УДК 004.6 DOI: 10.31799/2077-5687-2025-5-30-36

РЕФАКТОРИНГ КОДА НА ПРИМЕРЕ РЕАЛИЗАЦИИ ЛОКАЛЬНОЙ БАЗЫ ДАННЫХ, ОСНОВАННОЙ НА B+ ДЕРЕВЬЯХ

А. В. Шахомиров, Н. Е. Шахомирова
Санкт-Петербургский государственный университет аэрокосмического приборостроения

В работе рассмотрена модификация реализации локальной базы данных на основе B+ деревьев – эффективной структуры данных для хранения и быстрого поиска информации. На примерах программного кода показаны выявленные ошибки и недостатки предыдущих реализаций, и предложены варианты их исправления. Показано разбиение монолитного решения на отдельные модули. На примере ключевого модуля реализации ядра B+ дерева показаны варианты модификации функций создания и разделения узлов, показаны варианты борьбы с утечками памяти, показаны варианты решения проблемы некорректного выделения памяти при разделении узла и копировании информации.

Ключевые слова: базы данных, B+ деревья, сбалансированные деревья.

Для цитирования:
Шахомиров, А. В. Рефакторинг кода на примере реализации локальной базы данных, основанной на B+ деревьях / А. В. Шахомиров, Н. Е. Шахомирова // Системный анализ и логистика. – 2025. – № 5(48). – с. 30-36. DOI: 10.31799/2077-5687-2025-5-30-36.

CODE REFACTORING USING THE EXAMPLE OF A LOCAL DATABASE IMPLEMENTATION BASED ON B+ TREES

A. V. Shakhomirov, N. E. Shakhomirova
St. Petersburg State University of Aerospace Instrumentation

The paper examines a modification of the local database implementation based on B+ trees — an efficient data structure for storing and fast retrieval of information. Using code examples, the study identifies errors and shortcomings in previous implementations and proposes solutions for their correction. The decomposition of the monolithic solution into separate modules is demonstrated. Using the core B+ tree module as an example, the paper presents modifications to node creation and splitting functions, addresses memory leak issues, and provides solutions for incorrect memory allocation during node splitting and data copying.

Keywords: databases, B+ trees, balanced trees.

For citation:
Shakhomirov, A. V. Code refactoring using the example of a local database implementation based on B+ trees / A. V. Shakhomirov, N. E. Shakhomirova // System analysis and logistics. – 2025. – № 5(48). – p. 30-36. DOI: 10.31799/2077-5687-2025-5-30-36.

Введение

Рефакторинг кода – это систематическая практика улучшения внутренней структуры программного обеспечения без изменения его внешнего поведения. Основная цель рефакторинга заключается в повышении читаемости, поддерживаемости и расширяемости кода при сохранении функциональности системы. Как отмечают Мартин Фаулер и Кент Бек, рефакторинг позволяет «улучшить дизайн существующей программы, не меняя её внешнего поведения» [1]. Ключевым принципом рефакторинга является его постепенность, последовательность и итеративность: изменения вносятся малыми шагами с регулярным тестированием, для минимизации риска поломки системы [1, 2].

Ожидаемый результат рефакторинга:

1) Улучшение модульности программного обеспечения, которое получается снижением межмодульной связности (*low coupling*), то есть минимизацией зависимостей между компонентами, чтобы изменения в одном модуле меньше влияли на другие [1]; и одновременным повышением внутренней связности (*high cohesion*), то есть группировкой логически связанных элементов внутри одного модуля, чтобы каждый модуль решал чётко очерченный круг задач [1].

2) Упрощение тестирования, которое получается путём изоляции компонентов, то есть четкого разделения функционала, чтобы каждый компонент можно было тестировать независимо от других [2].
3) Прозрачность бизнес-логики и допущений, которые обеспечиваются «чистым» кодом, то есть использованием понятных имён переменных, функций и классов; согласованным соглашениями о единых правилах именования, структурирования и обработки ошибок, то есть единого стиля оформления кода [1, 2].

В ходе модификации и доработки программного обеспечения [3], реализующего построение локальной базы данных, основанной на структуре B+ дерева [4, 5], были выявлены некоторые недостатки предыдущей реализации, которые будем устранять в текущей работе.

Разделение на логические модули

Разделение программного монолита на логические модули представляет собой архитектурную стратегию, направленную на совершенствование структурной организации, сопровождаемости и масштабируемости программной системы при сохранении простоты процесса разработки и развёртывания [2].

В результате анализа кода было принято решение разделить код на логические функциональные модули. Структура каталогов и файлов проекта получилась следующая:

```
/bplus_tree
        /bplus_tree.c       ядро B+ дерева
        /bplus_tree.h
/cache
        /cache.c            работа с кэшем
        /cache.h
/compression
        /compression.c      сжатие данных
        /compression.h
/database
        /database.c         работа с базой данных
        /database.h
/sql_parser
        /sql_parser.c       парсинг SQL
        /sql_parser.h
/transactions
        /transactions.c     управление транзакциями
        /transactions.h
/utils
        /utils.c            вспомогательные функции
        /utils.h
```

В работе рассматривается B+ дерево, имеющее следующую **структуру узла**:

```
// Тип узла: лист или внутренний узел
typedef enum {
    NODE_LEAF,      // Листовой узел - содержит ключи и значения
    NODE_INTERNAL   // Внутренний узел - содержит ключи и указатели на дочерние узлы
} NodeType;

// Структура узла B+ дерева
typedef struct BPlusNode {
    int keys[MAX_KEYS];       // Массив ключей узла
```

```
    void* values[MAX_KEYS];   // Массив значений (только для листьев)
    struct BPlusNode* children[MAX_CHILDREN]; // Массив указателей на дочерние узлы
    struct BPlusNode* next;   // Указатель на следующий лист (для листовых узлов)
    struct BPlusNode* parent; // Указатель на родительский узел
    int key_count;            // Текущее количество ключей в узле
    NodeType type;            // Тип узла (лист или внутренний)
} BPlusNode;

// Структура B+ дерева
typedef struct {
    BPlusNode* root;          // Указатель на корневой узел дерева
    int order;                // Порядок дерева (максимальное количество ключей в узле)
    pthread_rwlock_t rwlock;  // Read-write lock для потокобезопасности
    int node_count;           // Счетчик узлов для статистики
    int leaf_count;           // Счетчик листьев для статистики
} BPlusTree;
```

Создание узлов

Анализ кода предыдущей реализации [3] показал, что при создании узлов происходит выделение памяти без учёта размера массивов. При этом может произойти дополнительное выравнивание размерности, так как операционная система округлит размер запрашиваемой в функции `malloc` памяти до удобного для CPU значения (выравнивание в 8 или 16 байт). Если в уже выделенной памяти будет найден подходящий свободный блок с непрерывной структурой адресации, то именно он и будет использован. Если в ранее использовавшейся, уже освобожденной, но еще не вернувшейся операционной системе, памяти будет найден подходящий блок, то будет использоваться механизм повторного выделения памяти `realloc`.

```
BPlusNode* node = malloc(sizeof(BPlusNode));
```

Так как память может быть не выделена, то необходимо модифицировать код, и добавить проверку:

```
// Проверка успешности выделения памяти
if (!node) return NULL;  // Возврат NULL при ошибке malloc
```

Разделение узлов

Предыдущая реализация [3] функции разделения узла `split_node` не имела проверки входных параметров `parent`, `*parent` и `child` на `NULL`, что могло привести к ошибке `segmentation fault` (разыменование пустого (`NULL`) указателя) [6].

Один из способов исправить это, заключается в том, чтобы проверить указатели на нулевое значение перед разыменование. То есть необходимо добавить проверку

```
if (!parent || !(*parent) || !child)...
```

для предотвращения обращения к `NULL`, и возвращать `false` в случае ошибки выделения памяти.

Была выявлена ошибка в формуле расчета количества ключей в новом узле при разделении текущего узла.

Было:

```
// Определяем середину
int mid = child->key_count / 2;
// Перемещаем половину ключей в новый узел
new_node->key_count = child->key_count - mid - 1;
```

Стало:

```
// Определение середины (округляем вверх для равномерного распределения)
int mid = (child->key_count + 1) / 2;
// При делении узла пополам новый узел должен получить:
new_node->key_count = child->key_count – mid; //ключей.
```

В цикле перебора дочерних узлов могла возникнуть ошибочная очистка указателей `children` во время выполнения, из-за некорректного значения верхней границы цикла:

```
for (int i = mid + 1; i <= child->key_count; i++)
      child->children[i] = NULL; // Возможен выход за границы массива!
```

Решение заключается в смещении нижней и верхней границ цикла перебора узлов относительно среднего узла:

```
// Обнуление перемещённых указателей в исходном узле
for (int i = mid; i < child->key_count + 1; i++)
      child->children[i] = NULL;
```

Также было выявлено некорректное смещение дочерних узлов родителя в цикле вставки нового узла в родительский:

```
for (int i = (*parent)->key_count; i > idx + 1; i--)
```

В этом случае начальное значение `i` должно быть `(*parent)->key_count + 1`.

Корректировка выделения памяти и копирования информации

Следующая проблема, которая была рассмотрена при рефакторинге кода – это возможные утечки памяти при создании и освобождении ресурсов. К сожалению, эта проблема касается библиотечных функций `memcpy` и `memset`, поэтому предлагаемое решение будет реализовываться одинаков во всех функциях проекта, где встречаются вызовы этих функций. В описании этих функций указано: «Security Note. Make sure that the destination buffer has enough room for at least *count* characters. For more information, see “Avoiding buffer overruns”» [7]. То есть необходимо контролировать буфер назначения, и проверять достаточно ли выделено памяти под него.

В предыдущих версиях кода были вызовы функции `memcpy`, в которых при копировании ключей используется значение `new_node->key_count`, но не учитывается, что количество копируемых значений также зависит и от величины количества ключей для переноса в новый узел. То есть объем выделяемой памяти был больше, чем требуется для операции копирования, особенно с учетом граничного выравнивания, кратного 16 байт.

```
memcpy(new_node->keys, &child->keys[mid + 1], new_node->key_count * sizeof(int));
```

Похожая ситуация была при копировании дочерних узлов:

```
memcpy(new_node->children, &child->children[mid + 1], (new_node->key_count + 1) * sizeof(BPlusNode*));
```

Для решения проблемы с `memcpy`, выполнили модификацию кода, в которой количество ключей для переноса в новый узел зависит от ранее вычисленной средней вершины:

```
// Количество ключей для переноса в новый узел
int keys_to_move = child->key_count - mid;
new_node->key_count = keys_to_move;
// Копирование ключей в новый узел (с учётом размера)
memcpy(
    new_node->keys,    // Целевой массив
    &child->keys[mid], // Исходный массив (начиная с mid)
    keys_to_move * sizeof(int)  // Размер копируемых данных
);
// Для внутренних узлов копируем дочерние указатели
if (child->type == NODE_INTERNAL) {
    // Количество дочерних узлов для переноса: keys_to_move + 1
    int children_to_move = keys_to_move + 1;
    memcpy(
        new_node->children,    // Целевой массив
        &child->children[mid], // Исходный массив
        children_to_move * sizeof(BPlusNode*) // Размер
    );
```

Утечки памяти

Следующая часть работы посвящена рассмотрению возможных утечек памяти при удалении узлов и всего дерева.

Было выявлено неполное освобождение памяти в функции `free_bplus_tree()`, освобождение памяти проводилось только для самого узла `node`, но не освобождались массивы `keys`, `values` и `children`. Что приводит к одной из самых распространенных ошибок работы с памятью, связанной с появлением «висячих» указателей (*dangling pointer*). Эти указатели ссылаются на область памяти, которая была освобождена и/или более не действительна. Висячие указатели могут вызвать критические ошибки выполнения, такие как: повреждение данных, утечки памяти, уязвимости безопасности [6, 7].

Поэтому, требуется провести модификацию кода, и добавить операции принудительного обнуления указателей на массивы и структуры.

```
// Массив ключей
if (node->keys) {
    free(node->keys);
    node->keys = NULL;  // Обнуление
}
// Массив значений
if (node->values) {
    free(node->values);
    node->values = NULL;
}
// Массив дочерних указателей
if (node->children) {
    free(node->children);
    node->children = NULL;
}
// Освобождение самого узла
free(node);
```

Итог

Результирующий код для функции разделения узла B+ дерева, с учетом внесенных изменений программного кода, теперь выглядит так:

```
/**
 * Разделение переполненного узла B+ дерева
 * @param parent Указатель на указатель родительского узла (может быть NULL для корня)
 * @param idx Индекс дочернего узла в родительском массиве (если parent != NULL)
 * @param child Узел, который нужно разделить
 * @return true при успешном разделении, false при ошибке
 */
static bool split_node(BPlusNode** parent, int idx, BPlusNode* child) {
    // Создание нового узла того же типа
    BPlusNode* new_node = create_node(child->type, child->order);
    if (!new_node) return false;  // Ошибка выделения памяти
    // Определение середины (округляем вверх для равномерного распределения)
    int mid = (child->key_count + 1) / 2;
    // Вычисляем количество ключей для переноса в новый узел
    int keys_to_move = child->key_count - mid;
    new_node->key_count = keys_to_move;
    // Копируем ключи в новый узел (с учётом размера)
    memcpy(
        new_node->keys,              // Целевой массив
        &child->keys[mid],           // Исходный массив (начиная с mid)
        keys_to_move * sizeof(int)); // Размер копируемых данных
    // Для внутренних узлов копируем дочерние указатели
    if (child->type == NODE_INTERNAL) {
        // Количество дочерних узлов для переноса: keys_to_move + 1
        int children_to_move = keys_to_move + 1;
```

```
        memcpy(
            new_node->children,                    // Целевой массив
            &child->children[mid],                 // Исходный массив
            children_to_move * sizeof(BPlusNode*)); // Размер
        // Обнуляем перемещённые указатели в исходном узле
        for (int i = mid; i < child->key_count + 1; i++)
            child->children[i] = NULL;
    }
    // Обновляем количество ключей в исходном узле
    child->key_count = mid - 1;  // Оставшиеся ключи
    // Если есть родитель, вставляем новый узел
    if (*parent) {
        // Вставляем разделительный ключ (последний ключ исходного узла)
        (*parent)->keys[idx] = child->keys[mid - 1];
        // Сдвигаем дочерние указатели вправо для вставки нового узла
        for (int i = (*parent)->key_count; i > idx; i--)
            (*parent)->children[i + 1] = (*parent)->children[i];
        // Вставляем новый узел в родительский массив
        (*parent)->children[idx + 1] = new_node;
        (*parent)->key_count++;  // Увеличиваем количество ключей родителя
    } else {
        // Если нет родителя — создаём новый корень
        *parent = create_node(NODE_INTERNAL, child->order);
        if (!*parent) {
            free_bplus_tree(new_node);  // Освобождаем новый узел при ошибке
            return false;
        }
        // Настраиваем новый корень
        (*parent)->keys[0] = child->keys[mid - 1]; // Разделительный ключ
        (*parent)->children[0] = child;            // Левый дочерний узел
        (*parent)->children[1] = new_node;         // Правый дочерний узел
        (*parent)->key_count = 1;                  // Один ключ в корне
    }
    // Обновляем связность листьев (если это листья)
    if (child->type == NODE_LEAF) {
        new_node->next = child->next;
        child->next = new_node;
    }
    return true;
}
```

Заключение

В работе представлен рефакторинг функций локальной базы данных на основе B+ деревьев. Показаны выявленные ошибки и недостатки предыдущих реализаций. Основные изменения, которые были внесены в реализацию bplus_tree: Разделение проекта на логические модули; Разделение функционала внутри модуля; Исправление критических ошибок в логике разделения узлов; Добавление проверки родительских и дочерних указателей на пустые значения; Исправление обработки ошибок выделения памяти; Исправление ошибки неполного освобождение памяти и проблемы «висячих» указателей.

СПИСОК ЛИТЕРАТУРЫ

1. Фаулер М. Рефакторинг: улучшение существующего кода. / М. Фаулер, пер. с англ. – СПб: Символ-Плюс, 2003. – 432 с.
2. Мартин Р. Чистый код: создание, анализ и рефакторинг. Библиотека программиста / Р. Мартин. – СПб.: Питер, 2013. – 464 с.
3. Шахомиров А.В. Пример реализации локальной базы данных, основанной на B+

деревьях / А. В. Шахомиров, Н. Е. Шахомирова // Системный анализ и логистика. – 2025. – № 2(45). – с. 47-54. DOI: 10.31799/2077-5687-2025-2-47-54.
4. Comer, D. The Ubiquitous B-Tree / D. Comer // ACM Computing Surveys. – 1979. – Vol. 11, № 2. – P. 121–137. – DOI: 10.1145/356770.356776.
5. Kerttu Pollari-Malmi. B+-trees. Computer Science, Faculty of Science, University of Helsinki [Электронный ресурс]. – URL: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf (дата обращения 01.11.2025).
6. Microsoft Learn. Предупреждение C28182: Разыменование пустого (NULL) указателя [Электронный ресурс]. – URL: https://learn.microsoft.com/ru-ru/cpp/code-quality/c28182?view=msvc-170 (дата обращения 01.11.2025).
7. Microsoft Learn. Avoiding Buffer Overruns [Электронный ресурс]. – URL: https://learn.microsoft.com/en-us/windows/win32/SecBP/avoiding-buffer-overruns (дата обращения 01.11.2025).

ИНФОРМАЦИЯ ОБ АВТОРАХ

Андрей Викторович Шахомиров
Доцент кафедры аэрокосмических компьютерных и программных систем, кандидат технических наук
Санкт-Петербургский государственный университет аэрокосмического приборостроения
Россия, 190000, Санкт-Петербург, ул. Большая Морская, д.67, лит. А
E-mail: shakhomirov@guap.ru

Наталья Евгеньевна Шахомирова
Ассистент кафедры аэрокосмических компьютерных и программных систем
Санкт-Петербургский государственный университет аэрокосмического приборостроения
Россия, 190000, Санкт-Петербург, ул. Большая Морская, д.67, лит. А
E-mail: shakhomirova.natalia@guap.ru

INFORMATION ABOUT THE AUTHORS

Andrey Viktorovich Shakhomirov
Associate Professor of the Department of Aerospace Computer and Program Systems, PhD.
Saint-Petersburg State University of Aerospace Instrumentation
67, Bolshaya Morskaia str., Saint-Petersburg, 190000, Russia
E-mail: shakhomirov@guap.ru

Natalia Evgenievna Shakhomirova
Assistant of the Department of Aerospace Computer and Program Systems
Saint-Petersburg State University of Aerospace Instrumentation
67, Bolshaya Morskaia str., Saint-Petersburg, 190000, Russia
E-mail: shakhomirova.natalia@guap.ru

Дата поступления: 15.11.2025
Дата принятия: 26.11.2025