Disk-based Record Manager

Database Management Systems Laboratory

CS39202

Team FooClub

Name	Roll Number
Akshat Pandey	22CS10005
Devansha Dhanker	22CS10021
Sahil Asawa	22CS10065
Abhinav Akarsh	22CS30004
Pranav Jha	22CS30061

Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

1 Problem Statement

Efficient storage and retrieval of data from disk is a key challenge in database systems, especially when dealing with large volumes of data and complex queries. In this project, we aim to explore various methods of storing records on disk and accessing them in an optimized way.

We study different record storage protocols and design our own approach that improves performance using techniques like buffering and indexing. To evaluate query processing efficiency, we implement and compare several query algorithms, including:

- Nested Loop Join
- Hash Join
- Merge Join
- B+ Tree-based Join
- Linear Search Select
- B+ Tree-based Select

By analyzing their performance under different workloads, we aim to identify which methods work best in specific scenarios. Through this project, we hope to gain practical insights into optimizing data storage and query execution in real-world database systems.

2 Methodology

This report explores various different approaches to querying data (SELECT and JOIN) stored in a disk-based storage system: iterating over the dataset directly, using an index/hash structure, and external merge sort across the search key. Both methods are evaluated under varying disk access strategies (RANDOM and SEQUENTIAL) and buffer replacement strategies (LRU and MRU). The aim of this study is to compare the performance of these techniques in terms of query execution time, resource utilisation, and their ability to efficiently handle range queries. The subsequent sections will provide detailed descriptions of each approach, followed by a comparison of the results and their associated statistics.

The system defines fixed-size structs to ensure consistent memory alignment and enables direct memory manipulation for high-performance operations. The core types (Employee, Company, and JoinEmployeeCompany) are defined with exact 128 or 256-byte sizes.

2.1 Disk

In this project, we implemented a global absolute addressing system where all addresses generated by the processes directly correspond to physical addresses on the disk. To perform read or write operations on a specific address, a process sends a request to the buffer manager. The buffer manager first checks if the requested address is present in the buffer. If not, it issues a block read request to the disk manager to fetch the block containing the required address. The disk manager reads the block from disk and returns it to the buffer manager. In the case of a write operation, the buffer manager updates the data in the buffer and marks the corresponding block as dirty, indicating it needs to be written back to disk later.

The Disk class simulates a block-based storage device using a file as the underlying medium. The simulated disk is divided into fixed-size blocks and supports both sequential and random access. Sequential access incurs additional cost based on the distance between consecutive block accesses. During initialization, the total number of blocks is calculated based on the disk and block sizes, and the backing file is pre-filled with empty blocks to closely mimic real disk behavior. The class also maintains a record of the number of I/O operations and their associated costs, which are updated every time a read or write is performed.

When reading from or writing to a block, the Disk class verifies the validity of the block number, calculates the access cost-taking into account the access type and the current file pointer position-seeks to the correct offset in the file, and then carries out the operation. All data is handled in binary form using byte-level vectors to replicate realistic block-level disk behavior. This approach enables an accurate and measurable simulation of disk operations, making it a valuable tool for testing and analysis of buffer and disk management techniques.

2.2 Buffer

The BufferManager class is designed to efficiently manage a memory buffer that stores disk blocks in RAM, significantly reducing disk I/O operations by caching frequently accessed pages. It supports two page replacement strategies-Least Recently Used (LRU) and Most Recently Used (MRU)-to decide which page to evict when space is needed. The class maintains important metadata, including pin counts, dirty bits, and mapping tables, which help track frame usage and content. This metadata ensures that data is handled properly in memory and that modified data is safely written back to disk when evicted.

The buffer pool consists of fixed-size frames, each capable of holding a single page of data. For simplicity and consistency, the page size, frame size, and block size are assumed to be the same. The mapping between disk pages and buffer frames is maintained using a global page table. When a process requests a page, the getFrame() method is invoked to either return a free frame or, if none are available, apply the selected replacement policy to evict an existing page. If the evicted page is marked dirty, it is first written back to disk to ensure that no data is lost.

To maintain data consistency, the buffer manager ensures that all dirty frames are flushed to disk before shutdown. This guarantees that any modifications made during execution are safely stored, even in the case of a system failure. Through careful management of memory resources and strategic replacement policies, the buffer manager plays a critical role in optimizing performance and preserving data integrity in the overall disk management system.

2.3 B+ Tree

The B+ Tree maintains a hierarchical structure where internal nodes only store keys and guide the traversal, while leaf nodes store actual key-value pairs. Each node-whether internal or leaf-is assigned a unique identifier and managed through a buffer system for disk I/O operations. An invariant is that internal nodes have one more child pointer than keys, maintaining the search property across the tree. Leaf nodes are linked sequentially, allowing for efficient range queries and ordered traversal.

The class supports core operations like insertion, deletion, and searching (including range-based), while internally handling tasks like node splitting and key promotion for balance. Iterators are implemented to allow traversal starting from the leftmost leaf node, moving sequentially using nextLeaf_id. All disk and memory interactions are abstracted using a

BufferManager, and space is pre-allocated on disk based on the expected maximum size of keys and values. The order of the tree (maximum children per node) can be varied as per need to achieve required performance.

To perform the join between Employees and Companies using a B+ Tree, we first construct indices on the join attribute, Company ID, for both relations. With the records sorted via their respective indices, a merge-style join is executed using two iterators traversing the sorted data streams. For selection operations, B+ Tree indices are particularly beneficial when executing range queries over the key attribute. Once all relevant pointers to result tuples are retrieved, we sort them by their physical addresses on disk. This optimization leverages spatial locality to minimize disk I/O by reducing unnecessary block accesses during tuple retrieval.

2.4 Hash Index

Hash Indexing revolves around two templated classes - Bucket and ExtendableHashIndex. Each Bucket stores a list of key-value pairs with a fixed bucketSize, a localDepth to support splitting during collisions, and functions like insert, search, and deleteKey. The main index, ExtendableHashIndex, uses a directory pointing to these buckets and employs hashing (through getBucketNo) to locate data. When a bucket overflows, it may be split (increasing its local depth) or trigger a global directory expansion (via grow).

The implementation efficiently supports operations like insertion, search, deletion, and resizing (split/merge) while abstracting disk storage with a BufferManager. The createBucket, loadBucket, saveBucket, and destroyBucket methods handle bucket persistence, making the index suitable for disk-based storage systems.

To perform the join between Employees and Companies using a Hash Index, we first build a hash index on the Company ID attribute of the Company relation. Once the index is constructed, we iterate over all Employee tuples and, for each Employee, perform a lookup on the hash index to retrieve the corresponding Company record. This approach enables efficient point lookups, significantly reducing the join cost compared to full scans, especially when the join attribute is well-distributed.

2.5 External Merge Sort

External Merge Sort efficiently sorts and joins two large datasets - Employee and Company records - using External Merge Sort and Sort-Merge Join. The sorting algorithm first divides the data into buffer-sized chunks, loads them into memory, sorts them individually (forming runs), and writes them back. If the number of runs exceeds available buffer frames, a multi-pass merge strategy is employed, recursively merging smaller subsets of runs until the sorted data fits within a single range. This approach ensures that the algorithm remains memory-efficient and scalable, while the use of a priority queue during merging guarantees optimal N-way merging with minimal comparisons.

Post-sorting, a merge join is applied to combine Employee and Company records based on a matching company_id. Since both files are now sorted, the join is performed using two pointers traversing the sorted datasets linearly, writing the joined result to a new address range. This avoids the costly nested loop join, making it ideal for large-scale data.

2.6 Nested Loop

In order to evaluate the performance of join operations in a disk-based database system, we implemented and analysed the Nested Loop Join algorithm with varying configurations. We examined the effect of different outer relations (either Employee or Company), disk access strategies (RANDOM and SEQUENTIAL), and buffer replacement policies (LRU and MRU). By measuring and recording statistics under all eight combinations, this set a reference scale to compare with the other protocols implemented.

3 Demonstration

In this section, we showcase the performance evaluation of various join and select algorithms executed under multiple system configurations. The results are summarized in tabular format, detailing key performance metrics such as memory accesses, block read/write operations, and the overall disk access cost. These statistics were gathered by running each algorithm employing different buffer replacement policies and disk access strategies.

The dataset includes two files: employee.csv and company.csv. The employee.csv file contains 5,000 tuples, each with five attributes: ID, first name, last name, salary, and company ID. The company.csv file contains 1000 tuples, each with three attributes: ID, name, and slogan. The use of a fixed dataset ensures a fair and consistent basis for comparison across all techniques. By applying the same operations over identical input, we are able to better highlight the differences in performance caused by algorithmic and system-level variations.

The following data presents the performance of various replacement strategies and disk access types. Memory Accesses indicate the total number of read/write requests generated by the process, representing the number of I/O operations to the main memory. Block Read/Write reflects the number of block transfers triggered by the disk I/O requests made by the buffer manager. Disk Access Cost varies depending on the access type. Random Access treats each block transfer as having a constant cost of 1. On the other hand, Sequential Access considers both the block transfer cost and the seek cost, which accounts for the movement of the disk arm between the initial and final positions, thus affecting the overall cost.

Disk Access Type	Replace Strategy	Memory Accesses	Block Read/Write	Disk Access Cost
RANDOM	LRU	24024	4328	4328
RANDOM	MRU	24024	47548	47548
SEQUENTIAL	LRU	24024	4328	2049244
SEQUENTIAL	MRU	24024	47548	35777724

Table 1: Statistics of the External Sort with varying disk access and replacement strategies.

Disk Access Type	Replace Strategy	Memory Accesses	Block Read/Write	Disk Access Cost
RANDOM	LRU	58304	12643	12643
RANDOM	MRU	58304	70738	70738
SEQUENTIAL	LRU	58304	12643	6387900
SEQUENTIAL	MRU	58304	70738	53484799

Table 2: Statistics of the B+ Tree Index Creation with varying disk access and replacement strategies.

Disk Access Type	Replace Strategy	Memory Accesses	Block Read/Write	Disk Access Cost
RANDOM	LRU	8580	38	38
RANDOM	MRU	8580	3922	3922
SEQUENTIAL	LRU	8580	38	7355
SEQUENTIAL	MRU	8580	3922	2927807

Table 3: Statistics of the Hash Index Creation with varying disk access and replacement strategies.

Join Strategy	Memory Accesses	Block Read/Write	Disk Access Cost
Nested Join (Employee outer)	5006000	321764	321764
Nested Join (Company outer)	5010000	338684	338684
Merge Join (excluding sorting)	16998	1004	1004
Index Join	16870	14250	14250
Hash Join	20000	10685	10685

Table 4: Statistics for different join strategies using RANDOM disk access and LRU replacement.

Join Strategy	Memory Accesses	Block Read/Write	Disk Access Cost
Nested Join (Employee outer)	5006000	313404	313404
Nested Join (Company outer)	5010000	318644	318644
Merge Join (excluding sorting)	16998	29546	29546
Index Join	16870	32636	32636
Hash Join	20000	38760	38760

Table 5: Statistics for different join strategies using RANDOM disk access and MRU replacement.

Join Strategy	Memory Accesses	Block Read/Write	Disk Access Cost
Nested Join (Employee outer)	5006000	321764	165339293
Nested Join (Company outer)	5010000	338684	177972412
Merge Join (excluding sorting)	16998	1004	432409
Index Join	58304	12643	6387900
Hash Join	20000	10685	5479421

Table 6: Statistics for different join strategies using SEQUENTIAL disk access and LRU replacement.

Join Strategy	Memory Accesses	Block Read/Write	Disk Access Cost
Nested Join (Company outer)	5006000	313404	160219293
Nested Join (Employee outer)	5010000	318644	162761916
Merge Join (excluding sorting)	16998	29546	19226937
Index Join	16870	32636	22256636
Hash Join	20000	38760	29454651

Table 7: Statistics for different join strategies using SEQUENTIAL disk access and MRU replacement.

Select Strategy	Disk Access Type	Replace Strategy	Memory Accesses	Disk Access Cost
B+ Tree Index	RANDOM	LRU	135	127
Iterating Method	RANDOM	LRU	5000	157
B+ Tree Index	RANDOM	MRU	135	140
Iterating Method	RANDOM	MRU	5000	157
B+ Tree Index	SEQUENTIAL	LRU	135	29844
Iterating Method	SEQUENTIAL	LRU	5000	157
B+ Tree Index	SEQUENTIAL	MRU	135	47048
Iterating Method	SEQUENTIAL	MRU	5000	157

Table 8: Statistics for range queries using B+ Tree Index and Iterating Method with different disk access types and replacement strategies.

4 Results

By analyzing the above performance statistics, we uncover the interplay between disk access patterns, buffer replacement policies, and join/select strategies on overall performance.

4.1 External Sort (Table 1)

Under random access, LRU achieves only 4328 block transfers, whereas MRU incurs over 47000. This dramatic increase demonstrates MRU's poor locality: recently used pages get evicted prematurely, leading to thrashing during merging of runs. Sequential access dramatically increases cost due to head movement. Even with LRU, the seek cost inflates the disk cost. MRU sequential disk access cost is catastrophic, around quadratic in number of runs.

For external merge sort on limited buffer, LRU is strongly preferred. Random access yields predictable performance, but sequential access must be accompanied by careful replacement and run scheduling to avoid excessive seeks.

Although external sort may not be the most efficient choice for join operations alone. It can be used when we have further queries. It optimizes subsequent queries like selection and search, as the sorted output allows these operations to be executed more efficiently.

4.2 B+ Tree Index Creation (Table 2)

During the construction of a B+ Tree index, the number of memory accesses remains consistent across different disk configurations. This is because each node insertion inherently involves a fixed number of page reads and writes determined by the structure and distribution of the data itself, independent of disk access strategies.

However, the number of block transfers varies significantly with the buffer replacement policy. Notably, the LRU strategy results in nearly five times fewer block transfers compared to MRU. This is due to the spatial locality of node accesses, subsequent insertions tend to access nodes located close to recently used ones, which are more likely to be retained in memory under LRU. In contrast, MRU evicts recently accessed nodes, leading to frequent reloads from disk.

Furthermore, sequential disk access exhibits poor performance in this context, as the dynamic nature of tree balancing involves non-linear access patterns. Internal nodes and leaf blocks that require frequent updates may reside in non-contiguous physical locations, making sequential access inefficient due to the added seek overhead.

4.3 Hash Index Creation (Table 3)

Hash index creation incurs significantly lower disk I/O costs compared to other indexing methods. It requires only 38 block transfers under the LRU strategy, whereas B+ tree index creation demands over 12,000 block transfers due to repeated node splitting and rebalancing operations. This stark difference stems from the simplicity and direct access nature of hash indexing, which avoids the structural maintenance overhead of tree-based indices. Additionally, as observed with B+ trees, the MRU replacement strategy results in substantially higher disk access costs than LRU in this case also.

4.4 Join Strategies (Tables 4,5,6,7)

These tables compare nested loop, merge, index, and hash joins on employee and company records across different disc access and buffer replacement protocols.

Nested Loop Join

Nested Loop Join exhaustively scans all pairs of tuples across the two relations, leading to poor performance, especially for large datasets. Due to its simplicity, it is often used as a baseline for benchmarking more optimized join algorithms. While generally inefficient, it can be advantageous in scenarios where the join produces a large result set, as it avoids the overhead of sorting or indexing. The disk access cost of Nested Loop Join is proportional to the product of the number of tuples in the two relations, making it highly sensitive to input size.

External Merge Sort Join

The merge join algorithm performs very good for joining the data. This is by far the most I/O efficient join, exploiting sorted order and full block streaming. LRU performs much better than MRU. But this comes with an extra cost of sorting the data.

B+ Tree Index Join

Index Join significantly outperforms Nested Loop Join by leveraging existing indices to directly access matching tuples, reducing unnecessary scans. Merge Join offers even better performance, especially when the inputs are already sorted or indexed-common in many practical scenarios, thus avoiding additional sorting overhead. As a result, Merge Join combines efficiency with practicality, making it a preferred choice when sorted inputs or indices are available.

Hash Index Join

Hash Index Join requires building an index on only one of the relations, unlike B+ Tree Index Join, which typically needs indices on both inputs. For moderate-sized datasets, the hash index can often be stored entirely in memory, allowing efficient in-buffer lookups and minimizing disk I/O. This makes Hash Index Join particularly effective in practice, offering near-optimal performance without the overhead of sorting or complex tree maintenance.

4.5 Selection Strategies (Table 8)

The full scan algorithm significantly underperforms compared to index-based search in most scenarios. However, it can be efficient when the query results cover a large portion of the table, making sequential access beneficial. For better performance of index search, the data are retrieved in order of their physical addresses. In contrast, index searches excel in sparse queries where only a small subset of data needs to be retrieved, reducing unnecessary disk reads. However for better performance of index search even for dense results, the data are retrieved in order of their physical addresses.

5 Conclusion

The above performance evaluation of various indexing and join strategies under different disk configurations, we observe that disk access patterns, buffer replacement policies, and algorithmic choices have too much impact on overall system efficiency.

External sorting, although not the most efficient for join operations in isolation, proves beneficial in multi-query scenarios where sorted data enhances the efficiency of subsequent operations like selection and search. Among replacement policies, LRU consistently outperforms MRU, especially under random access patterns, due to its ability to retain pages with higher spatial locality.

For B+ Tree index creation, memory access remains largely unaffected by disk type or policy, but block transfers differ substantially. LRU demonstrates significantly lower block transfer costs, highlighting the importance of spatial locality in index structures. Sequential disk access, despite its theoretical efficiency, underperforms due to the non-contiguous nature of tree updates.

Hash index creation, in contrast, exhibits minimal block transfer costs and excellent performance when the index fits in memory, making it highly effective for moderate-sized datasets. However, the choice of replacement strategy still influences performance, with LRU again outperforming MRU.

In join strategies, nested loop join performs poorly due to its exhaustive pairwise comparisons and is best used as a baseline for comparison. Index and merge joins offer far better performance, with merge join benefiting from pre-sorted data. Hash joins strike a balance by requiring indexing on only one relation and offering excellent buffer efficiency.

Finally, in selection strategies, full scans are outperformed by index-based searches in sparse query scenarios. Nonetheless, full scans remain competitive when queries cover a majority of the data. Efficient index searches also rely on aligning data access with physical storage order, enhancing cache and disk utilization.

This project highlights the importance of choosing the right data structure for different characteristics such as disk access type and buffer policy.

6 Future Scope

6.1 B+ Tree

- **Persistence:** Currently, the index is built every time we run a program. We can make the B+ Tree persistent by building it once and storing it the disk. Then loading it whenever required.

- **Key Type Limitation:** Currently supports only `string` and `int` types as keys. Extend support to additional data types.
- **Duplicate Entries:** Improve handling of duplicate key entries by using structures like linked lists of record pointers or overflow areas.

6.2 Hash Indices

- **Hash Function Diversity:** Explore and evaluate a variety of hash functions such as MurmurHash, CityHash, or custom-designed hash functions to enhance indexing performance and distribution uniformity.

6.3 External Merge Sort

- **Frame Usage Optimization:** Right now, each frame corresponds to a single frame in memory. Enhancements can include supporting multiple frames per run to reduce disk seek overhead and improve performance.

6.4 Buffer Manager

- **Replacement Strategies:** Incorporate advanced replacement algorithms like LRU-K, CLOCK, and ARC to improve buffer efficiency.
- **Pinning System:** Add a pinning mechanism to avoid evicting crucial pages during operations that rely on them.
- **Locking for Concurrency:** Introduce locking mechanisms to ensure safe exclusive access to buffer frames during concurrent transactions.

6.5 Disk Manager

- **Cost Estimation Improvements:** The current system only considers the difference between arm positions. More accurate disk models could include rotational latency and seek time calculations. Also include time to read a block.

6.6 Record Structure

- **Support for Variable-Length Records:** At present, only fixed-length records are supported. Variable-length records are stored inefficiently, leading to internal fragmentation.
- **Efficient Storage Techniques:** Future enhancements can involve using slotted page formats or variable-length record directories to minimize fragmentation and optimize space usage.

References

[1] S. Sudarshan Avi Silberschatz, Henry F. Korth. Database system concepts, 2025. URL https://www.db-book.com/.

[2] CS30202. Database Management Systems, 2024-25 (Spring), CSE IIT Kharagpur, 2025. URL https://cse.iitkgp.ac.in/~ksrao/cou-dbms-2025.html.

[3] Valerio Di Stefano, External Merge Sort Simulations. URL https://valeriodiste.github.io/ExternalMergeSortVisualizer.
[1] [2] [3]