SC3020 Database Systems Principles

Project 1

Cui Nan	U2221495L
Pu Fanyi	U2220175K
Zhang Kaichen	U2123722J
Shan Yi	U2222846C
Tian Yidong	U2220492B

Contents

1. Design of storage component
   - Storage & Data Components
   - Controller Components
   - Buffer Pool
2. Index Components
   - B+-Tree Structure
   - Bulk Loading Algorithm
   - Range Query on B+-Tree
3. Implementation and Experiments
   - Task 1
   - Task 2
   - Task 3

Storage & Data Components

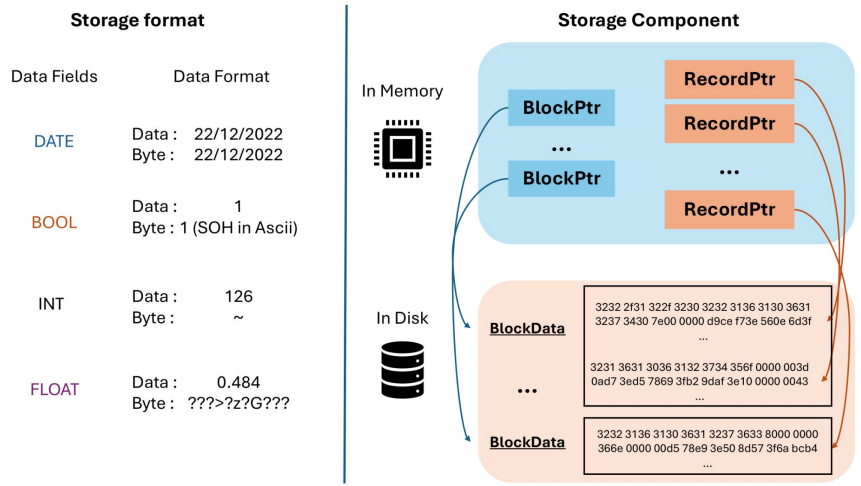


Figure 1: Illustration of the storage components design in our project. All the data are converted into array of Byte and store in the disk. Only pointers that points to the offset position will be stored in memory

BlockData: Stores the actual data as an array of bytes, with size dependent on the block size of the system.

BlockPtr: Points to the position of the BlockData on disk, allowing access to a range of positions within the file stream.

DataPtr: Manages specific positions within BlockData, using BlockPtr to load and store data in real-time operations.

Record: A subclass of DataPtr that accesses data within BlockPtr, handling record-level data management.

Schema: Defines field names and data types, determining the number of bytes required for reading and writing records from byte arrays.

Controller Components

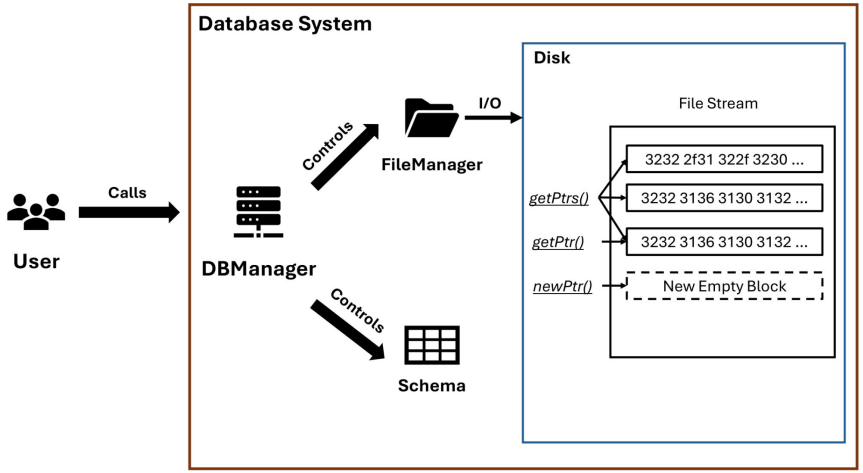


Figure 2: The controller components in how user interactive with the database.

FileManager:

- Manages all **BlockPtr** instances and file streams.
- Handles **read** and **write** operations between files and the buffer pool.

DataBaseManager:

- Controls the database, using the **FileManager** to load data or allocate new memory blocks.
- Oversees reading from the text file or byte array and returns a pointer to the record.
- During user interaction, **DataBaseManager** calls **FileManager** to perform I/O operations for data loading.

Buffer Pool

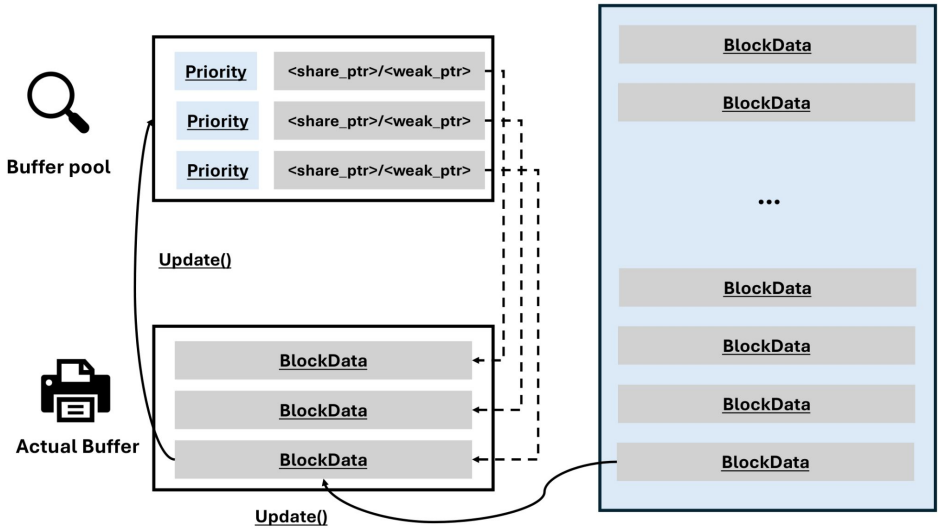


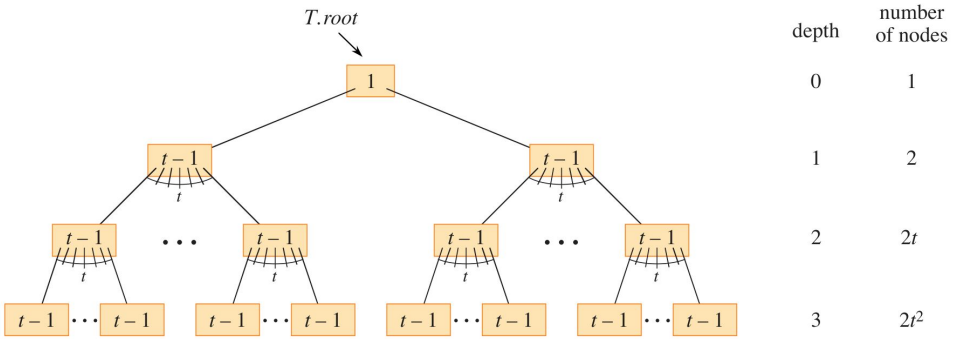
Figure 3: Design of our buffer pool. BlockData are shared with shared_ptr or weak_ptr so that no duplicate BlockData are constructed. During update, the data in the buffer pool will be override and the index and priority will be update

Key Properties of buffer pool:

1. **Hash Table**: Uses the **disk offset** as a key, mapping to the corresponding block data in memory.
2. **LRU List**: Tracks the access order of pages, with the **least recently used** page at the head. It helps implement the page replacement policy when the buffer is full.
3. **Dirty Bit**: Indicates whether a frame has been modified. If **D = 1**, the frame is modified but not written to disk; if **D = 0**, the frame is synchronized with the disk.

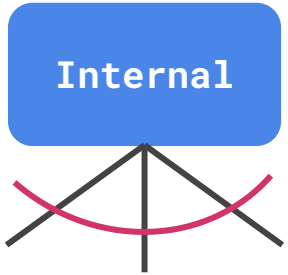
Index Components

- We use B+ tree
- Every nodes (except root) must have at least t pointers
- $n = 2t - 1$
- So n must be odd numbers
- This can reduce a lot of corner cases



depth	number of nodes
0	1
1	2
2	$2t$
3	$2t^2$

Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.



t := minimum degrees
Maximum degrees: $2t$

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

Bulk Loading Algorithm

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 ℓ to $\mathcal{L}$
        $\ell \leftarrow \ell_{\text{new}}$
    end if
end for
return BULKLOADINGUPPERLAYER($\mathcal{L}$)

Create Leaf Nodes

Bulk Loading Algorithm

Algorithm 1 BULKLOADING

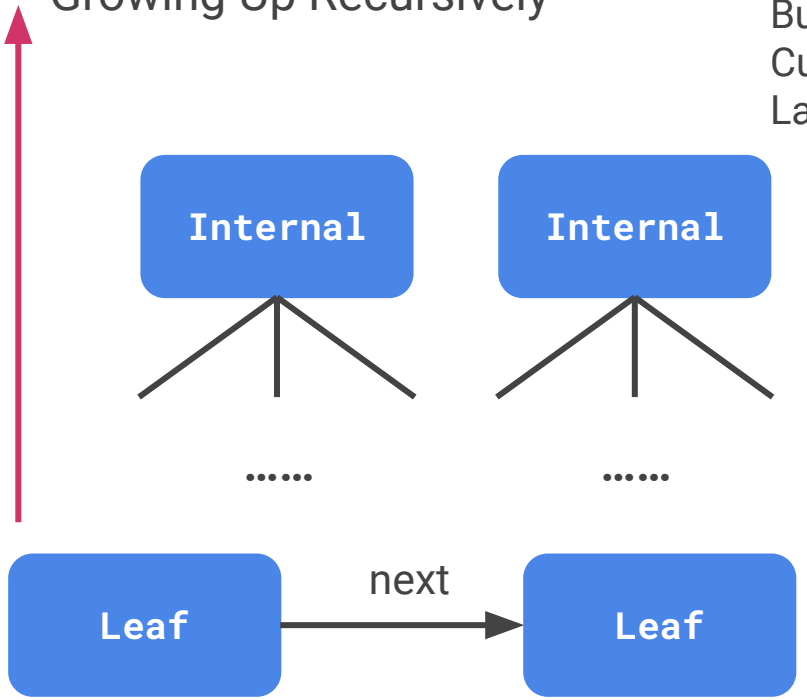
$\mathcal{R} \leftarrow$ input vector
$n \leftarrow$ number of records
SORT $\mathcal{R}$ by its index
$\mathcal{L} \leftarrow$ vector of `BPlusTreeLeafNode`, means the leaf layer
$\ell \leftarrow \emptyset$ as the new leaf node
for r in $\mathcal{R}$ **do**
  if $n_\ell \geq t$ and number of records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

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$(\mathcal{U})$

Bulk Loading Algorithm

Growing Up Recursively



Building Current Layer

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($\mathcal{U}$)

Recursively Growing up

Range Query Algorithm

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$(\text{NEXT}(o), \ell, r, \mathcal{R})$
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

Range Query Algorithm

If the node is an internal node

- We use binary search to find the son
- And query recursively

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

```
if ISLEAF(o) then
    for i ∈ [0, n) do
        if k_o^(i) ≥ ℓ then
            continue
        end if
        if k_o^(i) > r then
            return
        end if
    end for
    RANGEQUERY(NEXT(o), ℓ, r, R)
else
    BINARYSEARCH to find the first i with k_o^(i) ≥ ℓ
    if cannot find i then
        RANGEQUERY(s_o^(n_s+1), ℓ, r, R)
    else
        RANGEQUERY(s_o^(i), ℓ, r, R)
    end if
end if
```

Range Query Algorithm

If the node is a leaf

- We check the indices in the node
- If the index is already out of the right range, we directly return
- And query the right node recursively

If the node is an internal node

- We use binary search to find the son
- And query recursively

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

```
if IsLeaf(o) then
    for i ∈ [0, n) do
        if k_o^(i) ≥ ℓ then
            continue
        end if
        if k_o^(i) > r then
            return
        end if
    end for
    RangeQuery(Next(o), ℓ, r, R)
else
    BinarySearch to find the first i with k_o^(i) ≥ ℓ
    if cannot find i then
        RangeQuery(s_o^(n_s+1), ℓ, r, R)
    else
        RangeQuery(s_o^(i), ℓ, r, R)
    end if
end if
```

Task 1

Record:

- Each record in the database is 45 bytes, consisting of fields
- There are **26,651 records** in total, corresponding to the lines in the provided `games.txt` file.

Block:

- The block size is **4,096 bytes**, allowing each block to store **91 records** (4,096 / 45 = 91).
- A total of **293 blocks** are required to store all 26,651 records.

Database File:

- The data is stored in a binary file with the extension `.db`, which contains multiple blocks holding the records.

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

Statistic for Task 1

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

Task 2

B+-Tree Structure:

- The tree's minimum degree is $t = 102$, meaning each node can hold up to **203 keys** ($2t - 1$).

Parameter n:

- $n = 203$: The maximum number of keys stored in a node (calculated as $2t - 1$).

Statistics:

- **Total Nodes**: The tree consists of **266 nodes**:
  - **262 leaf nodes** (storing the actual data).
  - **3 internal nodes**.
  - **1 root node**.
- **Tree Levels**: The B+ tree has **3 levels**:
  - Leaf level (262 nodes), internal level (3 nodes), root level (1 node).

Root Node:

- The root node contains **3 keys** from its children (internal nodes).

Statistics for Task 2

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

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

Index Node Access

- Without using a buffer for the B+ tree, No. of I/O operations for accessing index blocks is 66.
- As the buffer size increases, the No. of I/O operations for accessing the index decreases gradually.

Data Block Access

- Non-clustered index is used for the data record, without using a buffer for the B+ tree, No. of I/O operations for accessing data blocks is 6812.
- Dense non-clustered index cause multiple block access.
- Larger buffer mitigates this, improving B+ tree performance.

Average of “FG3_PCT_home” for the Records

- 0.420801 (verified with Python)

Comparison Between Linear Scan, B+ Tree, B+ Tree with cache

The cache of the B+ Tree is set to be 250 pages for comparison

Running Time

- Without buffer: ~39 ms.
- With buffer: As the buffer size increases, running time decreases; and reach ~ 9 ms when buffer size=500. Detailed statistics can be found in the Table.

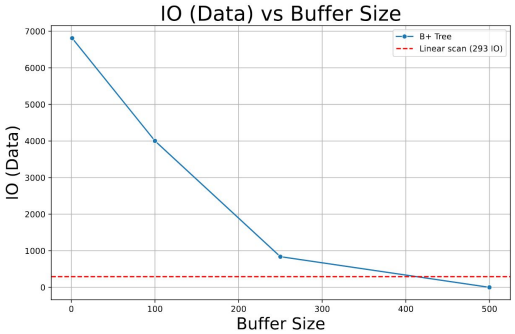
Method	IO (Index)	IO (Data)	Time (ms)
Linear Scan	0	293	22
B+ Tree	66	6812	39
B+ Tree w. cache	41	838	13

Data Blocks Accessed by Linear Scan

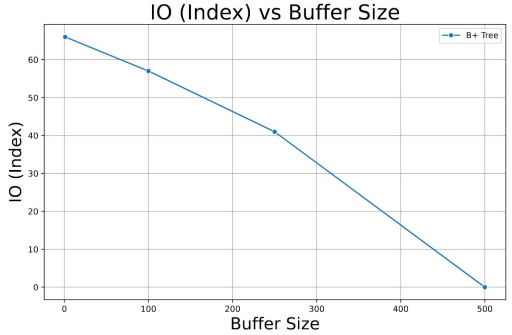
- Data blocks accessed: ~293

Running time of linear Scan

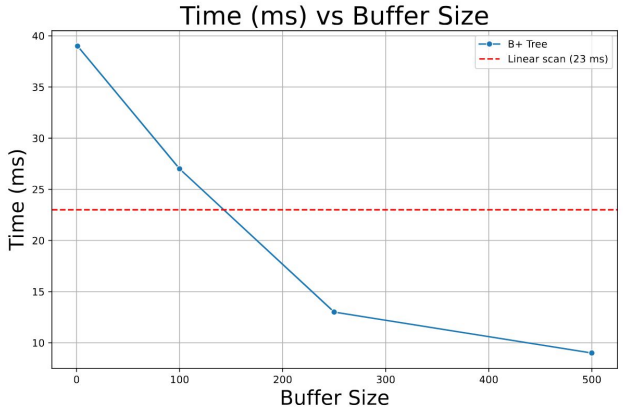
- Constant running time: ~22 ms



(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