Lecture #03: Database Storage (Part I)

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Storage

In this class, we focus on a "disk-oriented" DBMS architecture that assumes that the primary storage location of the database is on non-volatile disk(s). 非易失

At the top of the storage hierarchy, you have the devices that are closest to the CPU. This is the fastest storage, but it is also the smallest and most expensive. The further you get away from the CPU, the larger but slower the storage devices get. These devices also get cheaper per GB.

There's also a demarcation line in the middle of the hierarchy that separates volatile devices from non-volatile devices.

Volatile Devices

- Volatile means that the device does not retain its state after power loss. Therefore, the data that is stored in such devices can be lost.
- Volatile storage supports fast random access with byte-addressable locations. This means that the program can jump to any byte address and get the data that is there (e.g. in DRAM).
- For our purposes, we will always refer to this volatile storage class as "memory."

Non-Volatile Devices

- Non-volatile devices do retain their state even when the machine/computer is off or power loss occurs. Therefore, the data that these devices store can be retrieved even after the machine/computer shuts down and restarts (e.g. disk).
- Non-Volatile devices are block/page addressable. This means that in order to read a value at a particular offset (byte), the program first has to load the 4 KB page into memory that holds the value that the program wants to read.
- Non-volatile storage is traditionally better at sequential access (reading contiguous blocks of data because of its architecture e.g. magnetic hard drive).
- We will refer to this as "disk." We will not make a (major) distinction between solid-state storage (SSD) and spinning hard drives (HDD).

2 Access Time and Access Pattern

There is a large contrast between latencies accessing volatile vs. non-volatile devices. In order to better understand the orders of magnitute of latency difference, (suppose that reading data from the L1 cache reference took one second, then reading from an SSD would take 4.4 hours, and reading from an HDD would take 3.3 weeks.)

There are two major access patterns, *random access* and *sequential access*. On real-world hardwares the differences between their access latencies are significant. Random access on non-volatile storage is almost always slower than sequential access. The DBMS will always target maximizing sequential access, some

systems will avoid blocking on random writes by writing sequentially into a buffer and later perform random disk writes in the background.

3 System Design Goals

Virtual Memory

A design goal of the disk-oriented DBMS is to allow it to manage databases that exceeds the amount of memory available. As this requires frequent data movement between memory and disk, the DBMS should manage disk read and write carefully to avoid long stalls on disk I/O and maximize sequential access when possible.

4 Disk-Oriented DBMS Overview

The database is stored on disk, and the data within the database files are organized into pages, with the first page being the directory page. To operate on the data, the DBMS needs to bring the data into memory. It does this by having a *buffer pool* that manages the data movement back and forth between disk and memory. The DBMS also has an execution engine that will execute queries. The execution engine will ask the buffer pool for a specific page, and the buffer pool will take care of bringing that page into memory and giving the execution engine a pointer to that page in memory. The buffer pool manager will ensure that the page is there while the execution engine operates on that part of memory.

5 DBMS vs. OS

A high-level design goal of the DBMS is to support databases that exceed the amount of available memory. Since reading/writing to disk is expensive, disk use must be carefully managed. We do not want large stalls from fetching something from disk to slow down everything else. We want the DBMS to be able to process other queries while it is waiting to get the data from disk.

This high-level design goal is like virtual memory, where there is a large address space and a place for the OS to bring in pages from disk.

One way to achieve this virtual memory is by using `mmap` to map the contents of a file in a process' address space, which makes the OS responsible for moving pages back and forth between disk and memory. Unfortunately, this means that if `mmap` hits a page fault, the process will be blocked.

- You never want to use `mmap` in your DBMS if you need to write.
- The DBMS (almost) always wants to control things itself and can do a better job at it since it knows more about the data being accessed and the queries being processed.
- The operating system is not your friend.

It is possible to use the OS by using:

- `madvise`: Tells the OS know when you are planning on reading certain pages.
- `mlock`: Tells the OS to not swap memory ranges out to disk.
- `msync`: Tells the OS to flush memory ranges out to disk.

We do not advise using `mmap` in a DBMS for correctness and performance reasons.

Even though the system will have functionalities that seem like something the OS can provide, having the DBMS implement these procedures itself gives it better control and performance.

How the DBMS represents the database in files on disk?

6 File Storage

In its most basic form, a DBMS stores a database as files on disk. Some may use a file hierarchy, others may use a single file (e.g. SQLite).

The DBMS traditionally store files in a proprietary format that are specific to the DBMS, therefore only the specific DBMS knows how to decipher their contents. More recently there are also portable file formats that are open specs which allow all DBMSs to read and write from them. For either approach, the OS does not know anything about the contents of these files.

The DBMS typically runs on off-the-shelf file system provided by the OS. There are some high-end enterprise systems that could inject a custom file system into the OS specific for the DBMS (e.g. Oracle ASM)

The DBMS's *storage manager* is responsible for managing a database's files. It represents the files as a collection of pages. It also keeps track of what data has been read and written to pages as well how much free space there is in these pages.

Replication is not handled on storage manager level. Typically they are handled either in the file system below the storage manager (e.g. RAID), or above the storage manager where there could be logical copies of tuples.

7 Database Pages

The DBMS organizes the database across one or more files in fixed-size blocks of data called *pages*. Pages can contain different kinds of data (tuples, indexes, etc). Most systems will not mix these types within pages. Some systems will require that pages are *self-contained*, meaning that all the information needed to read each page is on the page itself.

[Virtual Page Number | offset → Page Frame Number | offset]

Each page is given a unique identifier (page ID). If the database is a single file, then the page id can just be the file offset. A page ID could be unique per DBMS instance, per database, or per table. Most DBMSs have an indirection layer that maps a page id to a file path and offset. The upper levels of the system will ask for a specific page number. Then, the storage manager will have to turn that page number into a file and an offset to find the page.

Most DBMSs use fixed-size pages to avoid the engineering overhead needed to support variable-sized pages. For example, with variable-size pages, deleting a page could create a hole in files that the DBMS cannot easily fill with new pages.

There are three concepts of pages in DBMS:

1. Hardware page (usually 4 KB).
2. OS page (4 KB).
3. Database page (1-16 KB).

Optimal database page size depends on the environment, database contents, and expected workload. DBMSs that specialize in read-only workloads tend to have larger page sizes (¿= 1MB), while those that specialize in write-heavy workloads tent to have smaller pages (4-16KB).

The storage device guarantees an atomic write of the size of the hardware page. If the hardware page is 4 KB and the system tries to write 4 KB to the disk, either all 4 KB will be written, or none of it will. This means that if our database page is larger than our hardware page, the DBMS will have to take extra measures to ensure that the data gets written out safely since the program can get partway through writing a database page to disk when the system crashes.

8 Database Heap

Index Sequential Access Method

There are a couple of ways to manage pages in files on the disk (e.g. {Tree, ISAM, Hashing} File Organization), and *Heap File Organization* is one of those ways. A *heap file* is an unordered collection of pages where tuples are stored in random order.

A common architecture for the DBMS to locate a page on disk given a `page_id` is *page directory*, which are special pages that contain one location entry for each logical database objects (e.g. data pages, index pages). The page directory has to be synchronized with the actual pages. The DBMS also tracks metadata about pages' contents, include the amount of free space on each page, a list of free/empty pages, and the page types.

9 Page Layout

Lec 5

Every page includes a header that records meta-data about the page's contents:

- Page size.
- Checksum. 校验和
- DBMS version.
- Transaction visibility.
- Self-containment. (Some systems like Oracle require this.)

There are three main approaches to laying out data in pages: (1) tuple-oriented, (2) log-structured, and (3) index-oriented.

① Tuple-Oriented

In tuple-oriented storage, the entire tuple is stored in the page. A strawman approach to laying out tuples in a page is to keep track of how many tuples the DBMS has stored in the page and then append to the end every time a new tuple is added. However, problems arise when tuples are deleted or when tuples have variable-length attributes. A common layout scheme that solves this is *slotted pages*.

Slotted Pages: Page maps slots to offsets.

- Most common approach used in DBMSs today.
- Header keeps track of the number of used slots, the offset of the starting location of the last used slot, and a slot array, which keeps track of the location of the start of each tuple.
- To add a tuple, the slot array will grow from the beginning to the end, and the data of the tuples will grow from end to the beginning. The page is considered full when the slot array and the tuple data meet.

② Log-structured and ③ index-oriented storage are covered in lecture 05.

10 Record IDs

The DBMS assigns each logical tuple a unique *record identifier* that represents its physical location in the database (e.g. file id, page id, slot number). Most DBMSs do not store ids in the tuple. Common record id size varies from 4 bytes to 10 bytes. Since these are physical locations within the DBMS, the application **cannot** rely on these IDs.

11 Tuple Layout

A tuple is essentially a sequence of bytes (these bytes do not have to be contiguous). It is the DBMS's job to interpret those bytes into attribute types and values.

Tuple Header: Contains meta-data about the tuple.

- Visibility information for the DBMS's concurrency control protocol (i.e., information about which transaction created/modified that tuple, will be covered later in the semester).
- Bit Map for `NULL` values.
- Note that the DBMS does not need to store meta-data about the schema of the database here.

Tuple Data: Actual data for attributes.

- Attributes are typically stored in the order that you specify them when you create the table.
- Attributes must be word aligned.
- Most DBMSs do not allow a tuple to exceed the size of a page.

12 Data Representation

Data representation defines the attribute storage format. Storage formats for common data types are described below.

INTEGER / BIGINT / SMALLINT / TINYINT

Storage layout follows native format in C/C++.

FLOAT / REAL vs. NUMERIC / DECIMAL

Storage layout follows IEEE-754 Standard or Fixed-point Decimals.

- **Variable precision numbers**
  Are "native" C/C++ types stored as specified by IEEE-754. They are **inexact** but typically **faster** than fixed precision numbers due to CPU ISA's instruction and register support.
- **Fixed precision numbers**
  Allows arbitrary precision and scale. One of many possible implementations is storing in exact, variable-length binary.

VARCHAR / VARBINARY / TEXT / BLOB

Stored as header with length with data bytes **or** pointer to data page and offset.

- **Overflow Pages**
  When the value cannot fit in a single page, separate *overflow pages* are used and record id to the tuple is stored inline. Optimizations are possible to compress the overflow page; or to store the prefix of the value inline to reduce indirection when scanning.
- **External Value Storage**
  Some system allow large value to be stored in external file and treated as *BLOB* type. DBMS cannot manipulate the content of external file, therefore has no durability and transaction guarantees on it.

TIME / DATE / TIMESTAMP / INTERVAL

Stored as 32/64-bit integer of micro or milli-seconds since Unix epoch.

Null Data Types

There are several approaches to storing null data types.

1. **Null Column Bitmap Header**
   Store bitmap in centralized header, bit is set when the corresponding attribute is NULL. Most common approach in row-stores.
2. **Special Values**
   Use a special placeholder for NULL for a data type (e.g. INT32_MIN). Most common in column-stores.
3. **Per-Attribute Null Flag**
   Stores a flag per-attribute that marks a value is null. Undesirable because the extra bit per-attribute would need to be padded for alignment. Don't do this !

Lecture #05: Database Storage (Part II)

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Buffer Pool Optimizations

There are several ways to optimize a buffer pool to tailor it to the application's workload.

Multiple Buffer Pools (2)

The DBMS can maintain multiple buffer pools for different purposes (i.e. per-database, per-table, per-page type, etc.). Then, each buffer pool can adopt local policies tailored to the data stored inside of it. This method can help reduce latch contention and improve locality.

Object IDs and hashing are two approaches to mapping desired pages to a specific buffer pool. **Object IDs** involve extending the record IDs to have an object identifier. A mapping from objects to specific buffer pools can be maintained via these object IDs. This allows a finer-grained control over buffer pool allocations but has a storage overhead. Another approach is **hashing**, where the DBMS hashes the page ID to select which buffer pool to access. This is a more general and uniform approach.

Pre-fetching (2)

The DBMS can also be optimized by pre-fetching pages based on the query plan. While the first set of pages is being processed, the second can be pre-fetched into the buffer pool. This method is commonly used by DBMSs when accessing many pages sequentially during a sequential scan. It is also possible for a buffer pool manager to prefetch leaf pages in a tree index data structure benefiting index scans. Note that this does not necessarily need to be the next physical page on disk, but instead the next logical page in the leaf scan.

e.g. B+Tree

Scan Sharing (Synchronized Scans)

Query cursors can reuse data retrieved from storage or operator computations. This allows multiple queries to attach to a single cursor that scans a table. If a query starts a scan and there is another active scan, then the DBMS will attach the second query's cursor to the existing cursor. The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure. This satisfies correctness since the order of scans is not guaranteed by a DBMS and is often useful when a table is frequently scanned.

Note: Continuous scan sharing is an academic prototype that constantly runs a sequential scan for certain tables and queries can hop on and off. However, this is very wasteful and costly when paying per I/O.

Overall, the DBMS can almost always manage memory better than the OS. It can leverage the semantics about the query plan to make better decisions:

- Evictions
- Allocations
- Pre-fetching

To reiterate, the Operating System is ***not*** your friend.

2 Tuple-Oriented Storage

Read ✓ Insert ✓ Update ?

The most common way to store tuples on disk is the Tuple-Oriented Storage architecture, using the slotted-page scheme described in previous lectures. Tuples are retrieved using its record ID:

- Check the page directory to find the page position on disk.
- Fetch the page from disk into memory (into the buffer pool).
- Use the slot array to find the tuple's offset within the page.

Inserting a new tuple is simple:

- Check the page directory to find a page with a free slot.
- Fetch the page from disk into memory.
- Use the slot array to check if there is enough free space in the page.
- If not, find another page with a free slot or create a new page.
- Insert the tuple into the page and update the slot array.

However, updating tuples can become expensive:

- Navigate to the tuple using the record ID with the same steps as retrieval.
- If the new value fits in the same space, update in place.
- Otherwise, mark the old value as deleted and insert the new value as if it were a new tuple.

Therefore, while tuple-oriented storage is great for reads, there are several problems associated with it:

- **Fragmentation**: Deletion of tuples can leave gaps in the pages, making them not fully utilized.
- **Useless Disk I/O**: Due to the block-oriented nature of non-volatile storage, the whole block needs to be fetched to update a tuple.
- **Random Disk I/O**: The disk reader could have to jump to 20 different places to update 20 different tuples, which can be very slow.

What if we were working on a system which only allows creation of new pages and no in-place updates (e.g. HDFS, Google Colossus, certain object stores)? The log-structured storage model works with this assumption and addresses some of the problems listed above.

3 Log-Structured Storage

Update → Append ✓

Instead of storing tuples in pages and updating them in-place, Log-Structured Storage maintains a log that records changes to tuples. This idea is based on log-structured file systems (LSFS) [1] and log-structured merge trees (LSM Tree)[2].

The DBMS applies changes to an in-memory data structure (**MemTable**) and writes out the changes sequentially to disk (**SSTable**). The records stored in these structures contain the tuple's unique identifier, the type of operation (`PUT`/`DELETE`), and for a `PUT` operation, the contents of the tuple. Effectively, you care about the latest values for each key (most recent `PUT`/`DELETE`).

Logs are first stored in **MemTable** through fast, in-memory operations. Once **MemTable** fills up, the DBMS serializes the logs it stores and writes them to disk as an **SSTable**. The DBMS also sorts each **SSTable** by key before writing it to disk. Since the **SSTable**s are immutable and written to disk sequentially, this results in less random disk I/O. This workload also maps nicely to **append-only** storage like many cloud storage options, etc.

[1] https://doi.org/10.1145/146941.146943
[2] https://doi.org/10.1007/s002360050048

To read a record, the DBMS first checks **MemTable** to see whether it exists there. If the key does not exist in **MemTable**, then the DBMS has to check the **SSTable**s at each level. A brute force solution is to scan down the **SSTable**s from newest to oldest and perform binary search within each **SSTable** to find the most recent contents of the tuple, which can be slow. To avoid this, the DBMS can maintain an in-memory **SummaryTable** to track additional metadata like min/max key per **SSTable** and a key filter (e.g., Bloom filter) per level.

Compaction

In a write-heavy workload, the DBMS will accumulate a large number of **SSTable**s on disk. Thus, the DBMS can periodically use a sort-merge algorithm to combine **SSTable**s by taking only the most recent change for each tuple. This reduces wasted space and speeds up reads.

There are many ways to compact log files. In **Universal Compaction**, **SSTable**s reside in a single "universal" level. DBMS will trigger compaction when size thresholds are met or too many **SSTable**s overlap in key ranges. In **Level Compaction**, the smallest files are level 0. Level 0 files can be compacted to create a bigger level 1 file, level 1 files can be compacted to a level 2 file, etc. **SSTable**s in the same level are managed with sorted and non-overlapping key ranges (except for level 0, which may have overlapping key ranges).

Tradeoffs

The tradeoffs of using Log-Structured Storage are summarized below:

- Fast sequential writes, good for append only storage.
- Reads may be slow.
- Compaction is expensive.
- Write amplification (for each logical write, there could be multiple physical writes during the compaction process).

4 Index-Organized Storage

Read vv

Both slotted-page storage and log-structured storage rely on an additional index to find individual tuples because the tables are inherently unsorted. In the **index-organized storage** scheme, the DBMS directly stores a table's tuples as the value of an index data structure (e.g. B+ tree, skip list, trie). The DBMS uses a page layout similar to a slotted page, and tuples are typically sorted in the page based on key.

Lecture #06: Storage Models & Compression

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Database Workloads

OLTP: Online Transaction Processing *(Simple Query + Write-Heavy)*

An OLTP workload is characterized by fast, short running operations, repetitive operations and simple queries that operate on single entity at a time. OLTP workloads typically handle more writes than reads, and only read/update a small amount of data each time.

An example of an OLTP workload is the Amazon storefront. Users can add things to their cart and make purchases, but the actions only affect their account. high concurrency

OLAP: Online Analytical Processing *(Complex Query + Read-Heavy)*

An OLAP workload is characterized by long running, complex queries (which often involves computing aggregates) and reads on large portions of the database. In OLAP workloads, the database system is often analyzing and deriving new data from existing data collected on the OLTP side.

An example of an OLAP workload that analyzes data collected on the OLTP side is personalized Amazon shopping ads. The website analyzes all of data collected from users' carts and purchases, and then selects different ads for different users.

HTAP: Hybrid Transaction + Analytical Processing

e.g. SingleStore

A new type of workload (which has become popular recently) is HTAP, where OLTP and OLAP workloads are present together on the same database instance.

Workload Focus:

- OLTP - Simple Queries, Write Heavy → Row Store
- OLAP - Complex Queries, Read Heavy → Column Store
- HTAP - A mix of both the above

2 Storage Models

There are different ways to store tuples in pages.

N-Ary Storage Model (NSM)

In the n-ary storage model, the DBMS stores all of the attributes for a single tuple (row) contiguously in a single page. Thus, it is also known as a **row store**. This approach is ideal for OLTP workloads where requests are write-heavy and accesses are mostly individual entities. It is ideal because it takes only one fetch to be able to get all of the attributes for a single tuple. NSM pages are typically some constant multiple of **4KB** hardware pages.

> **Advantages:**

- Fast inserts, updates, and deletes.
- Good for queries that need the entire tuple (OLTP).
- Can use index-oriented physical storage for clustering.

Disadvantages:

- Inefficient for scanning large portions of the table and/or a subset of the attributes.
- Poor memory locality in access patterns.
- Difficult to apply compression because of multiple value domains within a single page.

Decomposition Storage Model (DSM)

In the decomposition storage model, the DBMS stores a single attribute (column) for all tuples contiguously in a block of data. Thus, it is also known as a **column store**. This model is ideal for OLAP workloads with many read-only queries that perform large scans over a subset of the table's attributes.

Advantages:

- Reduces the amount of I/O wasted per query because the DBMS only reads the attributes that it needs for a given query.
- Better (faster) query processing because of increased locality and cached data reuse.
- Better data compression.

Disadvantages:

- Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.

To put the tuples back together when using a column store, there are two common approaches:

- The most commonly used approach is **fixed-length offsets**. Here, the value in a given column will belong to the same tuple as the value in another column at the same offset. Therefore, every single value within the column will have to be the same length.
- A less common approach is to use **embedded tuple ids**. Here, for every attribute in the columns, the DBMS stores a tuple id (ex: a primary key) with it. Then, the system would also store a mapping to tell it how to jump to every attribute that has that id. Note that this method has a large storage overhead because it needs to store a tuple id for every attribute entry.

Partition Attributes Across (PAX)

In the hybrid Partition Attributes Across storage model, the DBMS vertically partitions attributes within a database page. The goal of doing so is to get the benefit of faster processing on columnar storage while retaining the spatial locality benefits of row storage.

In PAX, the rows are horizontally partitioned into groups of rows. Within each row group, the attributes are vertically partitioned into columns. Each row group is similar to a column store for its subset of the rows.

A PAX file has a global header containing a directory with offsets to the file's row groups, and each row group maintains its own header with meta-data about its contents.

3 Database Compression

Compression is widely used in disk-based DBMSs as disk I/O is (almost) always the main bottleneck. It is especially popular in systems that have read-only analytical workloads. The DBMS can fetch more useful tuples, if they have been compressed beforehand at the cost of greater computational overhead for

compression and decompression.

In-memory DBMSs are more complicated since they do not have to fetch data from disk to execute a query. Memory is much faster than disks, but compressing the database reduces DRAM requirements and processing. They have to strike a balance between **speed** vs. **compression ratio**. Compressing the database reduces DRAM requirements and may decrease CPU costs during query execution.

Given this, we want a database compression scheme to have the following properties:

- Must produce fixed-length values. The only exception is variable length data stored in separate pools. This is because the DBMS should follow word-alignment and be able to access data using offsets.
- Allow the DBMS to postpone decompression as long as possible during query execution (**late materialization**).
- Must be a **lossless** scheme because people do not like losing data. Any kind of **lossy** compression has to be performed at the application level.

Compression Granularity 压缩粒度

The kind of data we want to compress greatly affects which compression schemes can be used. There are four levels of compression granularity:

- **Block Level:** Compress a block of tuples for the same table.
- **Tuple Level:** Compress the contents of the entire tuple (NSM only).
- **Attribute Level:** Compress a single attribute value within one tuple. Can target multiple attributes for the same tuple.
- **Columnar Level:** Compress multiple values for one or more attributes stored for multiple tuples (DSM only). This allows for more complicated compression schemes.

4 Naive Compression

The DBMS compresses data using a general purpose algorithm (e.g., Deflate, LZO, LZ4, Snappy, Oracle OZIP, Zstd, Lizard). Although there are several compression algorithms that the DBMS could use, engineers often choose ones that often provides lower compression ratio in exchange for faster compression/decompression.

The DBMS compresses disk pages, pads them to a power of 2KBs and stores them into the buffer pool. However, every time the DBMS tries to read/update data, the compressed data in the buffer pool must first be decompressed. For blind writes, no decompression required.

problems

Since accessing data requires decompression of compressed data, this limits the scope of the compression scheme. If the goal is to compress the entire table into one giant block, using naive compression schemes would be impossible since the whole table needs to be compressed/decompressed for every access. Therefore, MySQL breaks the table into smaller chunks since the compression scope is limited. e.g InnoDB Engine

Another problem is that these naive schemes also do not consider the high-level meaning or semantics of the data. The algorithm is oblivious to both the structure of the data, and how the query is planning to access the data. Thus, this eliminates the opportunity to utilize late materialization, since the DBMS cannot tell when it can delay the decompression of data.

5 Columnar Compression

Built-in compression schemes that allow DBMSs to read tuples without having to decompress them to their original form.

Run-Length Encoding (RLE)

RLE compresses runs (consecutive instances) of the same value in a single column into triplets:

1. The value of the attribute
2. The start position in the column segment (offset)
3. The number of elements in the run (length)

The DBMS should sort the columns intelligently beforehand to maximize compression opportunities. This clusters duplicate attributes, thereby increasing the compression ratio. Note that the effectiveness of RLE greatly depends on the underlying data characteristics (e.g. number and frequency of attributes in each data).

Bit-Packing Encoding

When all values for an attribute are less than the value's declared largest size, store them with fewer bits.

Patching / Mostly Encoding

Bit-packing variant that uses a special marker to indicate when a value exceeds the largest size and then maintains a look-up table to store them. Use when values are "mostly" less than the largest size.

Bitmap Encoding

e.g INTEGER → bit

The DBMS stores a separate bitmap for each unique value of a particular attribute where an offset in the vector corresponds to a tuple. The i^{th} position in the bitmap corresponds to the i^{th} tuple in the table and indicates whether that value is present in the attribute of that tuple. The bitmap is typically segmented into chunks to avoid allocating large blocks of contiguous memory.

This approach is only practical if the value cardinality is low, since the size of the bitmap is linearly proportional to the cardinality of the attribute value. If the cardinality of the value is high, then the bitmap can become larger than the original data set.

There are compressed data structures to for sparse data sets (for example: Roaring Bitmaps)

Delta Encoding

Instead of storing exact values, record the difference between values that follow each other in the same column. The base value can be stored in-line or in a separate look-up table. We can also use RLE on the stored deltas to get even better compression ratios.

Dictionary Compression

The most common database compression scheme is dictionary encoding. The DBMS replaces frequent patterns in values with smaller codes. It then stores only these codes and a data structure (i.e. the dictionary) that maps these codes to their original value. A dictionary compression scheme needs to support fast encoding/decoding, and needs to be order-preserving if it needs to work with range queries.

Encoding and Decoding: The dictionary needs to decide how to **encode** (convert uncompressed value into its compressed form) and **decode** (convert compressed value back into its original form) data. As such, it is not possible to use hash functions.

The encoded values also need to support sorting in the same order as original values (i.e. **order-preserving encodings**). This ensures that the compressed queries run on compressed data return results that are

consistent with uncompressed queries run on the original data. This order-preserving property allows operations to be performed directly on the codes.

On certain queries (such as string match), queries can be executed faster.