Chapter 13: Data Storage Structures

Database System Concepts, 7th Ed.

Adapted by Carlos Viegas Damásio

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 only one particular type
  - 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 (database) block
  - This guarantees efficient atomic writes of blocks to guarantee consistency of information upon failures
- We assume that each record is entirely contained in one block, i.e. it is not split into several blocks (two if its size if smaller than a database block)
  - This guarantees efficient access to data

Example

- Suppose you have declared a table

```
CREATE TABLE Instructor(
    ID varchar(5);
    name varchar(20);
    dept_name varchar(20);
    salary numeric(8,2);
)
```

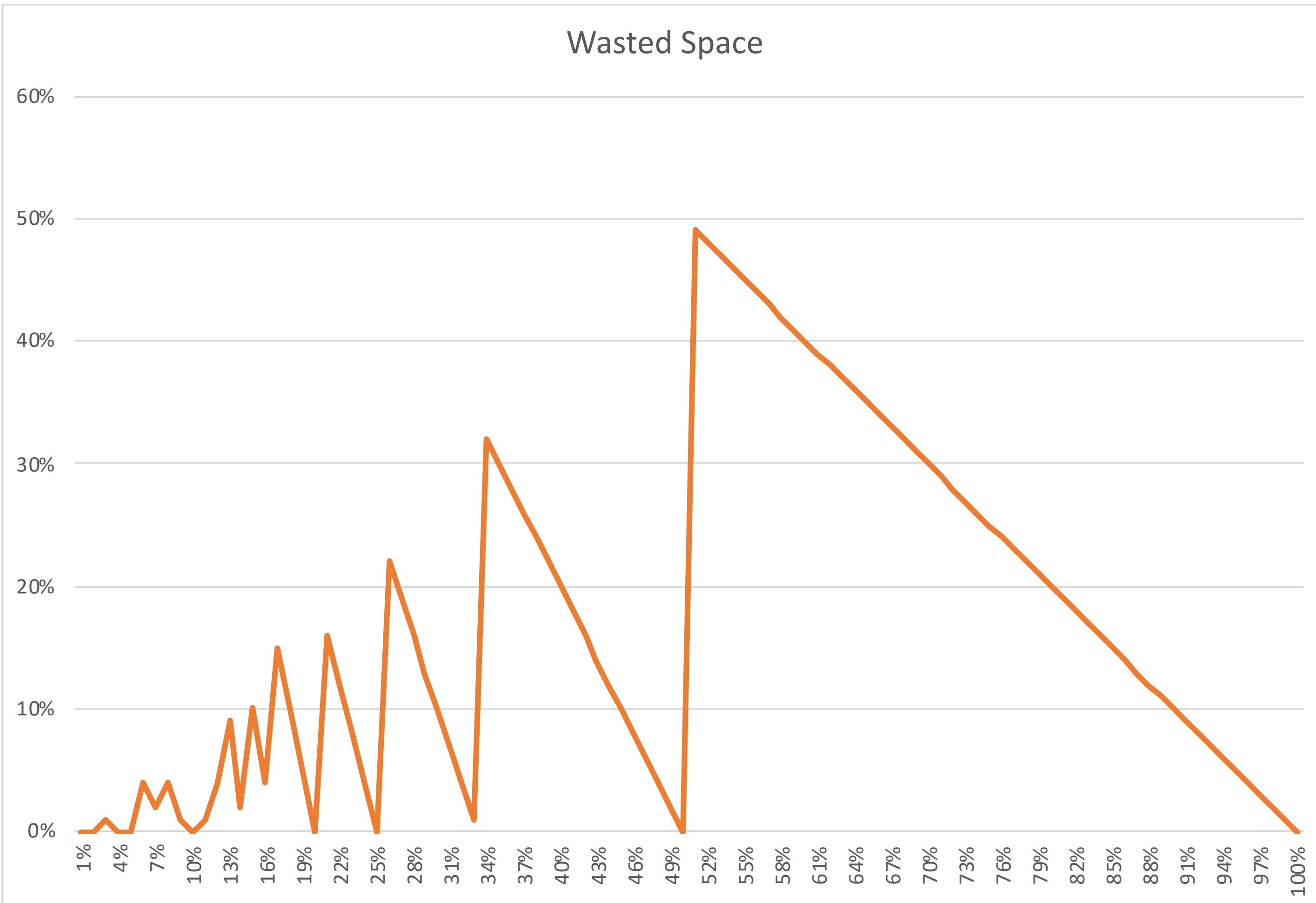
- A simple way to store the data is to used fixed-length records of size 5+20+20+8 = 53 bytes, assuming that strings are represented with ASCII characters (it can be a lot longer if other encoding is used, like Unicode)

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
    - Results in wasted space depending on the record-length

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

Wasted space versus record-length



Record-length / Block size

Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - **move records *i* + 1, . . ., *n* to *i*, . . . , *n* – 1**
  - move record *n* to *i*
  - 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, . . ., *n* to *i, . . .* , *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

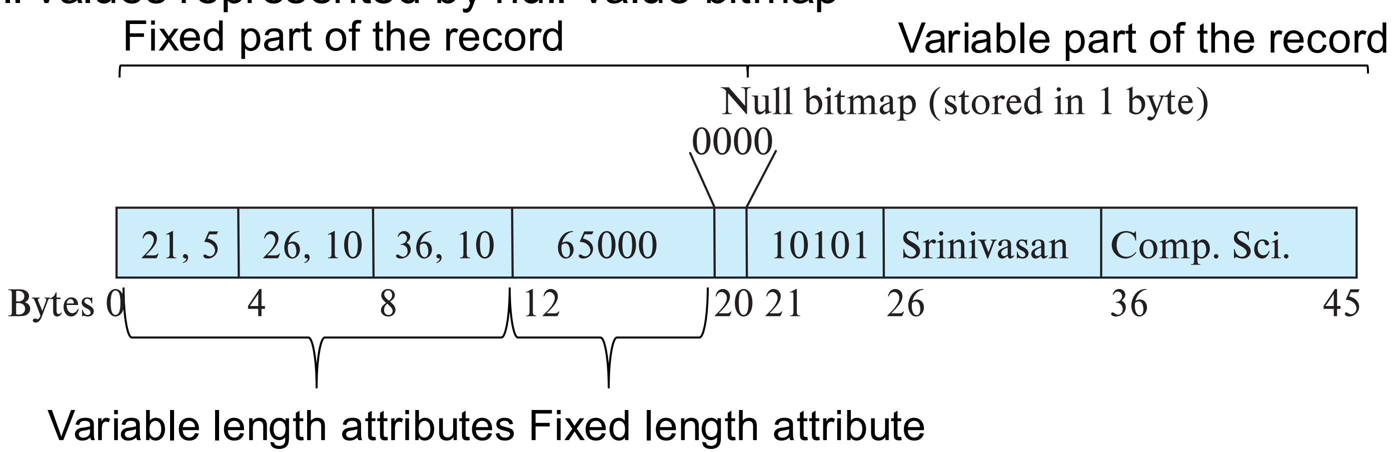
Fixed-Length Records

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

header	first			free
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



Variable-Length Records

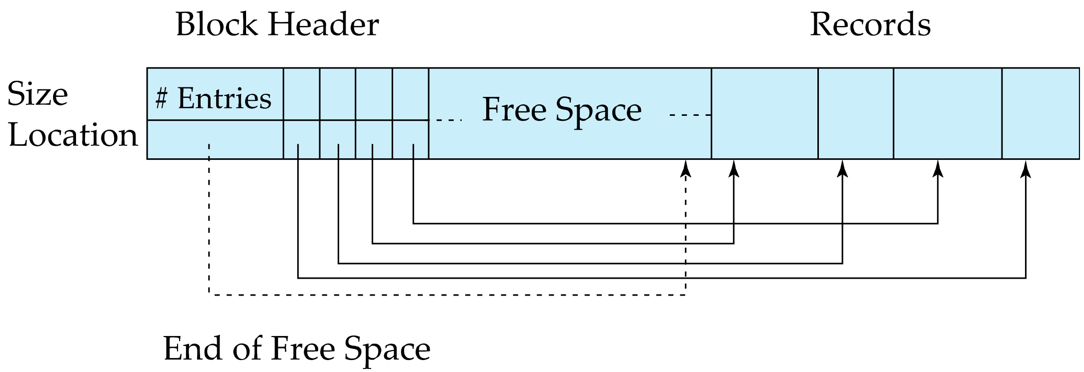
- If there are usually a lot of null values an alternative record organization can be used by putting the null map at the beginning and not storing (offset,length) values for null values.
- This reduces the space used at expenses of more complex retrieval of attributes from the record
- Suppose we have the following instructor record to store

  (678,'John Freelancer',NULL,47000)

	&ID	&name	salary		
0010	17,5	20,15	47000	678	John Freelancer

0 1 5 9 17 20 34

Slotted Page Structure



- **Slotted pages** store SEVERAL variable-length records
- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- 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

- E.g., blob/clob types
- Records must be smaller than pages
- Alternatives:
  - Store as files in file systems
  - Store as files managed by database (e.g. B^+-tree file organization)
  - Break into pieces and store in multiple tuples in separate relation
    - PostgreSQL TOAST (The Oversized-Attribute Storage Technique) (1Gb max, < 127 bytes are stored in the record)
- Storing files in the database may result in very large and long backups
  - Storing however in the file system may result in “broken links”
  - No control to access the files
  - Oracle’s SecureFiles avoids these problems

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 next week (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 next week (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

- **Free-space map** is used to estimate the free space available in a block
  - 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 (rounding down since there is always some occupied space) 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)
- See for instance PostgreSQL description

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

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
- The pointer space can be reused for the free-list

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 using cluster key dept_name*

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
- Can reduce space by not replicating the cluster key

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 SELECT * FROM *year*=2019, in which case only one partition in needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Queries are rewritten to only read the necessary partitions
  - 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

File System

- In sequential file organisation, each relation (or partition of a relation) is stored in a file
  - One may rely on the file system of the underlying operating system
- Multitable clustering may have significant gains in efficiency
  - But this may not be compatible with the file system of the operating system
- Several large scale database management systems do not rely directly on the underlying operating system
  - The relations are all stored in a single (multitable) file
  - The database management system manages the file by itself
  - This requires the implementation of an own file system inside the DBMS

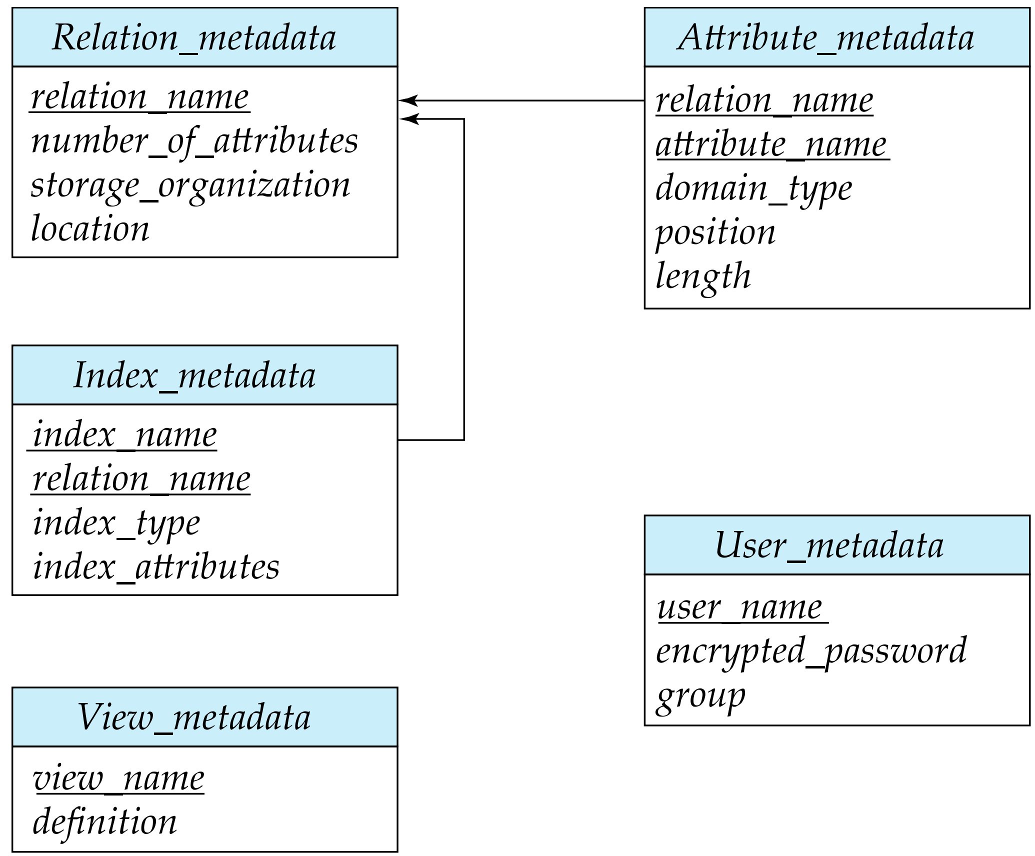
Data Dictionary Storage

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)

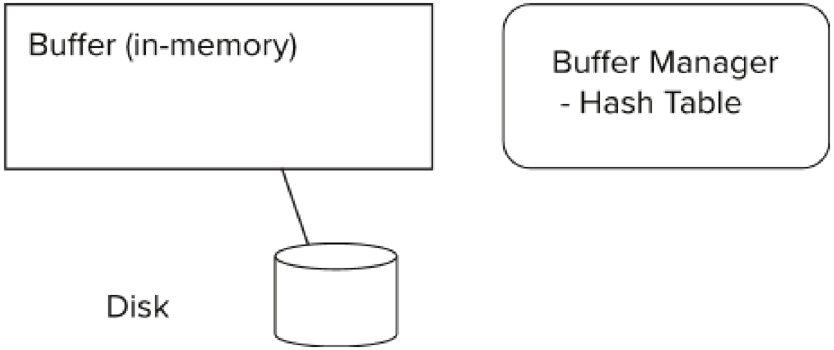
Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory
- Usually, DBMS load all this information into memory at database start



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
      - Evicting (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, atomically updated, 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

Locking policy

- Before carrying out any operation on a block, a process must pin the block as we saw earlier. Locks are obtained subsequently and must be released before unpinning the block.
- Before reading data from a buffer block, a process must get a shared lock on the block. When it is done reading the data, the process must release the lock.
- Before updating the contents of a buffer block, a process must get an exclusive lock on the block; the lock must be released after the update is complete.
- Extra steps will be needed to protect the database due to concurrent access (later on the course...)
  - Cannot update the block while writing it to disk
  - Forcing output of blocks in particular circumstances

Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
  - 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
- 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 …

Buffer-Replacement Policies (Cont.)

- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **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.
- 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
    - File systems perform consistency check to detect such situations
  - Careful ordering of writes can avoid many such problems

Optimization of Disk Block Access (Cont.)

- 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
  - Reordering without journaling: risk of corruption of file system data
  - See https://en.wikipedia.org/wiki/Journaling_file_system

File Organization in Oracle

- Oracle has its own buffer management, with complex policies
  - Oracle doesn't rely on the underlying operating system's file system
- A database in Oracle consists of **tablespaces**:
  - System tablespace: contains catalog metadata
  - User data tablespaces
- The space in a tablespace is divided into **segments** with different purposes
  - Data segment
  - Index segment
  - Temporary segment (for sort operations)
  - Rollback segment (for processing transactions)
- Segments are divided into **extents**, each extent being a set of contiguous **database blocks**.
  - Database blocks need not be the same size of operating system blocks, but are always a multiple

File Organization in Oracle

- A standard table is organised in a heap (no sequence is imposed)
- Partitioning of tables is possible for optimisation
  - Range partitioning (e.g by dates)
  - Hash partitioning
  - Composite partitioning
- Table data in Oracle can also be (multitable) clustered (with **create cluster**)
  - One may tune the clusters to significantly improve the efficiency of query to frequently used joins.
- Hash file organisation (to be studied later) is also possible for fetching the appropriate cluster

- A database can be tuned by an appropriate choice for the organisation of data:
  - Choosing partitions
  - Appropriate choice of clusters
  - Hash or sequential
- Tuning makes the difference in big (real) databases!

Column-Oriented Storage

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

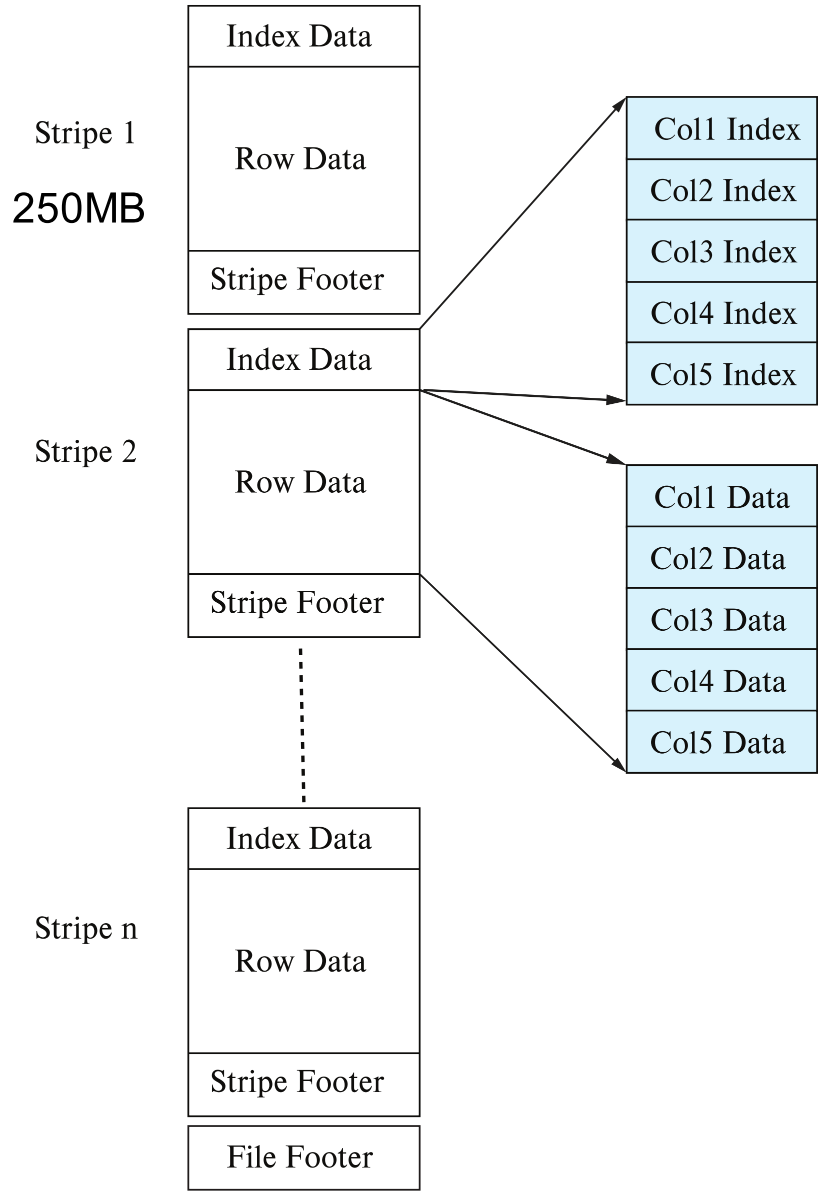
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
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**
- Some row stores use columnar representation inside blocks

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:



Some benchmarks (F. Barreiras)

Fuel Break	Original Format (CSV)	CSV-GZIP	CSV-ZIP	Parquet	ORC
Almada	1.5 Gb	393 Mb	401 Mb	375 Mb	924 Mb
Mação	8.5 Gb	2.0 Gb	2.0 Gb	1.4 Gb	5 Gb
Santarém	23 Gb	5.6 Gb	5.4 Gb	4.3 Gb	15 Gb

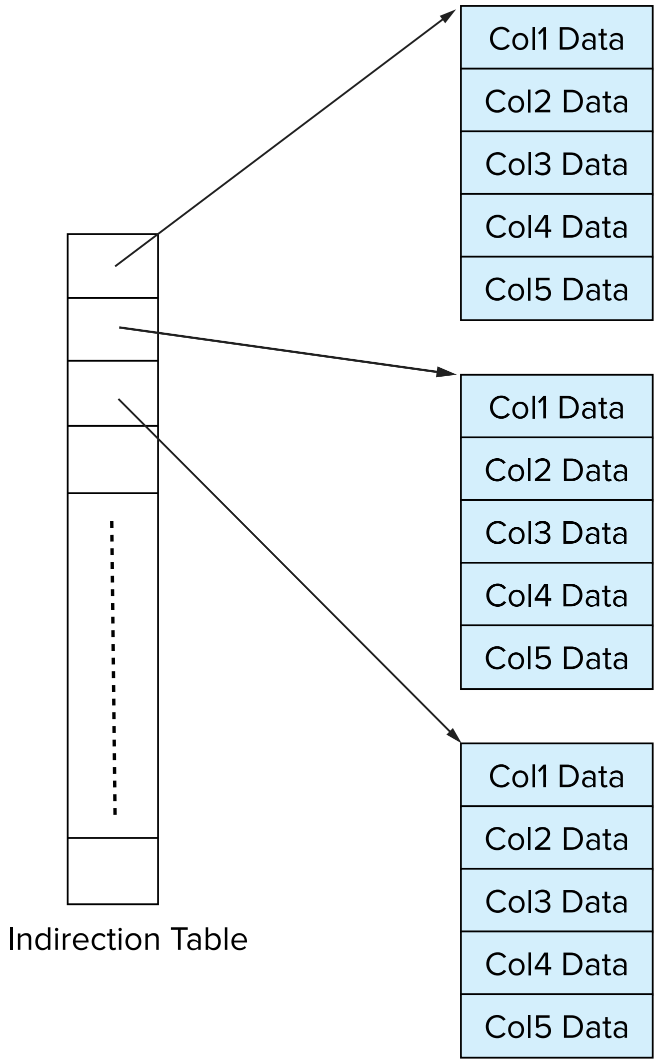
Table 8.1: Storage metrics between tabular data formats.

Fuel Break	Original Format (CSV)	CSV-GZIP	CSV-ZIP	Parquet	ORC
Almada	12m 38s	14m 9s	14m	11m 19s	11m 10s
Mação	32m 20s	38m 25s	34m 45s	27m 50s	28m 20s
Santarém	1h 22m	1h 39m	1h 29m	1h 10m	1h 11m

Table 8.2: Performance metrics between tabular data formats.

Storage Organization in Main-Memory Databases

- Can store records directly in memory without a buffer manager
- Pointer redirection avoided by not allowing records to move, thus slotted pages may not be used and requires extra work to avoid memory fragmentation
- Column-oriented storage can be used in-memory for decision support applications
  - Allocates contiguous regions of memory
  - Compression reduces memory requirement
- Indirection tables can be used to avoid reallocation when new data is appended (see figure)



End of Chapter 13