or log disk [*Tracking timestamps for easy tracking and reverting in case of incomplete writing*]
  - Reordering without journaling: risk of corruption of file system data

Column-Oriented Storage

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

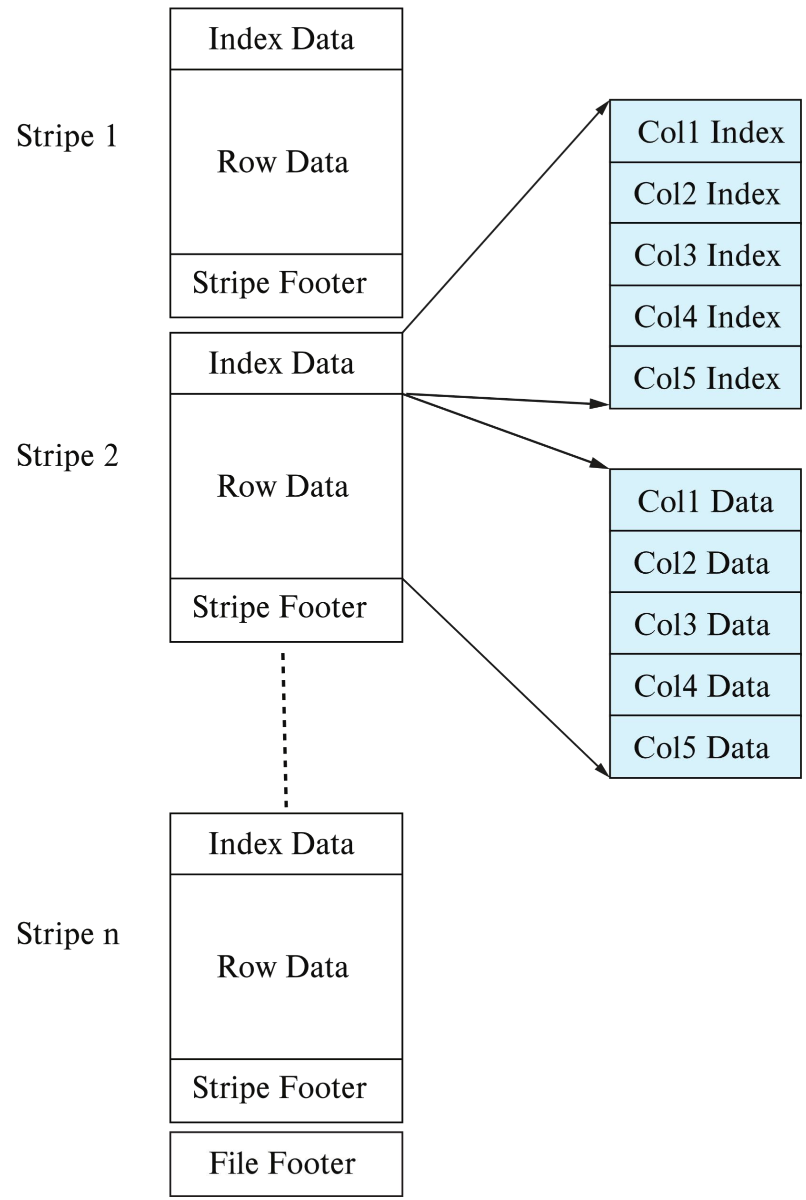
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
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 (.orc) file format shown on right:



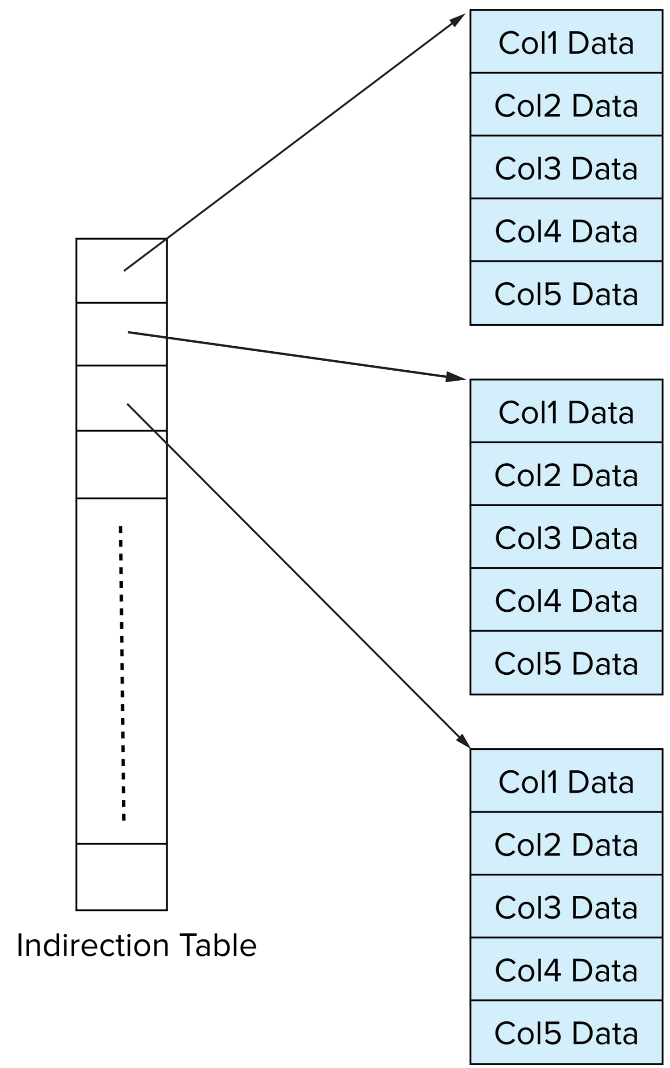
[index data - for easier faster access, filtering, etc.]

[Row data - actual data]

[Stripe Footer - Metadata]

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



End of Chapter 13