Tutorial 2: Records, Blocks and Caching

Implementation of Databases (DBS2)

Arik Ermshaus

Tutorial appointments

Week	Topic
16.10 - 20.10	-
23.10 - 27.10	Organisation, Exercise Sheet 1
30.10 - 03.11	Q&A
06.11 - 10.11	Q&A
13.11 - 17.11	**Exercise Sheet 2**
20.11 - 24.11	Q&A
27.11 - 01.12	Q&A
04.12 - 08.12	Exercise Sheet 3
11.12 - 15.12	Q&A
18.12 - 22.12	Q&A
25.12 - 29.12	-
01.01 - 05.01	-
08.01 - 12.01	Exercise Sheet 4
15.01 - 19.01	Q&A
22.01 - 26.01	Q&A
29.01 - 02.02	Exercise Sheet 5
05.02 - 09.02	Q&A
12.02 - 16.02	Exam preparation

Disclaimer: Timetable is provisional, and will (probably) change!

Table of Contents

Task 1: Hard drives

(a) Calculate the entire storage capacity of the magnetic disk in gigabytes (GB).

- $C_{disk} = \#_{surfaces} \cdot \#_{tracks} \cdot \#_{sectors} \cdot C_{sector}$
- $C_{disk} = 2 \cdot 5 \cdot 10.000 \cdot 1.000 \cdot 1.024B$
- $C_{disk} = 10^8 \cdot 2^{10}B = \frac{2^{10}}{10}GB$
- $C_{disk} = 102{,}4GB$

Property	Value
Sector size	1024 Byte
Sectors per track	1.000
Sectors per block	5
Tracks per surface	10.000
# Platters	5
# Surfaces per Platter	2
Moving head over *n* tracks	(1 + 0,001 * *n*) ms
Rotation Speed	5.000 R/M
Disk Crash Probability	$\frac{1}{10.000}$

Task 1: Hard drives

(b) Determine the number of blocks contained in a single cylinder of the disk.

- $\#_{blocks} = \#_{surfaces} \cdot \frac{\#_{sectors\ per\ track}}{\#_{sectors\ per\ block}}$
- $\#_{blocks} = 2 \cdot 5 \cdot \frac{1.000}{5}$
- $\#_{blocks} = 2.000$

Property	Value
Sector size	1024 Byte
Sectors per track	1.000
Sectors per block	5
Tracks per surface	10.000
# Platters	5
# Surfaces per Platter	2
Moving head over *n* tracks	(1 + 0,001 * *n*) ms
Rotation Speed	5.000 R/M
Disk Crash Probability	$\frac{1}{10.000}$

Task 1: Hard drives

(c) Calculate the average time needed to sequentially read an entire track.

- $t_{seek} = 1 + 0{,}001 \cdot \frac{10.000}{3} ms = \frac{13}{3} ms$
- $t_{rotation} = \frac{1}{2} \cdot \frac{60.000}{5.000} ms = \frac{1}{2} \cdot \frac{60}{5} ms = 6ms$
- $t_{track} = t_{seek} + t_{rotation} + t_{revolution}$
- $t_{track} = \frac{13}{3} ms + 6ms + 12ms = \frac{67}{3} ms$
- $t_{track} \approx 22{,}3ms$

Property	Value
Sector size	1024 Byte
Sectors per track	1.000
Sectors per block	5
Tracks per surface	10.000
# Platters	5
# Surfaces per Platter	2
Moving head over *n* tracks	(1 + 0,001 * *n*) ms
Rotation Speed	5.000 R/M
Disk Crash Probability	$\frac{1}{10.000}$

Task 1: Hard drives

(d) How many disks can you operate, keeping the probability of at least one disk crash per day below 1%?

- $P(\text{No Crash}) = 1 - \frac{1}{10.000}$
- $0.99 < (1 - \frac{1}{10.000})^n \iff \log 0.99 < n \cdot \log(1 - \frac{1}{10.000})$

$$\iff n < \frac{\log 0.99}{\log(1 - \frac{1}{10.000})} \approx 100{,}5$$

- Hence, you can operate at most 100 disks

Property	Value
Sector size	1024 Byte
Sectors per track	1.000
Sectors per block	5
Tracks per surface	10.000
# Platters	5
# Surfaces per Platter	2
Moving head over *n* tracks	(1 + 0,001 * *n*) ms
Rotation Speed	5.000 R/M
Disk Crash Probability	$\frac{1}{10.000}$

Task 2: RAID

(a) Which RAID level would be the most appropriate for this setup?

- RAID 5, because it provides the requested fault tolerance, maximises storage capacity and read/write performance

Property	Value
# Disks	5
Disk Capacity	1 TB
Tolerate Single Disk Failure	✓
Storage Capacity	as much as possible
Read/Write Performance	as much as possible

Task 2: RAID

(b) Calculate the net space of the RAID system.

- $C_{RAID5} = (5 - 1)TB = 4TB$
- We need one disk's worth of space for parity

Property	Value
# Disks	5
Disk Capacity	1 TB
Tolerate Single Disk Failure	✓
Storage Capacity	as much as possible
Read/Write Performance	as much as possible

Task 2: RAID

(c) Assuming each disk can read 100 blocks per second, determine the maximum number of data that can be read from the RAID system per second.

- $\#_{blocks\ per\ second} = 4 \cdot 100$ b/s $= 400$ b/s
- We can read blocks from 4 disks and parity information from 1 disk at the same time

Property	Value
# Disks	5
Disk Capacity	1 TB
Tolerate Single Disk Failure	✓
Storage Capacity	as much as possible
Read/Write Performance	as much as possible

Task 3: External Sorting

Sort a file containing 50 million integers (439 MB) using 50 MB RAM.

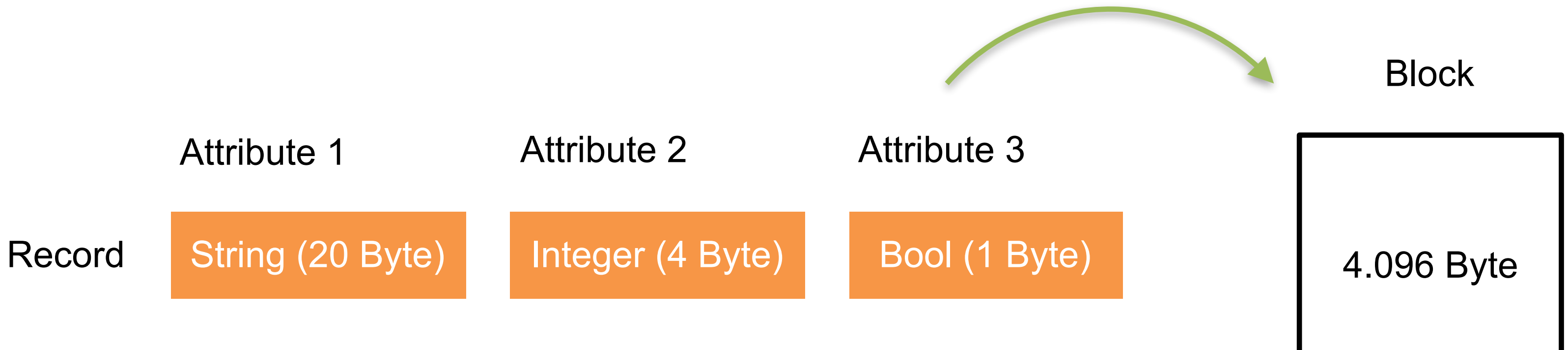
```cpp
int create_sorted_chunks(const std::string &input_file_name, int chunk_size)
{
    std::ifstream input_file(input_file_name, std::ios::in);

    if (!input_file.is_open()) {
        std::cerr << "Unable to open input file" << std::endl;
        return 0;
    }

    int n_chunks = 0;
    std::vector<int> buffer(chunk_size);

    while (input_file) {
        // Read integers into buffer
        int i = 0;
        for (; i < chunk_size && input_file >> buffer[i]; ++i);

        // sort buffer inplace
        std::sort(buffer.begin(), buffer.begin() + i);
        std::ofstream chunk_file("chunk_" + std::to_string(n_chunks) + ".txt");

        // Write integers into chunk
        for (int j = 0; j < i; ++j) {
            chunk_file << buffer[j] << '\n';
        }

        chunk_file.close();
        ++n_chunks;
    }

    input_file.close();
    return n_chunks;
}
```

```cpp
struct PairComparator {
    bool operator()(const std::pair<int, int> &lhs, const std::pair<int, int> &rhs) {
        return lhs.first > rhs.first;
    }
};

void multi_way_merge(const std::string &output_file_name, int n_chunks, int chunk_size) {
    // Organize smallest chunk elements in min-heap
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, PairComparator> min_heap;
    std::vector<std::ifstream> chunk_files(n_chunks);

    for (int i = 0; i < n_chunks; ++i) {
        chunk_files[i].open("chunk_" + std::to_string(i) + ".txt");

        if (!chunk_files[i].is_open()) {
            std::cerr << "Unable to open chunk file " + std::to_string(i) + "\n";
            return;
        }

        // Read first integer from sorted chunk
        int x;
        if (chunk_files[i] >> x) {
            min_heap.push(std::make_pair(x, i));
        }
    }

    std::ofstream output_file(output_file_name);
    while (!min_heap.empty()) {
        // Retrieve currently smallest integer from heap
        auto elem = min_heap.top();
        min_heap.pop();

        // Add element to sorted file
        output_file << elem.first << '\n';

        // Read next smallest element from chunk
        int x;
        if (chunk_files[elem.second] >> x) {
            min_heap.push(std::make_pair(x, elem.second));
        }
    }

    // Close and remove chunks
    for (int i = 0; i < n_chunks; ++i) {
        chunk_files[i].close();
        const std::string chunk_file_name = "chunk_" + std::to_string(i) + ".txt";

        if (std::remove(chunk_file_name.c_str()) != 0) {
            std::cout << "Failed to delete chunk file " << chunk_file_name << std::endl;
        }
    }

    output_file.close();
}

void sort_external_file(const std::string &input_file_name, const std::string &output_file_name)
{
    int chunk_size = 10000000;
    int n_chunks = create_sorted_chunks(input_file_name, chunk_size);
    multi_way_merge(output_file_name, n_chunks, chunk_size);
}
```

Table of Contents

Table of Contents

Recap: Records



Question: How do you represent records on the disk?

- Fundamental elements of data base systems: Records - collections of typed attributes
- Goal: Store records (safely), find them when needed (stable references)
- Challenges: fixed vs. variable-length fields, spanning vs. unspanned records

Example: Fixed-length Record

	Attribute 1	Attribute 2	Attribute 3
Record	String (20 Byte)	Integer (4 Byte)	Bool (1 Byte)

Access field directly

✓ Most space and access efficient

✗ Rigid, no variable-length attributes possible

Property/Type	Fixed			
Space (in Byte)	25			
Access attribute Ai at position *i* (in Byte)	SizeOf(A_i)			

Example: Variable-length Record

Attribute 1 | Attribute 2 | Attribute 3

Record: String (20 Byte) | Tag (1 Byte) | Integer (4 Byte) | Tag (1 Byte) | Bool (1 Byte) | Tag (1 Byte)

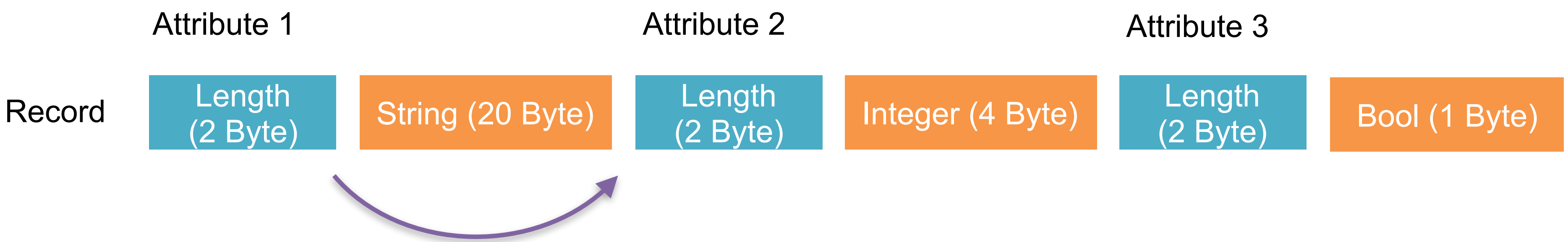
Access field reading from start

Variable-length attributes possible

Lots of useless data access

Property/Type	Fixed	Mark Ends		
Space (in Byte)	25	25 + 3		
Access attribute Ai at position *i* (in Byte)	SizeOf(A_i)	$\sum_{k=1}^{i}$ SizeOf$(A_k) + 1$		

Example: Variable-length Record



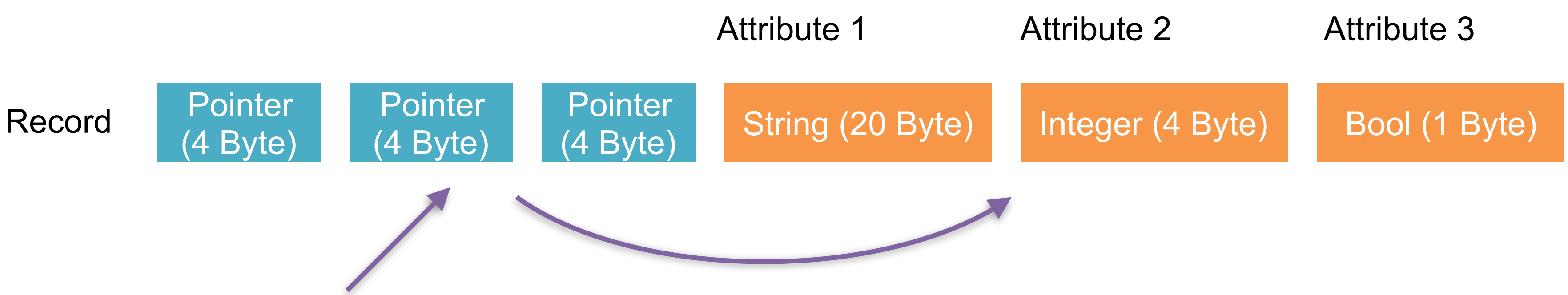
Access field “hopping” from start

✓ Variable-length, less data access

✗ Still traversing useless lengths

Property/Type	Fixed	Mark Ends	Store Lengths	
Space (in Byte)	25	25 + 3	25 + 6	
Access attribute Ai at position *i* (in Byte)	$\text{SizeOf}(A_i)$	$\sum_{k=1}^{i} \text{SizeOf}(A_k) + 1$	$i \cdot 2 + \text{SizeOf}(A_i)$	

Example: Variable-length Record



Access pointer directly, go from there to field

✓ Variable-length, only 2 data accesses ✗ Needs to store and manage dictionary

Property/Type	Fixed	Mark Ends	Store Lengths	Dictionary
Space (in Byte)	25	25 + 3	25 + 6	25 + 12
Access attribute Ai at position *i* (in Byte)	$\text{SizeOf}(A_i)$	$\sum_{k=1}^{i} \text{SizeOf}(A_k) + 1$	$i \cdot 2 + \text{SizeOf}(A_i)$	$(4 + 4) + \text{SizeOf}(A_i)$

Task 1: Records

Property/Type	Fixed	Mark Ends	Store Lengths	Dictionary
Space (in Byte)	25	25 + 3	25 + 6	25 + 12
Access attribute Ai at position *i* (in Byte)	$\text{SizeOf}(A_i)$	$\sum_{k=1}^{i} \text{SizeOf}(A_k) + 1$	$i \cdot 2 + \text{SizeOf}(A_i)$	$(4 + 4) + \text{SizeOf}(A_i)$

- **Question**: Which record structure do you choose if your records consist of 4 required attributes (three integers, 1 boolean)?

(A) Fixed

(B) Mark Ends

(C) Store Lengths

(D) Dictionary

Task 1: Records

Property/Type	Fixed	Mark Ends	Store Lengths	Dictionary
Space (in Byte)	25	25 + 3	25 + 6	25 + 12
Access attribute Ai at position *i* (in Byte)	$\text{SizeOf}(A_i)$	$\sum_{k=1}^{i} \text{SizeOf}(A_k) + 1$	$i \cdot 2 + \text{SizeOf}(A_i)$	$(4 + 4) + \text{SizeOf}(A_i)$

- **Question**: Which record structure do you choose if your records consist of 4 required attributes (three integers, 1 boolean)?

(A) Fixed

(B) Mark Ends

(C) Store Lengths

(D) Dictionary

No variable-length records, direct access,
best space utilisation

Task 2: Records

Property/Type	Fixed	Mark Ends	Store Lengths	Dictionary
Space (in Byte)	25	25 + 3	25 + 6	25 + 12
Access attribute Ai at position *i* (in Byte)	$\text{SizeOf}(A_i)$	$\sum_{k=1}^{i} \text{SizeOf}(A_k) + 1$	$i \cdot 2 + \text{SizeOf}(A_i)$	$(4+4) + \text{SizeOf}(A_i)$

- **Question**: Which record structure do you choose if your records consist of 3 required integer attributes and one optional string?

(A) Fixed

(B) Mark Ends

(C) Store Lengths

(D) Dictionary

Task 2: Records

Property/Type	Fixed	Mark Ends	Store Lengths	Dictionary
Space (in Byte)	25	25 + 3	25 + 6	25 + 12
Access attribute Ai at position *i* (in Byte)	$\text{SizeOf}(A_i)$	$\sum_{k=1}^{i} \text{SizeOf}(A_k) + 1$	$i \cdot 2 + \text{SizeOf}(A_i)$	$(4 + 4) + \text{SizeOf}(A_i)$

- **Question**: Which record structure do you choose if your records consist of 3 required integer attributes and one optional string?

(A) Fixed

(B) Mark Ends

(C) Store Lengths

(D) Dictionary

Variable-length records, fast access with only one indirection

Task 3: Records

Property/Type	Fixed	Mark Ends	Store Lengths	Dictionary
Space (in Byte)	25	25 + 3	25 + 6	25 + 12
Access attribute Ai at position *i* (in Byte)	$\text{SizeOf}(A_i)$	$\sum_{k=1}^{i} \text{SizeOf}(A_k) + 1$	$i \cdot 2 + \text{SizeOf}(A_i)$	$(4+4) + \text{SizeOf}(A_i)$

- **Question**: How large (in bytes) is your expected record if it consists of 3 required 4-byte integer attributes and one optional 256-byte string (occurs with $p = \frac{1}{2}$). You use a "mark ends" record structure.

(A) 140 Byte

(B) 144 Byte

(C) 272 Byte

(D) 145 Byte

Task 3: Records

Property/Type	Fixed	Mark Ends	Store Lengths	Dictionary
Space (in Byte)	25	25 + 3	25 + 6	25 + 12
Access attribute Ai at position *i* (in Byte)	SizeOf(A_i)	$\sum_{k=1}^{i}$ SizeOf(A_k) $+ 1$	$i \cdot 2 +$ SizeOf(A_i)	$(4 + 4) +$ SizeOf(A_i)

- **Question**: How large (in bytes) is your expected record if it consists of 3 required 4-byte integer attributes and one optional 256-byte string (occurs with $p = \frac{1}{2}$). You use a “mark ends” record structure.

(A) 140 Byte

(B) 144 Byte

(C) 272 Byte

(D) 145 Byte

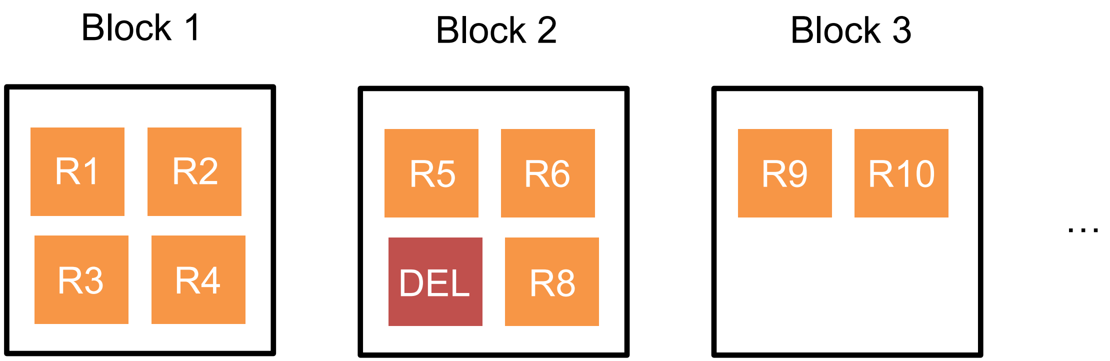
$$E[Record] = 3 \cdot (4 + 1)B + \frac{1}{2} \cdot 256 + 1B$$

Implementation: Variable-length Records

```cpp
Record::Record(std::string const& record_id, std::vector<Attribute> const& attributes)
{
    // Check record_id shape
    if (record_id.size() != Record::RECORD_ID_SIZE) {
        throw std::invalid_argument("record_id must be exactly 10 bytes long.");
    }

    // Calculate total size needed
    // = record size field size + dictionary size + record_id + attribute sizes
    int dictionary_size = (attributes.size() + 2) * sizeof(int);

    int size = sizeof(int) + dictionary_size;
    size += Record::RECORD_ID_SIZE;

    for (const auto& attr : attributes) {
        if (std::holds_alternative<int>(attr)) {
            size += sizeof(int);
        } else if (std::holds_alternative<std::string>(attr)) {
            size += std::get<std::string>(attr).size();
        } else if (std::holds_alternative<bool>(attr)) {
            size += sizeof(bool);
        }
    }

    // Allocate memory block
    char* buffer = new char[size];

    // Offset for copying data
    int dictionary_offset = sizeof(int);
    int offset = sizeof(int) + dictionary_size;

    // Copy size of record
    std::memcpy(buffer, &size, sizeof(int));

    // Copy record ID as first attribute
    std::memcpy(buffer + dictionary_offset, &offset, sizeof(int));
    std::memcpy(buffer + offset, record_id.c_str(), Record::RECORD_ID_SIZE);

    dictionary_offset += sizeof(int);
    offset += Record::RECORD_ID_SIZE;

    // Copy attributes
    for (const auto& attr : attributes) {
        std::memcpy(buffer + dictionary_offset, &offset, sizeof(int));
        dictionary_offset += sizeof(int);

        if (std::holds_alternative<int>(attr)) {
            int val = std::get<int>(attr);
            std::memcpy(buffer + offset, &val, sizeof(int));
            offset += sizeof(int);
        } else if (std::holds_alternative<std::string>(attr)) {
            const std::string& val = std::get<std::string>(attr);
            std::memcpy(buffer + offset, val.c_str(), val.size());
            offset += val.size();
        } else if (std::holds_alternative<bool>(attr)) {
            bool val = std::get<bool>(attr);
            std::memcpy(buffer + offset, &val, sizeof(bool));
            offset += sizeof(bool);
        }
    }

    // Store end offset in dictionary
    std::memcpy(buffer + dictionary_offset, &offset, sizeof(int));

    // Create shared_ptr
    data = std::shared_ptr<void>(buffer, [](void* ptr) { delete[] static_cast<char*>(ptr);
});
}
```

```cpp
std::string Record::get_string_attribute(int attribute_index)
{
    if (data)
    {
        // get attribute start position
        int position = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + sizeof(int) +
attribute_index * sizeof(int));

        // get (next) attribute start position
        int next_position = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + sizeof(int) +
(attribute_index+1) * sizeof(int));

        // retrieve the data
        char *ptr = static_cast<char *>(data.get()) + position;
        return std::string(ptr, ptr + next_position - position);
    }
    return "";
}

int Record::get_integer_attribute(int attribute_index)
{
    if (data)
    {
        // get attribute start position
        int position = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + sizeof(int) +
attribute_index * sizeof(int));
        // retrieve the data
        return *reinterpret_cast<int*>(static_cast<char*>(data.get()) + position);
    }
    return 0;
}

bool Record::get_boolean_attribute(int attribute_index)
{
    if (data)
    {
        // get attribute start position
        int position = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + sizeof(int) +
attribute_index * sizeof(int));
        // retrieve the data
        return *reinterpret_cast<bool*>(static_cast<char*>(data.get()) + position);;
    }
    return false;
}
```

Recap: Blocks

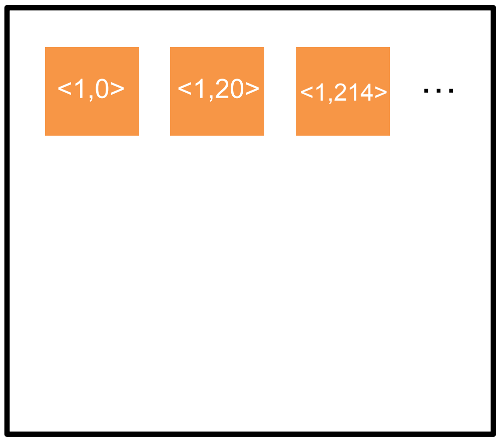


Question: How do you find and manage records on disk?

- Collections of records on disk (organised as physical pages), loaded on demand in cache for data access
- Goal: Make records addressable and easily retrievable from disk (and cache) for data base system
- Challenges: semi-physical referencing, efficient space utilisation vs fast data access

Example: Addressing Records

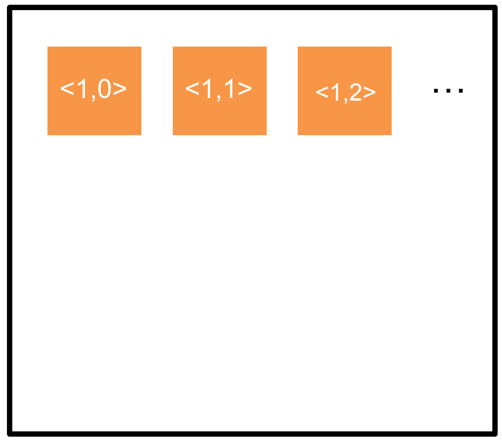
Block 1 (Direct Access)



Record-ID: <Block-ID, Offset>

- ✓ Fast data access
- ✗ Records cannot be reorganised

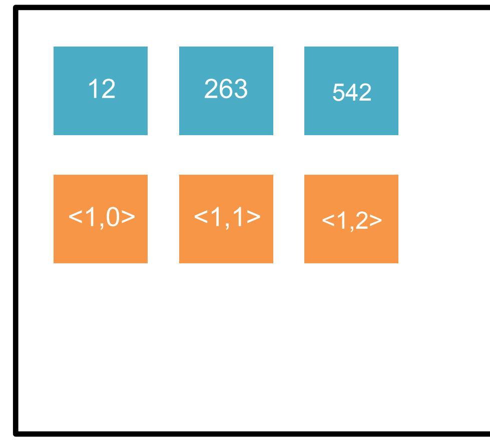
Block 1 (Local Search)



Record-ID: <Block-ID, Local-ID>

- ✓ Records can be reorganised
- ✗ Single record retrieval requires entire block scan

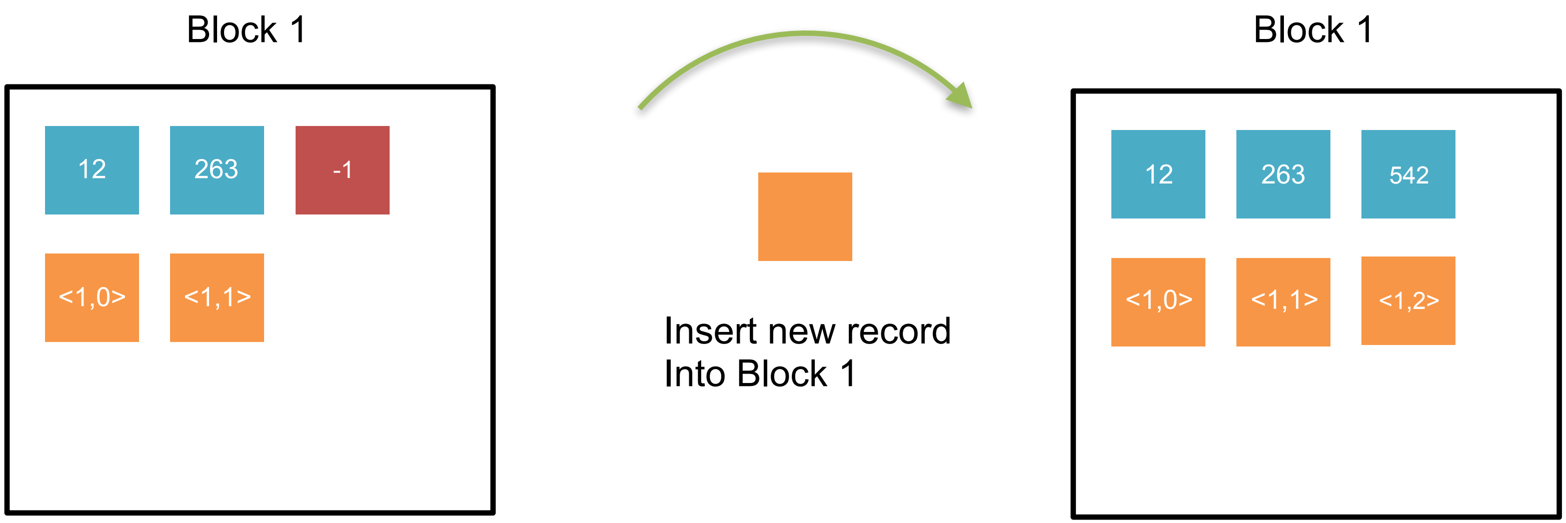
Block 1 (Directory)



Record-ID: <Block-ID, Dir-Offset>

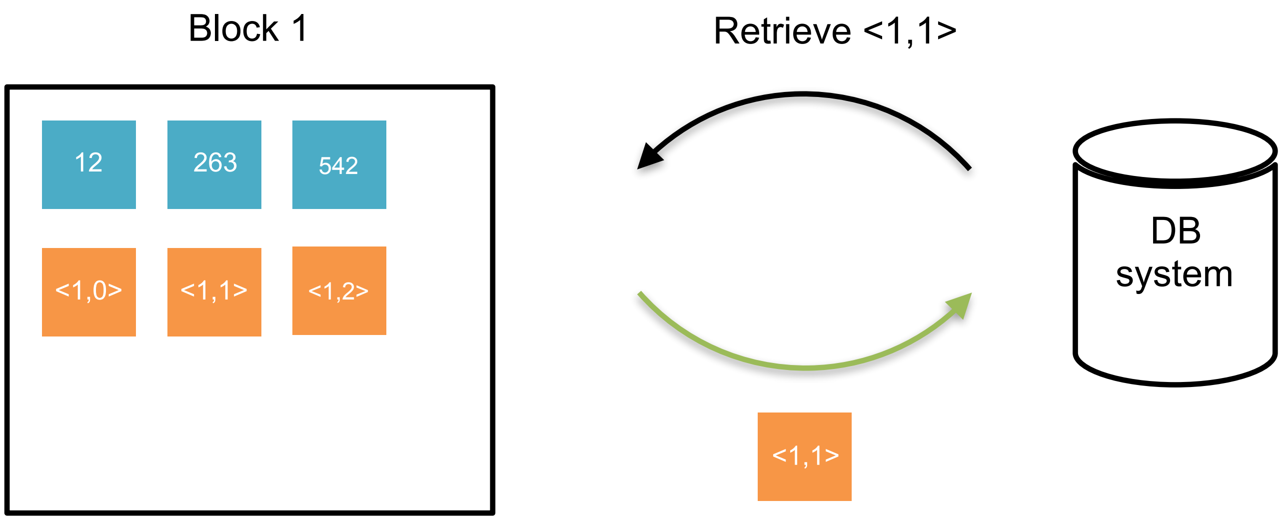
- ✓ Fast data access and reorganisation possible
- ✗ Directory needs space and must be managed

Example: Block-Directory (Create Record)



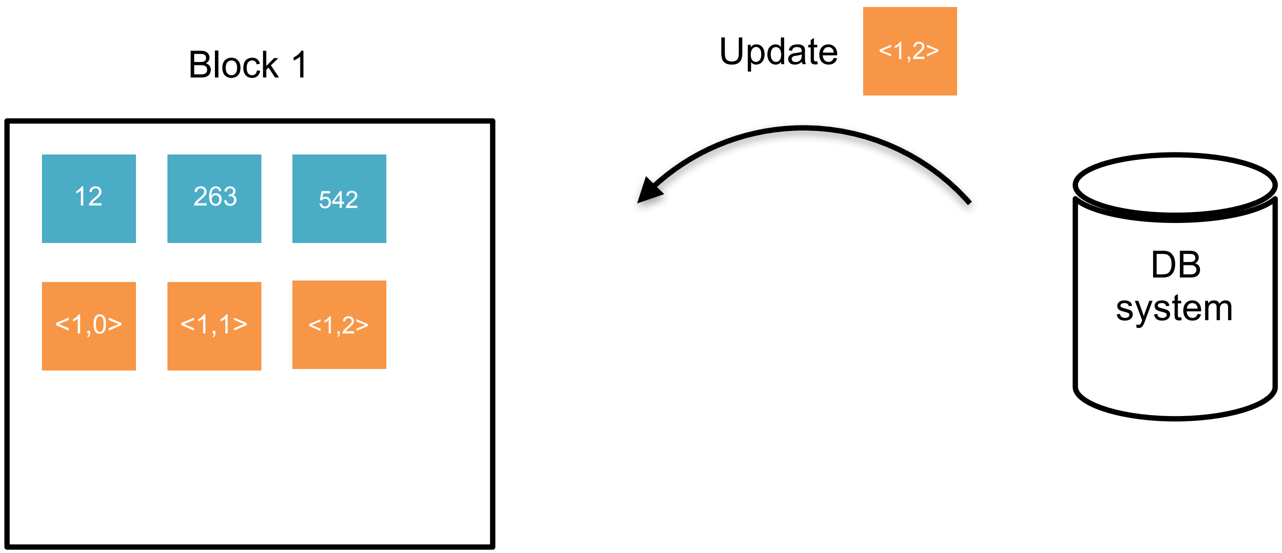
1. Find unused offset in record directory (e.g. offset = 2)
2. Create record-id as <Block-ID, Dir-Offset> (e.g. <1,2>)
3. Find empty space; insert record and update directory

Example: Block-Directory (Read Record)



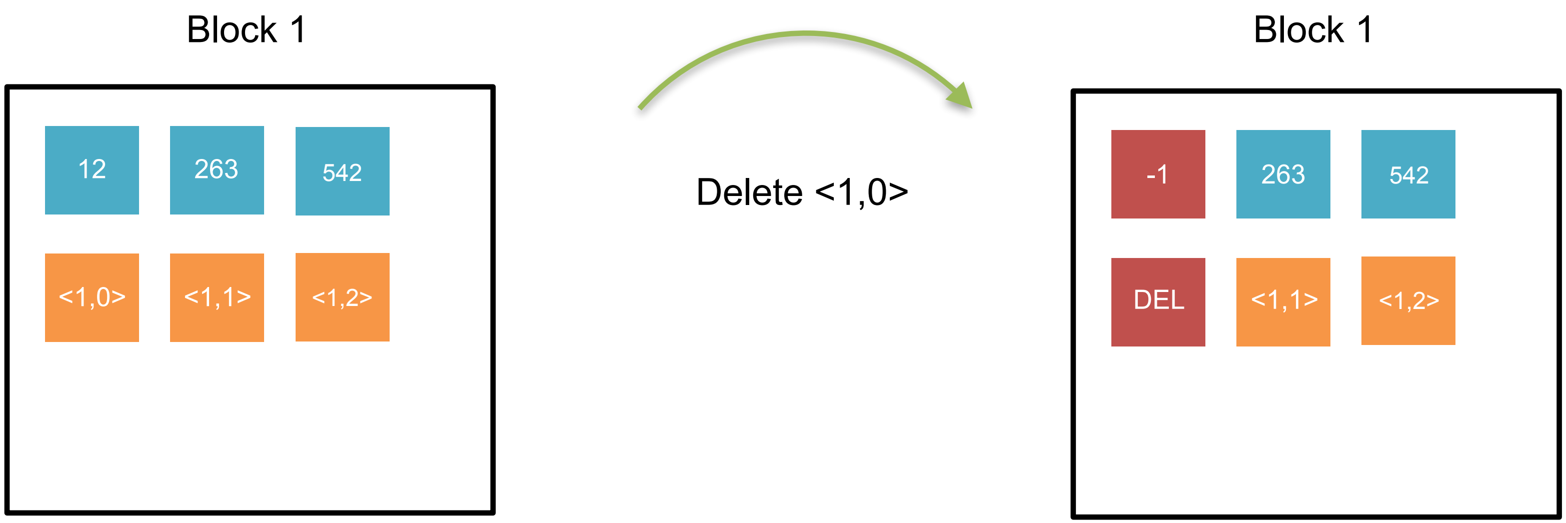
1. Access *i*-th directory offset (e.g. offset = 1)
2. Read record referenced at the offset (e.g. position 263)

Example: Block-Directory (Update Record)



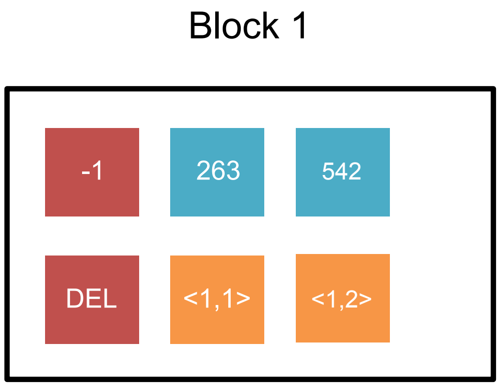
1. Access *i*-th directory offset (e.g. offset = 2)
2. Write updated record referenced at the offset (if enough space present, otherwise delete existing record and insert updated one)

Example: Block-Directory (Delete Record)



1. Find *i*-th offset in record directory (e.g. offset = 0)
2. Mark offset as unused (e.g. offset = -1)
3. Find and overwrite record with tombstone (e.g. DEL)

Task 4: Blocks



Block Size: 4.096 Byte
Max. Directory Entries: 31
Directory Offset Size: 4 Byte
Record Size: 256 Byte

- **Question**: How much net space is available for records in this block configuration?

(A) 3972 Byte

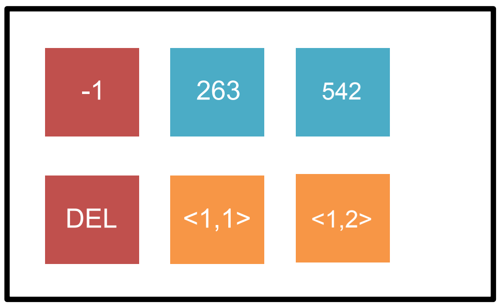
(B) 4096 Byte

(C) 1924 Byte

(D) 2084 Byte

Task 4: Blocks

Block 1



Block Size: 4.096 Byte
Max. Directory Entries: 31
Directory Offset Size: 4 Byte
Record Size: 256 Byte

- **Question**: How much net space is available for records in this block configuration?

(A) 3972 Byte

(B) 4096 Byte

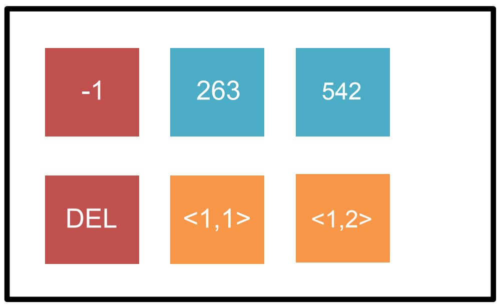
(C) 1924 Byte

(D) 2084 Byte

$C_{block} = 4096 - 4 \cdot 31$

Task 5: Blocks

Block 1



Block Size: 4.096 Byte
Directory Size: 96 Byte
Record Size: 400 Byte

- **Question**: How many blocks do you need to store 100 records with this block configuration?

(A) 5

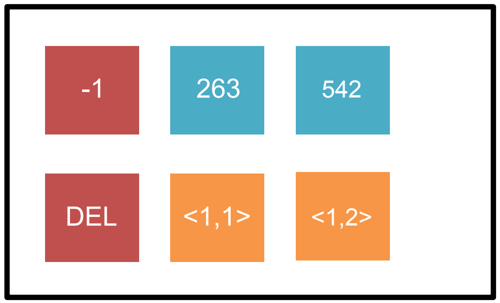
(B) 10

(C) 20

(D) 100

Task 5: Blocks

Block 1



Block Size: 4.096 Byte
Directory Size: 96 Byte
Record Size: 400 Byte

- **Question**: How many blocks do you need to store 100 records with this block configuration?

(A) 5

(B) 10

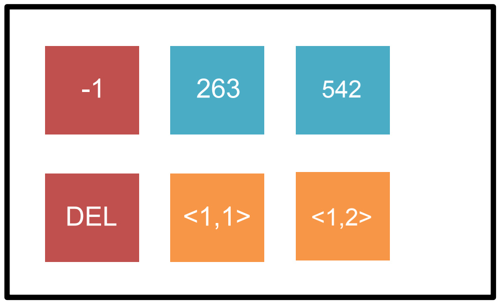
(C) 20

(D) 100

$$\#_{blocks} = \frac{100 \cdot 400B}{(4096B - 96B)}$$

Task 6: Blocks

Block 1



Block Size: 4.096 Byte
Directory Size: 96 Byte
Record Size: 400 Byte

- **Question**: How should you distribute records from tables into blocks?

(A) Random

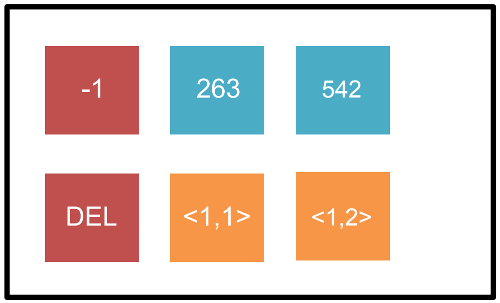
(B) Append last

(C) Group tables

(D) Group associations

Task 6: Blocks

Block 1



Block Size: 4.096 Byte
Directory Size: 96 Byte
Record Size: 400 Byte

- **Question**: How should you distribute records from tables into blocks?

(A) Random

(B) Append last

(C) Group tables

(D) Group associations

Implementation: Block (with directory)

```cpp
Block::Block(std::string const& block_id) {
    // enforce length constraint
    if (block_id.size() != BLOCK_ID_SIZE) {
        throw std::invalid_argument("block_id must be exactly 5 bytes long.");
    }

    // try to read existing data
    data = load_data(block_id);
    dirty = false;

    // reserve data for new block
    if (!data) {
        // Allocate memory block
        char *buffer = new char[Block::BLOCK_SIZE];

        // add block id as first entry
        std::memcpy(buffer, block_id.c_str(), BLOCK_ID_SIZE);

        // set block dictionary to invalid offsets
        for (int i = 0; i < MAX_RECORDS; i++) {
            int val = -1;
            std::memcpy(buffer + BLOCK_ID_SIZE + i * sizeof(int), &val, sizeof(int));
        }

        data = std::shared_ptr<void>(buffer, [](void *ptr) { delete[] static_cast<char *>(ptr); });
        dirty = true;
    }
}
```

Implementation: Block (with directory)

```
std::shared_ptr<void> Block::load_data(std::string const& block_id)
{
    // Create block directory (if needed)
    if (!std::experimental::filesystem::exists(BLOCK_DIR)) {
        if (!std::experimental::filesystem::create_directory(BLOCK_DIR)) {
            std::cerr << "Failed to create block directory." << std::endl;
        }
    }

    // open file handle
    std::ifstream file(BLOCK_DIR + block_id, std::ios::binary | std::ios::in);

    if (!file.is_open())
        return nullptr;

    // Allocate memory and read the file into it
    char* buffer = new char[BLOCK_SIZE];
    file.read(buffer, BLOCK_SIZE);

    // Handle partial read (error)
    if (!file) {
        delete[] buffer;
        file.close();
        return nullptr;
    }

    file.close();
    return std::shared_ptr<void>(buffer, [](void* ptr) { delete[] static_cast<char*>(ptr); });
}
```

Implementation: Block (with directory)

```cpp
std::shared_ptr<Record> Block::add_record(std::vector<Record::Attribute> const &attributes) {
    // retrieve first invalid offset
    int offset_index = -1;

    for (int i = 0; i < MAX_RECORDS; i++) {
        int offset_val = *reinterpret_cast<int *>(static_cast<char *>(data.get()) + BLOCK_ID_SIZE + i * sizeof(int));

        if (offset_val == -1) {
            offset_index = i;
            break;
        }
    }

    // could not find empty space
    if (offset_index == -1)
        return nullptr;

    // find start position to write record
    int offset_val = BLOCK_ID_SIZE + MAX_RECORDS * sizeof(int);

    if (offset_index > 0) {
        // get position from last valid record
        int prev_offset_val = *reinterpret_cast<int *>(static_cast<char *>(data.get()) + BLOCK_ID_SIZE +
                                                       (offset_index - 1) * sizeof(int));

        // read last record id
        int prev_record_offset_val = *reinterpret_cast<int *>(static_cast<char *>(data.get()) + prev_offset_val + sizeof(int));
        char *ptr = static_cast<char *>(data.get()) + prev_offset_val + prev_record_offset_val;
        std::string last_record_id = std::string(ptr, ptr + Record::RECORD_ID_SIZE);

        // set offset_val for new record
        std::shared_ptr<Record> last_record = get_record(last_record_id);
        int last_record_size = last_record->get_size();
        offset_val = prev_offset_val + last_record_size;
    }

    // create new record offset
    std::stringstream ss;
    ss << std::setw(5) << std::setfill('0') << offset_index;
    std::string record_offset = ss.str();

    // create new record
    std::string record_id = get_block_id() + record_offset;
    std::shared_ptr<Record> record = std::make_shared<Record>(Record(record_id, attributes));

    // empty space is not large enough
    if (offset_val + record->get_size() >= BLOCK_SIZE)
        return nullptr;

    // write record to block (and its offset to the directory)
    std::memcpy(static_cast<char *>(data.get()) + BLOCK_ID_SIZE + offset_index * sizeof(int), &offset_val, sizeof(int));
    std::memcpy(static_cast<char *>(data.get()) + offset_val, static_cast<char *>(record->get_data().get()), record->get_size());
    dirty = true;

    return record;
}
```

Implementation: Block (with directory)

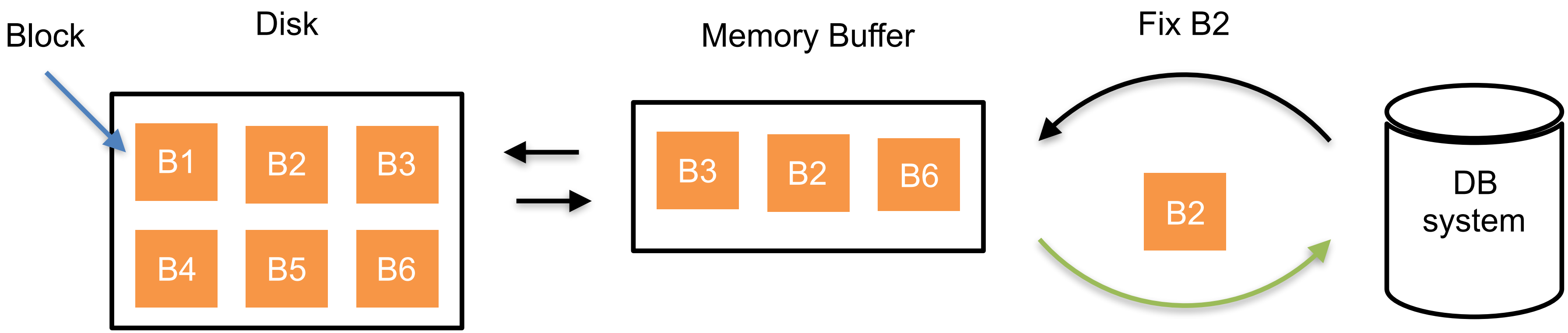
```
std::shared_ptr<Record> Block::get_record(std::string const& record_id)
{
    // enforce length constraint
    if (record_id.size() != Record::RECORD_ID_SIZE) {
        throw std::invalid_argument("record_id must be exactly 10 bytes long.");
    }

    // get directory offset
    std::string block_id = get_block_id(record_id);
    int offset_index = get_block_dictionary_offset(record_id);

    // enforce correct block id
    if (block_id != get_block_id()) {
        return nullptr;
    }

    // get record position in block
    int offset_val = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + BLOCK_ID_SIZE + offset_index * sizeof(int));

    // check for invalid or deleted records
    if (offset_val < 0)
        return nullptr;

    // extract record data
    int record_size = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + offset_val);

    char* buffer = new char[record_size];
    std::memcpy(buffer, static_cast<char *>(data.get()) + offset_val, record_size);

    // create record object
    std::shared_ptr<void> record_data = std::shared_ptr<void>(buffer, [](void* ptr) { delete[] static_cast<char*>(ptr); });
    return std::make_shared<Record>(Record(record_data));
}
```

Implementation: Block (with directory)

```
bool Block::delete_record(std::string const& record_id)
{
    // enforce length constraint
    if (record_id.size() != Record::RECORD_ID_SIZE) {
        throw std::invalid_argument("record_id must be exactly 10 bytes long.");
    }

    // get directory offset
    std::string block_id = get_block_id();
    int offset_index = get_block_dictionary_offset(record_id);

    // enforce correct block id
    if (block_id != get_block_id()) {
        return false;
    }

    // get record position in block
    int offset_val = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + BLOCK_ID_SIZE + offset_index * sizeof(int));

    // check for invalid or deleted records
    if (offset_val < 0)
        return false;

    int record_size = *reinterpret_cast<int*>(static_cast<char*>(data.get()) + offset_val);

    // delete record dictionary entry and data
    int val = -2;
    std::memcpy(static_cast<char *>(data.get()) + BLOCK_ID_SIZE + offset_index * sizeof(int), &val, sizeof(int));

    std::stringstream ss;
    ss << std::setw(record_size) << std::setfill('0');
    std::memcpy(static_cast<char *>(data.get()) + offset_val, ss.str().c_str(), record_size);

    dirty = true;
    return true;
}
```

Table of Contents

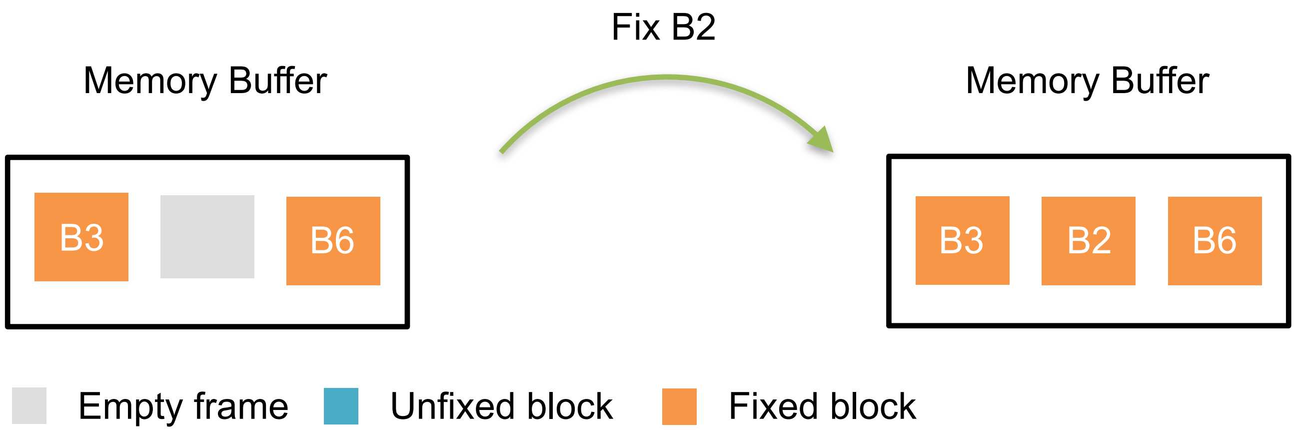
Recap: Buffer Manager



Question: How do you efficiently retrieve blocks at request?

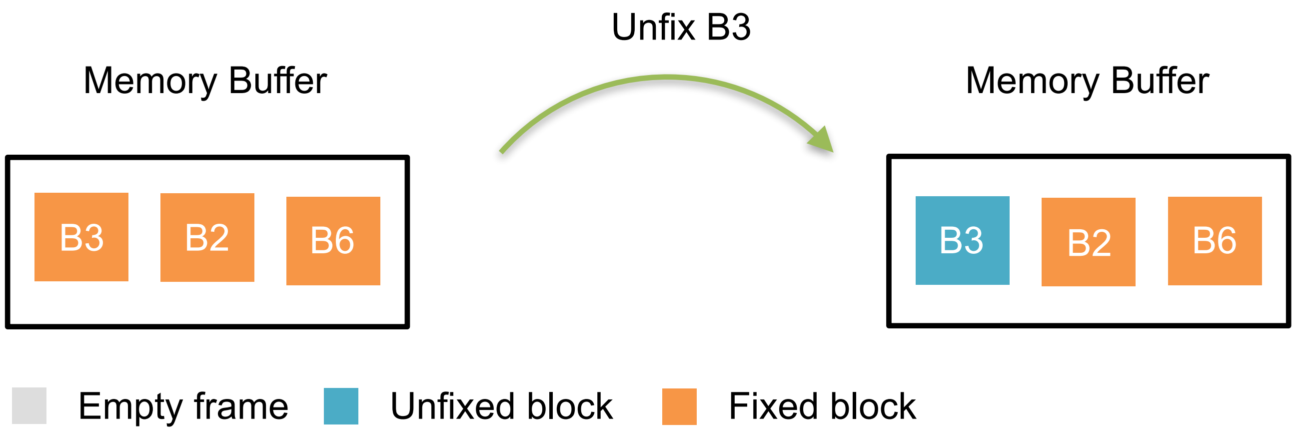
- IO Buffering: keep limited number of blocks in cache for fast access, load (and evict) blocks from (to) disk only when needed
- Fixing: Manage number of references to blocks, only evict ones that are not referenced by the DB system
- Many challenges: cache replacement strategies, prefetching, concurrent transactions, exclusive/shared access

Example: Fixing Blocks



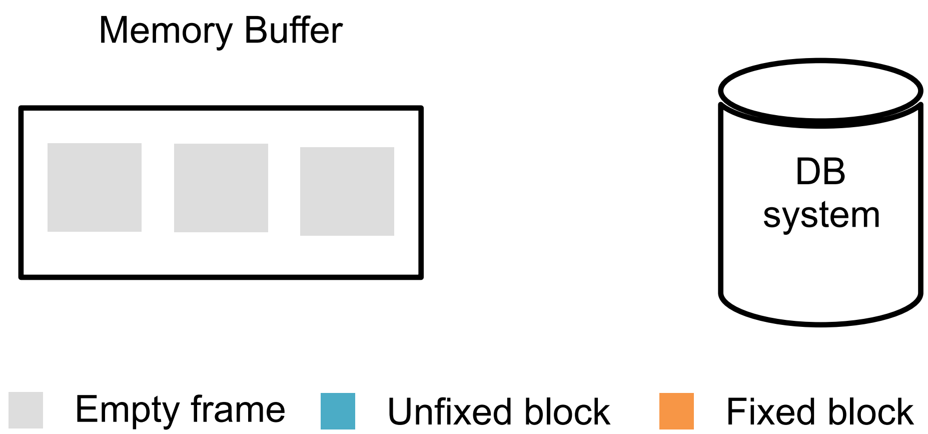
- Fixing a block retrieves it for the DB system at request
- Blocks can be fixed multiple times; each fix is registered
- Fix incurs one of two actions
    - Return requested block from memory buffer
    - Load block from disk to buffer and return it
- If cache is full, an **unfixed** block must first be evicted

Example: Unfixing Blocks



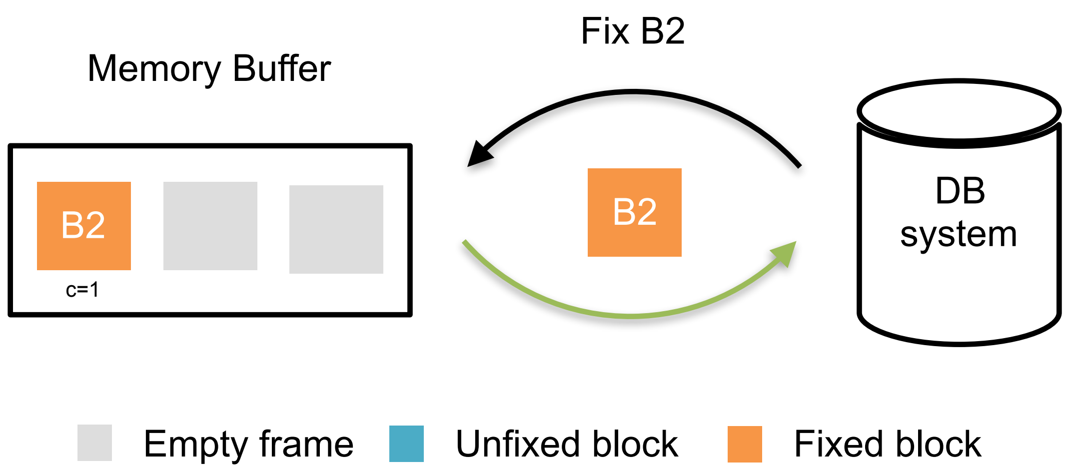
- Unfixing a block “gives it back” from the DB system
- Blocks can still be fixed from other requests
- Unfixed block may now be dirty (changed data)
- Changes are not written to disk directly
  - … only when new a new fix forces to evict the block

Example: Least Recently Unfixed (LRUn)



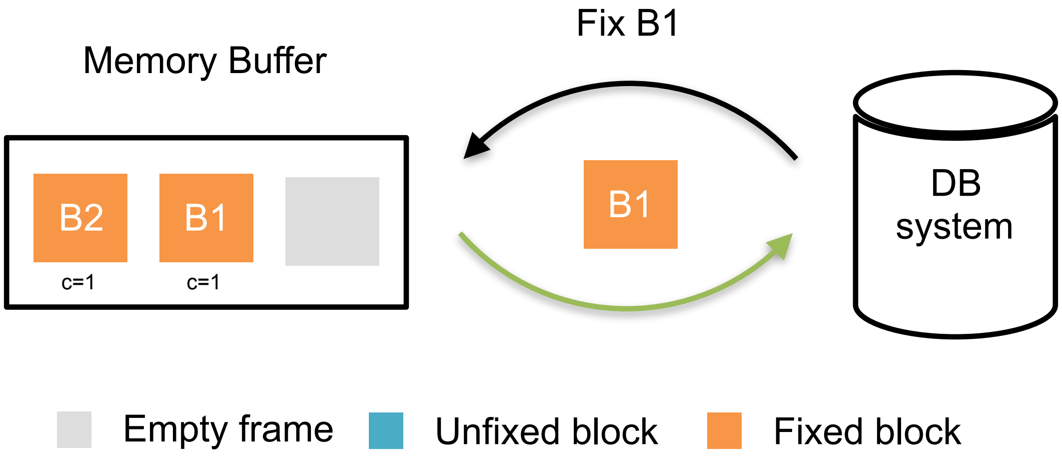
- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
  - Count data structure for fixes, queue for unfixes
  - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



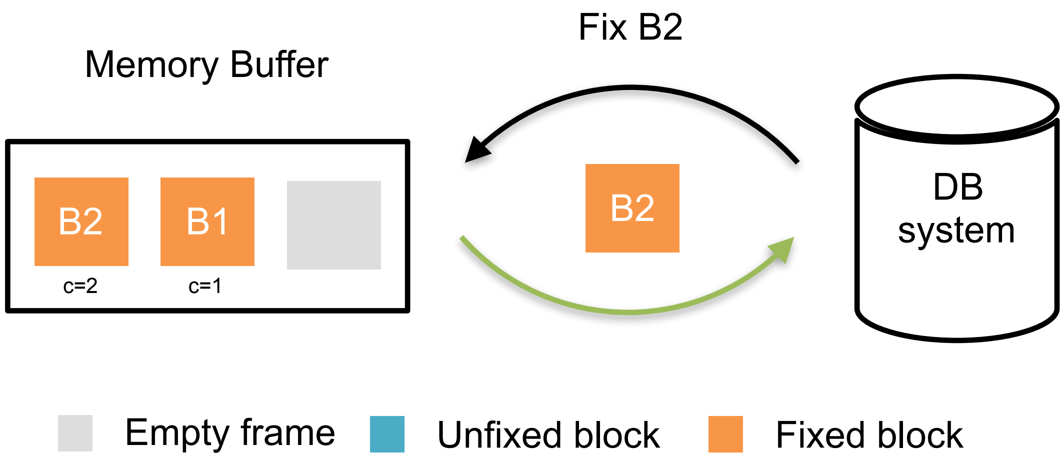
- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
  - Count data structure for fixes, queue for unfixes
  - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



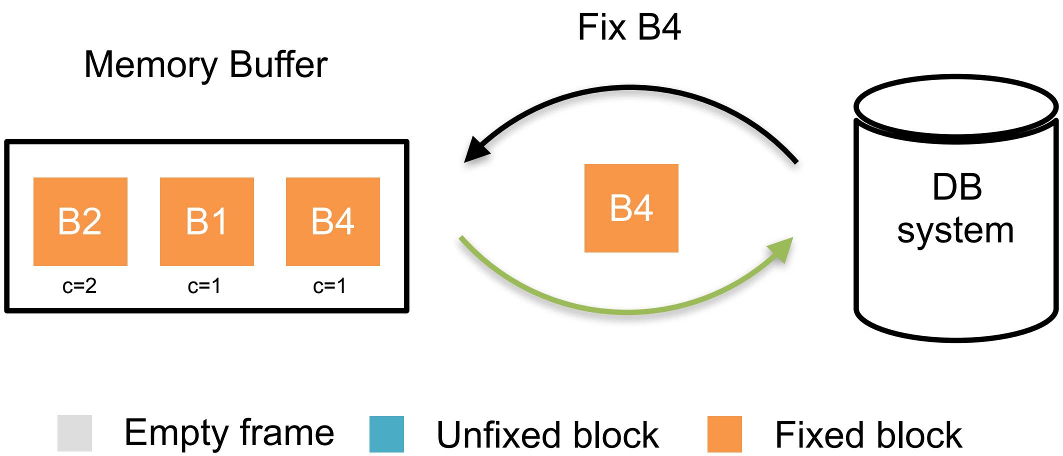
- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
  - Count data structure for fixes, queue for unfixes
  - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



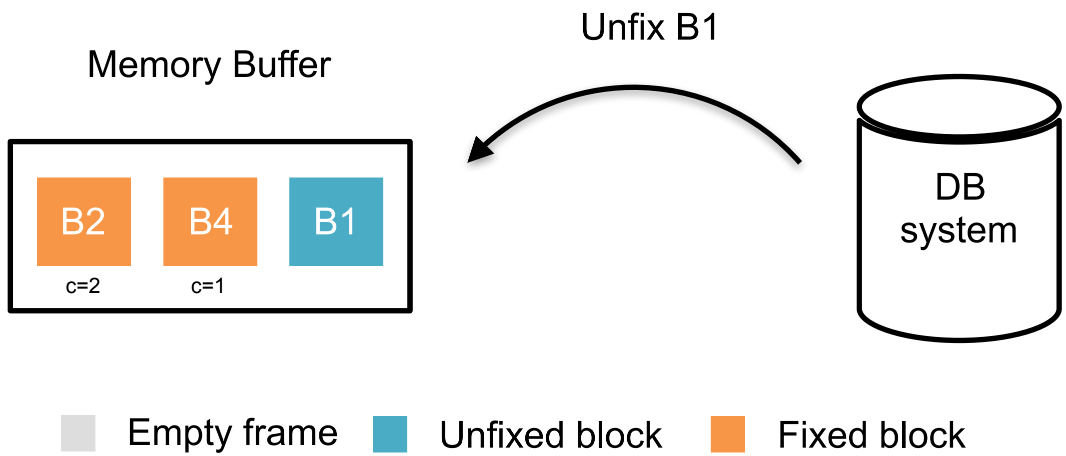
- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
  - Count data structure for fixes, queue for unfixes
  - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



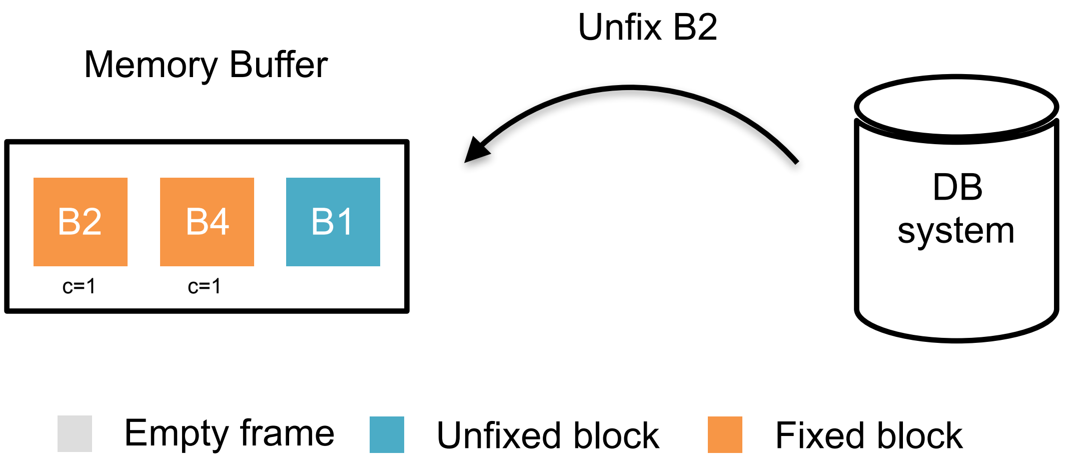
- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
  - Count data structure for fixes, queue for unfixes
  - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



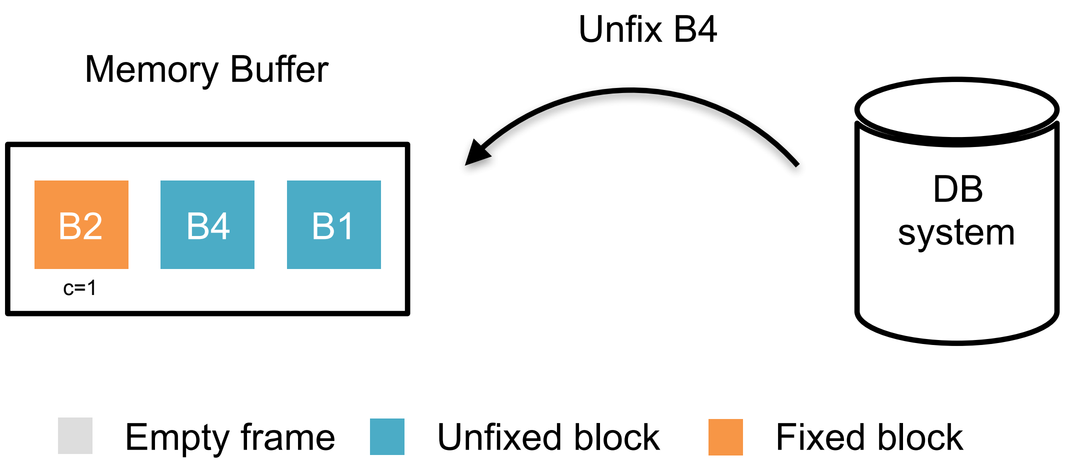
- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
    - Count data structure for fixes, queue for unfixes
    - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



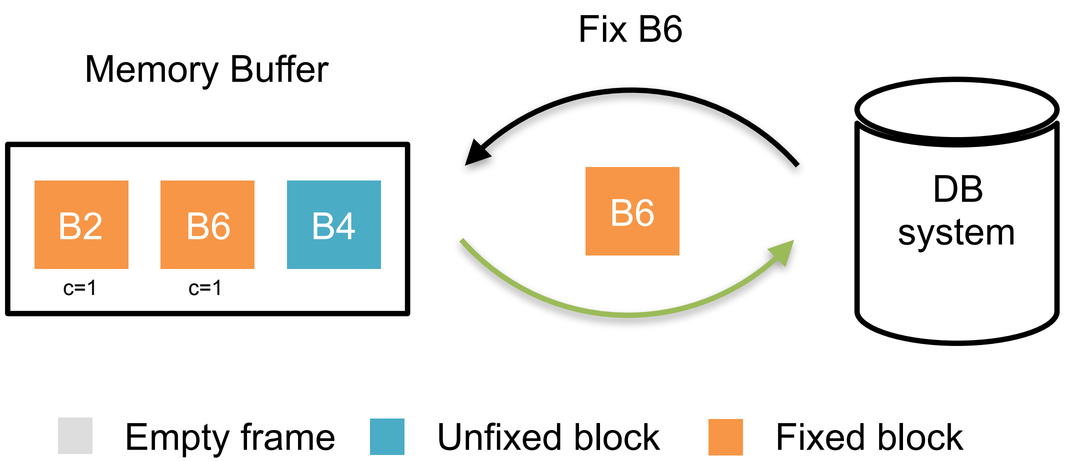
- LRUn Cache replacement: Evict "least" recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
  - Count data structure for fixes, queue for unfixes
  - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
    - Count data structure for fixes, queue for unfixes
    - Hashmap for cache

Example: Least Recently Unfixed (LRUn)



- LRUn Cache replacement: Evict “least” recently **unfixed** block
- Do not evict fixed blocks (still in use)
- Typical implementation:
    - Count data structure for fixes, queue for unfixes
    - Hashmap for cache