CS60203: Design Optimization of Computing Systems *November 01,2024*

Database storage models

Instructor: Mainack Mondal *Scribe-By: Parthiv Reddy Anuguru*

1 Summary

Databases use two primary storage models: the N-ary Storage Model (NSM) and the DSM -Decomposition Storage Model. NSM stores entire tuples together, making it ideal for transactional workloads (OLTP) but inefficient for queries accessing subsets of attributes, as it has poor memory locality and limited compression opportunities. DSM stores attributes column-wise, optimizing analytical queries (OLAP) by reducing I/O and enabling better compression but making point queries and updates slower. A hybrid approach, like Partition Attributes Across (PAX), balances these trade-offs. Key optimizations include using B-trees for fast lookups, transparent huge pages to reduce mapping overhead, efficient NULL handling, and appropriate numeric representations. Poor indexing, inefficient data layout, and ORM overuse can significantly impact performance, emphasizing the importance of careful design for scalable database systems.

2 Database storage

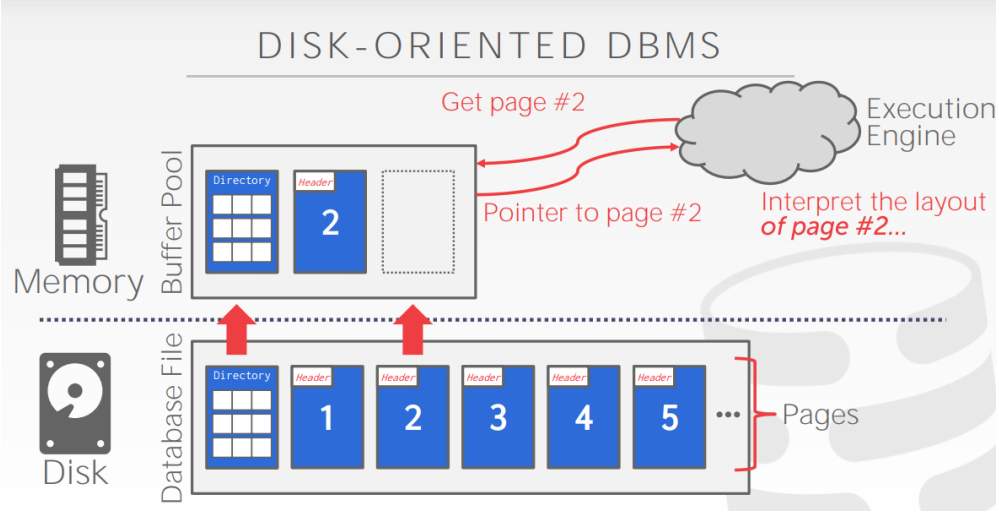

Figure 1: Disk oriented database

The DBMS stores a database as one or more files on disk. The OS doesn't know anything about the contents of these files. The **storage manager** in a database system is responsible for managing the database's files and ensuring efficient data storage and retrieval. It handles **scheduling** of read and write operations to optimize spatial and temporal locality, which improves performance by minimizing access times and making better use of caching mechanisms. Files are organized as

collections of **pages**, which are the fundamental units of data storage in a database. The storage manager keeps track of which data is read or written to specific pages and monitors available space to ensure efficient use of storage. A page is a fixed-size block of data. It can contain tuples, meta-data, indexes, log records etc., What data is stored and how is it stored in a page are to be chosen as per the workload requirement. There are multiple notions of page here we are talking about database pages.

3 N-ary storage model

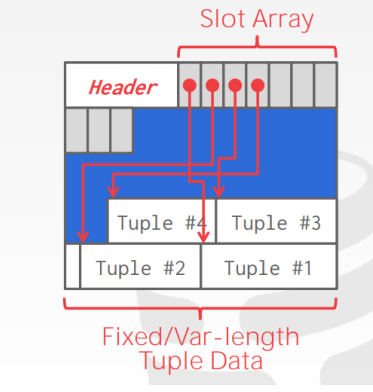

Figure 2: Slotted page

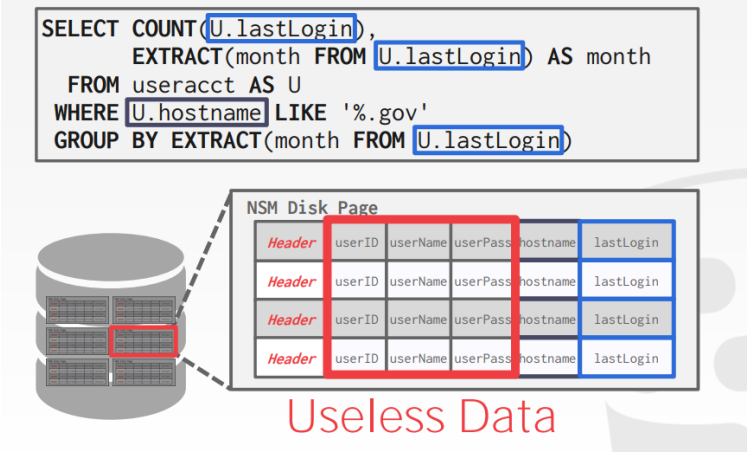

Figure 3: NSM disadvantage

It means all the attributes belonging to a row are continuously stored in a page. Ideal for OLTP workloads where queries tend to operate only on an individual entity and insert heavy workloads. Slotted pages are used for this purpose. The slot array maps "slots" to the tuples' starting position offsets. The header keeps track of the number of used slots and the offset of the starting location of the last slot used. As the figure : 2 depicts this approach allows use to store both fixed length and variable length attributes using the data tracked by the header. To look at how tuples are filled into the page refer slides:pg.30. Each tuple is prefixed with a header that contains meta-data about it. Visibility info (concurrency control) and bit Map for NULL values. And NSM in practice works as shown in figure : 3. Where some subset of attributes is required but all the attributes are brought to memory which is a performance cost. Since different datatypes are present in continuous memory it is difficult to compress the data as well. Nary model is good to use when all the attributes are involved in query like insert queries.

3.1 Logical Distribution vs. Physical Layout

In databases, schemas and tablespaces serve distinct purposes that separate the logical organization of objects from their physical storage. Schemas and databases define the logical structure, organizing data into tables, views, and other database objects. In contrast, tablespaces determine the physical layout, controlling where and how the data is stored on disk. This separation allows for greater flexibility in managing the physical and logical aspects of the database independently.

3.2 What is a Tablespace?

A tablespace in a database is implemented as a directory in the file system where data files are stored. It is a physical storage unit that provides the flexibility to distribute data across different storage devices. For example, frequently accessed data or data that requires high performance can be stored on fast disks (e.g., SSDs), while rarely accessed or archival data can be placed on slower, larger disks (e.g., HDDs). This approach helps optimize performance and cost efficiency.

3.3 Shared Use of Tablespaces

Tablespaces can be shared across multiple databases, meaning that the same tablespace can be used to store data for different databases. Additionally, a single database can utilize multiple tablespaces to distribute its data. This enables a database to store different tables or indexes in separate tablespaces based on usage patterns, access frequency, or size requirements. For instance, a large table might be placed in a dedicated tablespace on a high-capacity disk, while smaller, frequently updated tables could be stored in a tablespace on a high-speed disk.

3.4 Logical Structure and Physical Layout are Independent

A key advantage of tablespaces is that they decouple the logical organization of data (schemas and databases) from the physical layout on disk. This means that changes to the logical structure, such as renaming a schema or adding new tables, do not require changes to the physical data layout. Similarly, physical storage configurations, such as relocating tablespaces to new storage devices, do not affect the logical structure of the database.

3.5 Default Tablespaces

Each database has a default tablespace, which serves as the default storage location for objects created in that database unless a specific tablespace is specified during object creation. This default tablespace also stores system catalog objects related to the database. If no other location is defined, all tables, indexes, and other objects are placed in this default tablespace, ensuring a consistent and organized storage approach.

3.6 File-Level Layout in PostgreSQL

In PostgreSQL, the physical storage of tables and indexes is carefully organized to ensure efficient data management and retrieval. At the **file level**, each table and index is stored in a **separate file**, which simplifies physical data management and allows PostgreSQL to scale efficiently. These files are named based on a unique identifier called the `file-node number`, which is stored in the `pg_class.relfilenode` column. This ensures that each table or index can be uniquely identified and managed on disk.

3.6.1 Free Space Map (FSM)

Every table and index is associated with a **Free Space Map (FSM)**. The FSM tracks the amount of **free space** available on each page of the relation. This is crucial for operations like *inserts* and *updates*, as it allows PostgreSQL to quickly locate pages with enough free space to accommodate new rows or modified data. The FSM is stored as a separate file, named using the file-node number of the table or index, with the suffix `_fsm`. By maintaining this map, PostgreSQL

minimizes the need for scanning unnecessary pages, improving write performance and storage efficiency.

3.6.2 Visibility Map (VM)

PostgreSQL also associates a **Visibility Map (VM)** with each table, which is stored in a **fork** (a separate logical extension of the table's physical storage) using the file-node number with the suffix `_vm`. The visibility map tracks which pages of a table have **no dead tuples**, meaning all tuples on those pages are visible to all active transactions. This information is critical for optimizing *index-only scans* and *vacuuming operations*. During an index-only scan, PostgreSQL can skip reading pages marked as "fully visible" in the visibility map, significantly improving query performance. Similarly, the `VACUUM` process uses the visibility map to avoid scanning pages that don't require cleaning, reducing overhead.

3.7 Decomposition Storage Model (DSM)

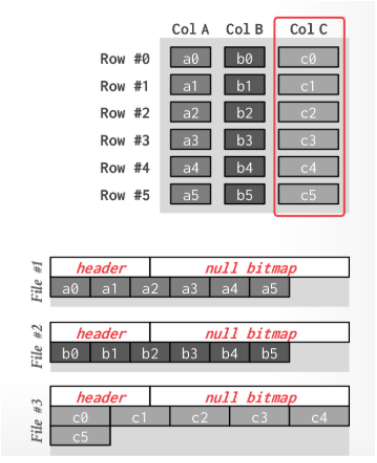

Figure 4: DSM

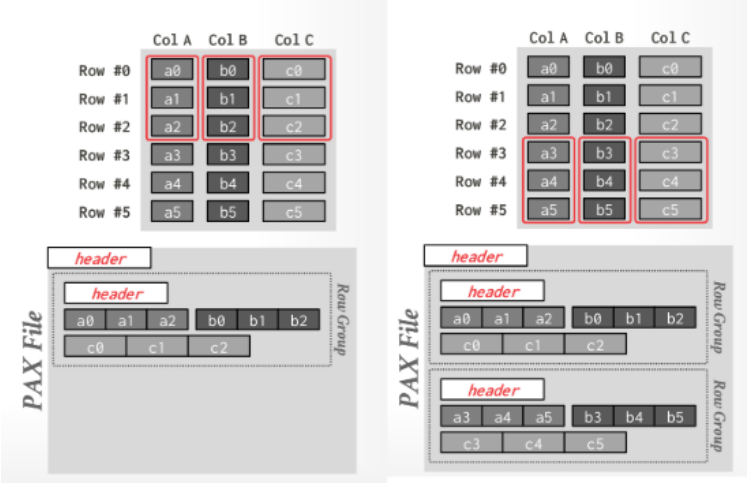

Figure 5: PAX-hybrid of NSM and DSM

The **Decomposition Storage Model (DSM)**, also known as columnar storage, stores a single attribute (column) for all tuples contiguously in a block of data. This design is **ideal for OLAP workloads** where read-only queries perform large scans over a subset of the table's attributes. DSM typically utilizes a *batched vectorized processing model* to process queries efficiently.

File Organization: In DSM, files are typically larger (hundreds of megabytes), but tuples may still be organized within these files into smaller groups for easier management and processing. Attributes and metadata, such as null values, are stored in **separate arrays** of fixed-length values to streamline access and processing. Most systems identify unique physical tuples using *offsets* into these attribute arrays.

Handling Variable-Length Values: For attributes with variable-length values, DSM maintains a **separate file per attribute**. These files include a dedicated *header area* containing metadata about the entire column, enabling efficient management and access to variable-length data.

Advantages: By storing each attribute contiguously, DSM significantly improves performance for analytical queries that require scanning specific columns rather than entire rows. This storage layout reduces I/O costs and improves memory locality, making it well-suited for operations such as aggregation, filtering, and data summarization.

3.8 Huge Pages: Motivation and Transparent Huge Pages (THP)

Modern CPUs, such as AMD's Zen 4 Microarchitecture, have limited **Translation Lookaside Buffers (TLBs)** to ensure speed. For example, Zen 4 has a first-level TLB with only 72 entries and a second-level TLB with 3072 entries. This limits the number of pages that can be cached in the TLB. For applications with working sets larger than approximately 4 KiB $\times$ 3072 = 12 MiB, frequent **page table lookups** become inevitable, which increases memory access latency.

Transparent Huge Pages (THP): To address this limitation, Linux supports allocating memory in **larger pages** (2MB to 1GB) instead of the default 4KB pages. This reduces the number of TLB entries required, improving performance by minimizing TLB misses. THP enables the OS to manage these large pages transparently:

- Split larger pages into smaller pages when needed.
- Combine smaller pages into larger pages.

However, creating huge pages requires **contiguous memory**, and the OS must perform memory defragmentation in the background, which can cause latency and stall DBMS processes during memory access.

Issues with Transparent Huge Pages: While THP can reduce TLB misses, its transparency comes at a cost due to defragmentation overhead. Many DBMS vendors, including Oracle, SingleStore, MongoDB, and Sybase, historically recommend disabling THP in Linux. Some systems, like Vertica, recommend enabling THP only for newer Linux distributions. Despite these challenges, huge pages (transparent or otherwise) often improve performance. Hence, it is advised to **measure performance impact** before enabling or disabling THP.

3.9 Numeric Representation

Variable Precision: Variable precision types, such as `FLOAT` or `REAL/DOUBLE`, are **inexact numeric types** that use native `C/C++` representations. These are stored directly as specified by the **IEEE-754** floating-point standard. Such types are typically **faster** than fixed precision because modern CPUs (e.g., Intel Xeon, ARM) include specific instructions and registers for floating-point arithmetic. However, they do not guarantee **exact values**, which makes them unsuitable for use cases where precision is critical.

Fixed Precision: Fixed precision types ensure **exact values** but are generally less efficient than variable precision. For example, PostgreSQL's `NUMERIC` type supports arbitrary precision but is computationally expensive. These are implemented using a variable-length binary representation along with additional metadata. Fixed precision should only be used when **rounding errors are unacceptable**, such as in financial applications. While optimizations exist, their use is discouraged unless precision is essential.

3.10 Representing NULLs

Efficiently representing NULL values in databases is critical for both performance and storage optimization. Below are the three common approaches:

- **Special Values:** Use a reserved value (e.g., `INT32_MIN`) to represent `NULL`. While simple, this approach may introduce ambiguities if the reserved value is valid for some use cases.
- **Null Column Bitmap Header:** Store a bitmap in a centralized header that specifies which attributes are `NULL`. This approach is compact and minimizes storage overhead but requires additional processing to check the bitmap during queries.
- **Per-Attribute Null Flag:** Store a dedicated flag for each attribute to indicate whether it is `NULL`. While straightforward, this method consumes more space than a single bit due to **word alignment issues**, which can increase memory overhead.

3.11 References

All the images used in this slides are taken from advanced databases course by CMU.