SC3020 Database Systems Principles

Project 1 Report

Code: https://bit.ly/SC3020Project1
Presentation Video: https://bit.ly/SC3020Project1Video

Name	Email	Matric Number
Cui Nan	C220133@e.ntu.edu.sg	U2221495L
Pu Fanyi	FPU001@e.ntu.edu.sg	U2220175K
Shan Yi	SH0005YI@e.ntu.edu.sg	U2222846C
Zhang Kaichen	ZHAN0564@e.ntu.edu.sg	U2123722J
Tian Yidong	YTIAN006@e.ntu.edu.sg	U2220492B

College of Computing and Data Science
Nanyang Technological University, Singapore

2024/2025 Semester 1

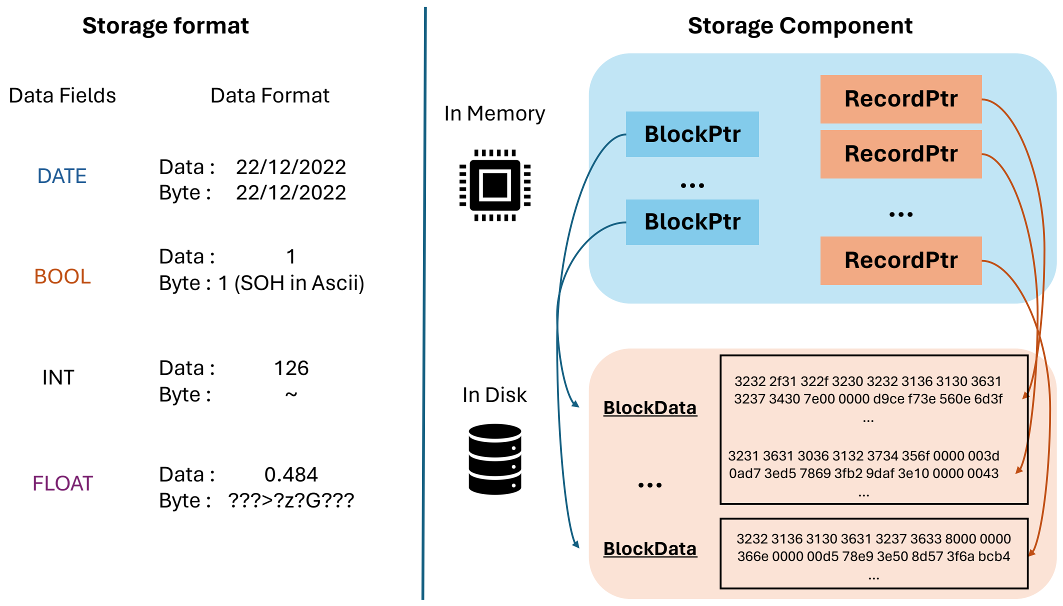
1 Introduction

This project focuses on the design and implementation of two key components of a database management system: storage and indexing.

Our report is structured as follows:

1. First, we illustrate the design of our storage component, where data is stored in blocks on computer disks.
2. Next, we describe the implementation of our B+ tree and the process of storing the index on the disk.
3. Finally, we present the statistics and experimental results obtained using our database system.

2 Storage Components



```
3232 2f31 322f 3230 3232 3136 3130 3631
3237 3430 7e00 0000 d9ce f73e 560e 6d3f
...

3231 3631 3036 3132 3734 356f 0000 003d
0ad7 3ed5 7869 3fb2 9daf 3e10 0000 0043
...

3232 3136 3130 3631 3237 3633 8000 0000
366e 0000 00d5 78e9 3e50 8d57 3f6a bcb4
...
```

Figure 1: Illustration of the storage components design in our project. All the data are converted into array of Byte and store in the disk. Only pointers that points to the offset position will be stored in memory

In this section, we illustrate the design of our storage components. We begin by listing all the components in our design, followed by a demonstration of how each component interacts with the others.

2.1 Data Components

The data components primarily consist of the following objects:

- **BlockData**: This object stores the actual data. The data is stored as an array of bytes, with the size determined by the block size of different computers.
- **BlockPtr**: This object points to the position of the `BlockData`. Given a file stream, the `BlockPtr` can access a specific range of positions within that file stream.
- **DataPtr**: This object points to a specific range of positions within a `BlockData`. In real-time operations, a `BlockPtr` is used to manage the loading and storing of the `DataPtr`.
- **Record**: A Record is a subclass of the `DataPtr`. It can access a specific range within the `BlockPtr`, corresponding to the size of the record.

- **Schema**: The Schema manages the field names and data types of a table. A Schema object is used to determine the number of bytes required to write input data and the number of bytes needed to read a record when loading from an array of bytes.

During the reading and writing processes, all data is converted into a uniform format, specifically an array of bytes. For example, an int or float data type is converted into an array of size 4 bytes, while a bool data type is converted into an array of size 1 byte. During reading, we access only the BlockPtr or RecordPtr, which points to a specific location in the data. When a specific range is needed, the corresponding byte array is loaded into memory. A detailed illustration of our storage format and components is provided in Figure 1.

2.2 Controller Components

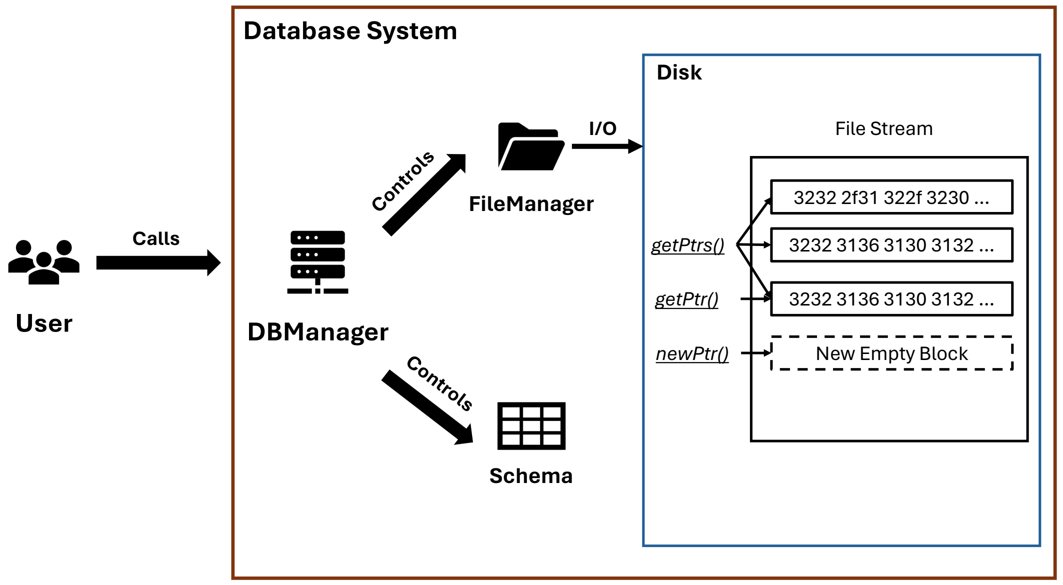


Figure 2: The controller components in how user interactive with the database.

We will now illustrate our controller components, which are primarily responsible for managing the data components described in Section 2.1. The two main controller objects are as follows:

- **FileManager**: This object manages all BlockPtr instances and the file stream. It handles read and write operations between the file and the buffer pool.
- **DataBaseManager**: This object controls the database, using the FileManager to load data or allocate new memory blocks for data storage. It also oversees the process of reading from the original text file or from the byte array, returning a pointer to the record position.

When interacting with the user, the DataBaseManager calls the FileManager to allocate new memory or read from existing memory. During a `load()` operation, the DataBaseManager initiates an I/O operation for one block, as detailed in Figure 2.

2.3 Buffer Pool

We use the buffer pool to manage disk read and write operations uniformly. All database read and write operations must go through the buffer pool.

A buffer pool $\mathcal{P}$ consists of the following properties:

1. A hash table $\mathcal{H}_{\mathcal{P}}$ in memory, using the disk offset as the key. $\mathcal{H}_{\mathcal{P}}(o)$ represents the block data corresponding to the disk offset o.

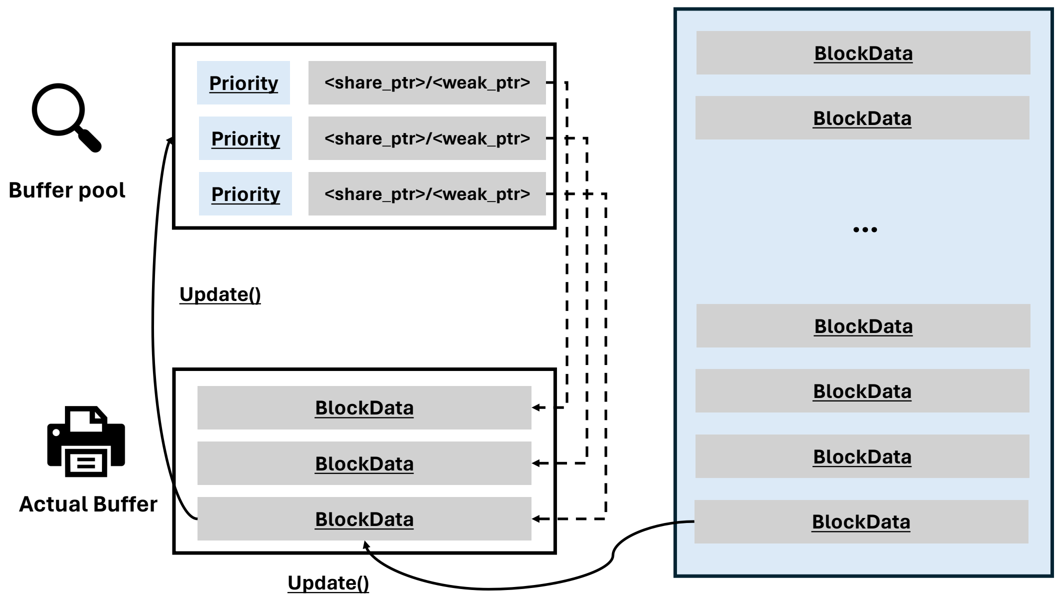


Figure 3: Design of our buffer pool. BlockData are shared with shared_ptr or weak_ptr so that no duplicate BlockData are constructed. During update, the data in the buffer pool will be override and the index and priority will be update

2. A Least Recently Used (LRU) queue $\mathcal{Q}_\mathcal{P}$ that tracks the access order of frames, with the least recently used page at the head. This queue implements the page replacement policy in the buffer pool. If the page is replaced in the memory, the block data in the buffer where the old page was stored is expired; when I/O is requested to access the data block corresponding to this buffer, the data block has to be loaded from the memory to this buffer.

3. Each frame in the buffer pool is associated with a dirty bit $\mathcal{D}$. If $\mathcal{D} = 1$, the frame has been modified in memory but not yet written back to disk. If $\mathcal{D} = 0$, the frame is synchronized with the corresponding block on disk.

During actual I/O operations, the `FileManager` manages the buffer and reuses it if the block is already cached in the buffer pool. If the buffer pool is full, it is updated using the *Least Recently Used (LRU)* policy described above. The structure of our buffer pool is illustrated in Figure 3, where each BlockData is shared using `std::shared_ptr` or `std::weak_ptr` to allow resource sharing. During updates, the data in the buffer is modified accordingly, and the pointers are updated to reflect the changes.

The entire structure simulates the LRU process, achieving $\mathcal{O}(1)$ time complexity for each fetch operation.

3 Index Components

3.1 B-Tree Structures

We use a B+ tree as the implementation of the B-tree. Instead of using n as a constraint, we adopted t as the minimum degree of the B+ tree. So every node can have $n = 2t$ values and $2t + 1$ pointers. The formal definition of our B+ tree is presented in Appendix A.

In the code, we created an abstract class `BPlusTreeNode`, with the classes `BPlusTreeLeafNode` and `BPlusTreeInternalNode` inheriting from it. This abstract class can have various abstract functions such as insert, search, delete, and more. The root of the B+ tree is a `BPlusTreeNode`.

3.2 The Bulk Loading Algorithm

The bulk loading algorithm for constructing a B+ tree with a sorted sequence of records requires $\mathcal{O}(n)$ memory and $\mathcal{O}(n)$ disk operations, where n is the number of records. If the record sequence is unsorted, an additional $\mathcal{O}(n \log n)$ memory time is needed for sorting.

We build the B+ tree layer-by-layer recursively. The BULKLOADING function takes in a vector of pointers to records stored on disk, denoted as $\mathcal{R}$. It returns a `BPlusTreeNode` object.

Algorithm 1 BULKLOADING

```
$\mathcal{R} \leftarrow$ input vector
$n \leftarrow$ number of records
SORT $\mathcal{R}$ by its index
$\mathcal{L} \leftarrow$ vector of BPlusTreeLeafNode, means the leaf layer
$\ell \leftarrow \emptyset$ as the new leaf node
for $r$ in $\mathcal{R}$ do
    if $n_\ell \geq t$ and number of records remains $\geq t$ then
        $\ell_{\text{new}} \leftarrow \emptyset$ as a new leaf node
        NEXT($\ell$) $\leftarrow \ell_{\text{new}}$
        Append $\ell$ to $\mathcal{L}$
        $\ell \leftarrow \ell_{\text{new}}$
    end if
end for
return BULKLOADINGUPPERLAYER($\mathcal{L}$)
```

Function BULKLOADINGUPPERLAYER is a helper method for BULKLOADING, which takes in a vector layer of `BPlusTreeNode`, and returns the root of the B+ tree.

Algorithm 2 BULKLOADINGUPPERLAYER

```
$\mathcal{O} \leftarrow$ input vector
$n \leftarrow$ number of nodes
if n = 1 then
    return FIRST($\mathcal{L}$)
end if
$\mathcal{U} \leftarrow$ vector of BPlusTreeInternalNode, means the leaf layer
$u \leftarrow \emptyset$ as the new internal node of the upper layer
for $o$ in $\mathcal{O}$ do
    if $n_\ell \geq t$ and number of nodes remains $\geq t$ then
        Append $u$ to $\mathcal{U}$
        $u \leftarrow \emptyset$ as a new internal node of the upper layer
    end if
end for
return BULKLOADINGUPPERLAYER($\mathcal{U}$)
```

3.3 Range Query on B-Tree

RANGEQUERY is an abstract method for `BPlusTreeNode`. RANGEQUERY$(o, \ell, r, \mathcal{R})$ means that the currently we are in node o, we want to query the records between $[\ell, r]$, and the results should be saved in $\mathcal{R}$. In real coding situations, we use dynamically to implement it. Algorithm 3 shows how the algorithm works.

4 Implementation and Experiments

In this section, we will show our details on our implementations and report the result as required from the 3 tasks.

Algorithm 3 RANGEQUERY$(o, \ell, r, \mathcal{R})$

```
if IsLeaf($o$) then
    for $i \in [0, n)$ do
        if $k_o^{(i)} \geq \ell$ then
            continue
        end if
        if $k_o^{(i)} > r$ then
            return
        end if
    end for
    RangeQuery(Next($o$), $\ell, r, \mathcal{R}$)
else
    BinarySearch to find the first $i$ with $k_o^{(i)} \geq \ell$
    if cannot find $i$ then
        RangeQuery$\left(s_o^{(n_s+1)}, \ell, r, \mathcal{R}\right)$
    else
        RangeQuery$\left(s_o^{(i)}, \ell, r, \mathcal{R}\right)$
    end if
end if
```

4.1 Implementation Details

This project is implemented in pure C++ using minimum version 17. The block size is determined based on the system settings and statistics of the running machine. Therefore, different machines may result in varying block sizes. For simplicity, we use a block size of 4096, which is the default setting on most machines, throughout the rest of the paper unless otherwise specified.

4.2 Task 1

Column Name	**Data Type**	**Row Size** (Bytes)
`GAME_DATE_EST`	`DATE`	10
`TEAM_ID_home`	`VARCHAR(10)`	10
`PTS_home`	`INT`	4
`FG_PCT_home`	`FLOAT32`	4
`FT_PCT_home`	`FLOAT32`	4
`FG3_PCT_home`	`FLOAT32`	4
`AST_home`	`INT`	4
`REB_home`	`INT`	4
`HOME_TEAM_WINS`	`BOOLEAN`	1
Total		45

Table 1: Fields statistics inside a record

File	Num. of lines	26651
Record	Num. of records	26651
	Size (Bytes)	45
Block	Num. of blocks	293
	Size (Bytes)	4096

Table 2: The size of a record; The number of records; The number of records stored in a block; The number of blocks for storing the data

Record In our database system, and in the provided *games.txt* file, a record consists of an array of 45 bytes. A detailed explanation of how we chose the data types and corresponding row sizes for each field is provided in Table 1. Examples of the converted byte arrays for different data types can be found in Figure 1. Since there are 26651 lines in the txt file, there will be 26651 records in the database.

Block Our block size is set to the default value of 4096 bytes. Therefore, each block can store $\left\lfloor \frac{4096}{45} \right\rfloor = 91$ records. To store all the records in our database, we will need $\left\lceil \frac{26651}{91} \right\rceil = 293$ blocks.

Database File A database file in our system is a binary file named with the extension *.db*. It contains multiple blocks used to store the database's data.

4.3 Task 2

Degree t In our B+ tree structure, degree (t) is the minimum number of subtrees of an internal node. In our B+ tree, the degree is 102.

Statistic	Value
Parameter n	203
Total number of nodes	266
Number of levels	3
Content of root node	3 keys from the file

Table 3: the parameter n of the B+ tree; the number of nodes of the B+ tree; the number of levels of the B+ tree; the content of the root node

Parameter n The parameter n is restricted to be an odd integer, which refers to the maximum number of keys that can be stored in a node; and it is equal to two times degree minus 1, which is $2 \times 102 - 1 = 203$.

Number of Nodes There are $26,651$ records. Since each leaf node must store at least 102 keys, the minimum number of leaf nodes is $\lceil \frac{26651}{102} \rceil = 262$, So there will be 262 leaf nodes in this case. Each internal node can have at least 102 pointers (since internal nodes must have at least t=102 children). Therefore, the number of internal nodes at the first level is $\lceil \frac{262}{102} \rceil = 3$. Since there are only 3 internal nodes in the first level, the second level will contain 1 root node (because the root can store at least 101 keys and have up to 102 children). Therefore, totally there are $262 + 3 + 1 = 266$ nodes.

Number of levels in the B+ tree For leaf level there are 262 leaf nodes; at first internal level there are 3 internal nodes; at root Level there are 1 root node. Therefore, the B+ tree has **3 levels**.

Content of the root node (only the keys) The root node will contain the first keys from each of its 3 children (the internal nodes). Since there are only 3 internal nodes, the root will contain 3 keys that guide the search process to the appropriate internal node.

4.4 Task 3

Buffer Size	TIME (μs)	TIME (ms)	IO (Index Block)	IO (Data Block)
1	39774	39	66	6812
100	27891	27	57	4003
250	13469	13	41	838
500	9351	9	0	0

Table 4: The number of IO and time for different buffer sizes using B+ tree. Our indexes are built in a non-clustered order and thus may access a data block multiple times and getting the result

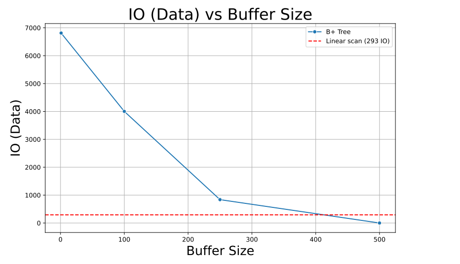
Method	IO (Index)	IO (Data)	Time (ms)
Linear Scan	0	293	22
B+ tree	66	6812	39
B+ tree w. cache	41	838	13

Table 5: Comparison with different methods: Linear Scan, B+ tree, B+ tree with cache. The cache of the B+ tree is set to be 250 pages.

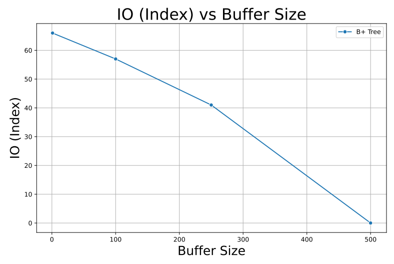
We will now present the statistics required for Task 3. The results were obtained by running the system with different buffer sizes, and the outcomes are summarized in Tables 4 and 5.

Index Node Access Without using a buffer [*], the number of I/O operations for accessing index blocks is 66, which is approximately equal to $\lfloor \frac{6902}{102} \rfloor = 67$, where 6902 is the number of records that satisfy the query conditions. As the buffer size increases, the number of I/O operations for accessing

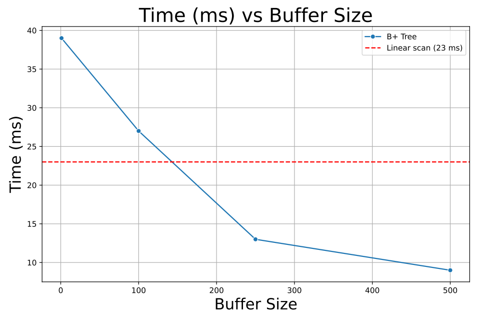
[*]The minimum buffer size for our program is 1 since we always update our buffer after reading one block. However, in this scenario, this buffer would not have any effect. Thus, we consider the case when buffer size is 1 as no buffer



(a) Comparison between different buffer size and data block access for B+ tree



(b) Comparison between different buffer size and index block access for B+ tree



(c) Comparison between different buffer size and runtime for B+ tree and linear scan

Figure 4: Exploring different buffer size

the index decreases gradually, as portions of the index are stored in the buffer during the building phase.

Data Block Access Since we use a dense non-clustered index, many data blocks are accessed multiple times because the records are not stored sequentially. The high I/O in data block access results in the runtime of the B+ tree being longer than that of the range query. This issue can be mitigated by increasing the buffer size, as the linear scan cannot fully utilize the buffer, and the B+ tree will have a shorter runtime with an increased buffer size.

Average of "FG3_PCT_home" for the Records The average "FG3_PCT_home" value retrieved from the returned records is 0.420801. We verified this result using a simple Python program, which returned a value of 0.4208015. Thus, we confirm that the query result is accurate.

Running Time The algorithm's running time without using a buffer is approximately 39 ms. As the buffer size increases, the running time decreases to around 9 ms. Detailed statistics can be found in Table 4.

Data Blocks Accessed by Linear Scan The number of data blocks accessed by a brute-force linear scan is constant at $\lceil \frac{26651}{102} \rceil = 293$.

Running Time of Linear Scan The running time for the linear scan is constant, taking approximately 22 ms. A detail comparison can be found in Figure 4c

5 Conclusion

In this report, we present our design of our storage and index components for the database system in Sections 2 and 3. We then report all of our statistics and experiment results in Section 4 that is required from the manual.

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

A Formal Definition of Our B Plus Tree

Our B+ tree is constructed following the foundational definition of B-trees as given in [1], with key modifications to accommodate the structure and search needs of our data storage and indexing system.

The primary difference between our B+ tree and the one in [1] is that in our implementation is that only the leaf nodes store the actual data records, while internal nodes store only the keys.

A.1 Node Structure

Each node x in the B+ tree has the following properties:

1. **Number of Keys:** n_x, representing the number of keys currently stored in node x. For internal nodes, these keys help guide the search by partitioning the search space.
2. **Keys:** The n_x keys, $k_x^{(1)}, k_x^{(2)}, \cdots, k_x^{(n_x)}$, stored in monotonically increasing order, such that $k_x^{(1)} < k_x^{(2)} < \cdots < k_x^{(n_x)}$,
3. **Leaf Indicator:** ℓ_x, a boolean value that is `true` if x is a leaf and `false` if x is an internal node.
4. **Pointers:** Each node, depending on weather it is an internal or leaf node, contains a specific number of pointers (described in the next sections) that link to either child nodes or data blocks

A.2 Internal Nodes

Each internal node x contains $n_x + 1$ pointers, denoted $s_x^{(1)}, s_x^{(2)}, \cdots, s_x^{(n_x+1)}$, which point to its child nodes. These pointers separate the key space such that each pointer directs the search towards the subtree containing the relevant keys.

- **Pointers as Navigational Aids**: The internal nodes do not store actual data records but only serve as guides for navigating to the correct leaf node or subtree where the data is stored.

- **Key-based Navigation**: The keys in an internal node are used to define search boundaries. If a search for a key *k* lands in node *x*, then the key is passed to the appropriate child pointer based on the relationship:

  $$k_1 < k_x^{(1)} \leq k_2 < k_x^{(2)} \leq \cdots < k_x^{(n_x)} \leq k_{n_x+1}$$

  where $s_x^{(i)}$ leads to the subtree containing all keys in the range defined by the keys in the parent node.

A.3 Leaf Nodes

Each leaf node x contains $n_x + 1$ pointers, denoted $s_x^{(1)}, s_x^{(2)}, \cdots, s_x^{(n_x+1)}$, which are structured differently from the internal node pointers:

1. For $1 \leq i \leq n_x$,each pointer $s_x^{(i)}$ points directly to the data record corresponding to the key $k_x^{(i)}$.
2. The last pointer, $s_x^{(n_x+1)}$ points to the **next leaf node** in the B+ tree's depth-first search (DFS) order. This allows for efficient range queries by following the chain of leaf nodes:
   - If x is the last leaf node, then $s^{(n_x+1)}$ is $\emptyset$, indicating the end of the tree.

A.4 Key Separation in Subtrees

The keys $k_x^{(i)}$ separate the ranges of keys stored in each subtree. For any key subtree, k_i is stored in the subtree with root $s_x^{(i)}$, the following relationship holds:

$$k_1 < k_x^{(1)} \leq k_2 < k_x^{(2)} \leq \cdots < k_x^{(n_x)} \leq k_{n_x+1}$$

This property ensures that a search operation is guided down the correct subtree by comparing the key against the keys in the parent node.

A.5 Leaf Depth

All leaf nodes in the B+ tree are located at the same depth, equal to the height h_T of the tree. This ensures balanced access times for all records, as every search or insertion operation requires traversing the same number of levels to reach a leaf node.

A.6 Key Boundaries and Node Capacity

Each node in the B+ tree has lower and upper bounds on the number of keys it can hold, based on the integer parameter $t \geq 2$:

1. Lower Bound:
   - Any node(except the root) must contain at least $t - 1$ keys.
   - Each internal node, therefore, has at least t children.
   - The root node may contain fewer than $t - 1$ keys but must contain at least one key if the tree is non-empty.
2. Upper Bound:
   - Any node can hold a maximum of $2t - 1$.
   - An internal node can have up to $2t$ children, allowing for balanced growth as new records are inserted.

A.7 Operations on the B+ Tree

- **Search:** To locate a record, the search process navigates from the root to the appropriate leaf node, using the keys in internal nodes to direct the search path. Once the correct leaf node is located, the corresponding data is fetched via the leaf node's pointers.
- **Insertion:** Insertion occurs by first finding the correct leaf node for the key. If the leaf node has space for additional keys, the new key and data record are added. If the leaf is full, it is split, and a new key is promoted to the parent node. If the parent node also becomes full, the splitting and promotion process continues recursively, potentially resulting in the tree growing in height.
- **Range Queries:** Range queries are highly efficient in the B+ tree due to the linked list structure of the leaf nodes. Once the range's starting key is located, the query can traverse the linked list of leaf nodes to retrieve all records within the specified range.