



Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет имени
Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Компьютерные системы и сети (ИУ-6)»

ОТЧЕТ О ВЫПОЛНЕНИИ ДОМАШНЕЙ РАБОТЫ
по дисциплине «Математические методы анализа данных и
принятия решений»

Студент:	Козлов Владимир Михайлович
Группа:	ИУ6-13М
Тип задания:	домашняя работа
Тема:	Байесовский классификатор

Студент _____
подпись, дата

Козлов В.М.
Фамилия, И.О.

Преподаватель _____
подпись, дата

Фамилия, И.О.

Москва, 2025

бла бла

Содержание

Введение	4
1 Основная часть	5
1.1 Постановка задачи распределения графа	5
1.2 Общая архитектура	5
1.3 Программная реализация	6
1.3.1 Структура проекта и основные зависимости	6
1.3.2 Классы для представления графовых структур (graph.hpp)	7
1.3.3 Структура класса Storage	11
1.4 Класс оптимизатора (optimizer.hpp).....	15
1.4.1 Структура класса оптимизатора	15
1.4.2 Основной метод расчёта метрик	16
1.4.3 Метод оптимизации	17
1.5 Тестирование.....	18
Заключение	21
Список использованных источников	22
Приложение А	23

Введение

Современные распределённые графовые базы данных сталкиваются с фундаментальной проблемой эффективного распределения вершин графа по узлам хранения (шардам). Оптимальное распределение становится критически важным для производительности систем, где основной операцией является поиск путей между вершинами, которые могут находиться в разных шардах. Неэффективное распределение приводит к значительным задержкам при выполнении запросов и избыточным сетевым коммуникациям между узлами.

Актуальность данной работы обусловлена стремительным ростом объёмов графовых данных в таких областях, как социальные сети, рекомендательные системы, биоинформатика и интернет вещей. Традиционные подходы к распределению данных демонстрируют ограниченную эффективность при работе с графиками, требующими учёта структурных особенностей и связности вершин.

В рамках исследования проводится сравнительный анализ двух принципиально различных подходов к распределению графов: потоковых методов (*online partitioning*), работающих в реальном времени по мере поступления данных, и методов оптимизации распределения (*offline partitioning*), требующих полного знания структуры графа. Особое внимание уделяется алгоритмам библиотеки METIS, представляющей собой промышленный стандарт для задач разбиения графов, и современным потоковым алгоритмам, таким как Fennel и Streaming Graph Partitioning.

Целью работы является исследование и сравнение эффективности различных методов распределения вершин графа по гомогенным хранилищам, а также разработка предложений по комбинированию подходов для достижения оптимального баланса между качеством разбиения и вычислительной эффективностью в условиях реальной эксплуатации распределённых графовых баз данных.

1 Основная часть

1.1 Постановка задачи распределения графа

Формальная постановка задачи, рассматриваемой в работе, может быть сформулирована следующим образом. Дан граф $G = (V, E)$, где $|V| = n$ - количество вершин, $|E| = m$ - количество рёбер. Распределение графа представляет собой разбиение $P = \{S_1, S_2, \dots, S_k\}$, где S_i - набор вершин (шард) такой, что $S_i \cap S_j = \emptyset$ для $i \neq j$ и $\bigcup_{i=1}^k S_i = V$.

Требуется найти такое распределение $P^* = \{S_1^*, S_2^*, \dots, S_k^*\}$, которое:

1. Минимизирует общую мощность разрезов:

$$|\partial e(P)| = \left| \bigcup_{i=1}^k e(S_i^*, V \setminus S_i^*) \right| \rightarrow \min \quad (1)$$

или относительную величину:

$$\lambda = \frac{|\partial e(P)|}{m} \times 100\% \rightarrow \min \quad (2)$$

2. Минимизирует нормализованную максимальную нагрузку (максимизирует балансировку):

$$\rho = \frac{\max_{i=1..k}(|S_i^*|)}{\frac{n}{k}} \rightarrow \min \quad (3)$$

Эта задача относится к классу NP-сложных задач [?], что обуславливает необходимость использования эвристических подходов, среди которых алгоритм Кернигана-Лина занимает важное место [?].

1.2 Общая архитектура

Упрощённая архитектура представлена на рисунке 1. БД предствалаяет состоит из одного мастера и множество хранилищ, соединённых общей шиной, через которую происходит общение.

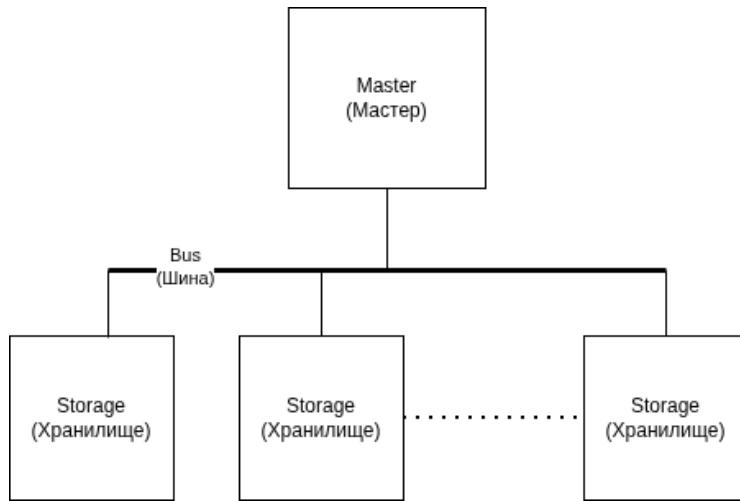


Рис. 1: Упрощённая архитектура

1.3 Программная реализация

В рамках данной курсовой работы была реализована вычислительная модель для расчёта и оптимизации метрики улучшения распределения g_v для граничных вершин графа согласно алгоритму Кернигана-Лина. Реализация включает в себя четыре основных компонента:

1. Классы для представления графовых структур (вершины, рёбра, ключи).
2. Классы имитации шины для общения компонентов
3. Класс хранилища (Storage) для управления вершинами и их связями.
4. Класс оптимизатора (StorageOptimizer) для расчёта метрики g_v .

Основной задачей практической части являлось создание инфраструктуры для вычисления функции улучшения распределения, определенной в алгоритме Кернигана-Лина:

$$g_v = \sum_{\substack{(v,u) \in E \\ P[v] \neq P[u]}} w(v, u) - \sum_{\substack{(v,u) \in E \\ P[v] = P[u]}} w(v, u)$$

1.3.1 Структура проекта и основные зависимости

Языком разработки был выбран C++ в силу его низкоуровневости и скорости, а также с намерением в дальнейшем внедрить эти наработки в графовую БД на C++ научного руководителя.

Проект организован в виде набора заголовочных файлов (header files), что соответствует современным подходам разработки на C++. Основные файлы проекта:

- `graph.hpp` – содержит базовые классы для представления графовых структур.

- `interface_bus.hpp` – содержит интерфейс, описывающий основные сообщения в шине.
- `bus.hpp` – содержит простую реализацию имитации шины.
- `storage.hpp` – реализует класс хранилища для управления вершинами.
- `optimizer.hpp` – содержит реализацию оптимизатора с вычислением метрики g_v .
- `main.cpp` – демонстрационный файл с тестовым сценарием.

Ниже представлен релевантный для решения задачи практики код. Полный код представлен в приложении А.

1.3.2 Классы для представления графовых структур (`graph.hpp`)

Класс NodeKey

Класс `NodeKey` представляет собой обёртку для ключа вершины графа. Он обеспечивает типобезопасность и возможность использования различных типов данных в качестве ключей (целые числа, строки и т.д.).

Листинг 1: Класс `NodeKey` в `graph.hpp`

```

1 template <typename KeyType>
2 class NodeKey {
3 public:
4   KeyType key_value;
5
6   NodeKey(): key_value(KeyType()) {};
7   NodeKey(const KeyType& key): key_value(key) {};
8
9   // Оператор присваивания
10  NodeKey<KeyType>& operator=(const NodeKey<KeyType>& other) {
11    if (this != &other) {
12      key_value = other.key_value;
13    }
14    return *this;
15  };
16
17 // Дружественные операторы сравнения
18 friend bool operator<(const NodeKey<KeyType>& lhs, const
19   NodeKey<KeyType>& rhs) {
20   return lhs.key_value < rhs.key_value;
21 }
22 friend bool operator>(const NodeKey<KeyType>& lhs, const
23   NodeKey<KeyType>& rhs) {
24   return rhs < lhs;

```

```

24 }
25
26 friend bool operator==(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
27     return lhs.key_value == rhs.key_value;
28 }
29
30 friend bool operator!=(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
31     return !(lhs == rhs);
32 }
33 };

```

Класс `NodeKey` является шаблонным, что позволяет использовать различные типы данных в качестве ключей вершин. Это важно для обеспечения гибкости при работе с различными типами графовых данных. Известно, что в конечной реализации используются строковые ключи, но была добавлена гибкость для потенциального использования пользовательских гибридных ключей.

Класс Edge

Класс `Edge` представляет ребро графа с весом и дополнительными параметрами.

Листинг 2: Класс Edge в `graph.hpp`

```

1 template <typename KeyType>
2 class Edge {
3 private:
4     float weight;
5     bool directional;
6     std::map<std::string, Parameter> parameters;
7     NodeKey<KeyType> from;
8     NodeKey<KeyType> to;
9 public:
10    float get_weight() const {
11        return weight;
12    }
13    bool is_directional() const {
14        return directional;
15    }
16    const NodeKey<KeyType>& get_from() const {
17        return from;
18    }
19    const NodeKey<KeyType>& get_to() const {
20        return to;
21    }
22
23    NodeKey<KeyType> get_other(const NodeKey<KeyType>& node)
const {

```

```

24     if (node == to) {
25         return from;
26     } else if (node == from) {
27         return to;
28     }
29
30     return node;
31 }
32
33 NodeKey<KeyType> get_other(NodeKey<KeyType>* node) const {
34     if (*node == to) {
35         return from;
36     } else if (*node == from) {
37         return to;
38     } else {
39         return *node;
40     }
41 }
42 };

```

Класс Node

Класс `Node` представляет вершину графа и содержит всю информацию о её связях с другими вершинами.

Листинг 3: Класс `Node` в `graph.hpp` (часть 1)

```

1 template <typename KeyType>
2 class Node {
3 private:
4     NodeKey<KeyType> key;
5 public:
6     std::map<std::string, Parameter> parameters;
7     std::map<NodeKey<KeyType>, Edge<KeyType>> edges;
8 }
9     NodeKey<KeyType> get_key() const {
10     return key;
11 }
12
13 void add_edge(Edge<KeyType> new_edge) {
14     NodeKey<KeyType> current_key = this->get_key();
15     edges[new_edge.get_other(current_key)] = new_edge;
16 }
17
18 void add_edges(std::map<NodeKey<KeyType>, Edge<KeyType>>
19 new_edges) {
20     edges.merge(new_edges);
21 }

```

```

21
22     bool remove_edge_to(NodeKey<KeyType> neighbour_key) {
23         return edges.erase(neighbour_key);
24     }
25
26     bool remove_edge(Edge<KeyType> removing_edge) {
27         NodeKey<KeyType> current_key = this->get_key();
28         return edges.erase(removeing_edge.get_other(current_key));
29     }
30 };

```

Класс имеет несколько конструкторов для удобного создания вершин с различными конфигурациями связей (см. приложение А). Все рёбра хранятся в вершине, от ссылок было решено отказаться, так как в конечной реализации хранилища должны находиться на разных машинах, а в вершины периодически перемещаться между ними.

Класс интерфейса шины (interface_bus.hpp)

Крайне важный интерфейс, через который происходит взаимодействие всей системы. Описывает методы добавления, удаления и запроса вершин, а также объявлений о добавлении и удалении для возможности другим хранилищам знать где находится другая вершина.

И наконец ключевые для алгоритма Кернигана-Лина методы получения граничных вершин и рёбер ask_neighbours_to_storage и ask_edges_to_storage.

Листинг 4: Базовая структура класса IBus

```

1 template <typename KeyType>
2 class IBus {
3 public:
4     virtual Node<KeyType> request_node(const NodeKey<KeyType>&
5     node) = 0;
6     virtual int send_add_node(const Node<KeyType>& node) = 0;
7     virtual bool send_add_node(const Node<KeyType>& node, int
8     storage_id) = 0;
9     virtual bool send_remove_node(const NodeKey<KeyType>& node) =
10    0;
11    virtual bool send_remove_node(const NodeKey<KeyType>& node,
12    int storage_id) = 0;
13    virtual int ask_who_has(int asker_id, NodeKey<KeyType> key) =
14    0;
15    virtual void announce_add(NodeKey<KeyType> key, int
16    storage_id, std::set<Edge<KeyType>> edges) = 0;
17    virtual void announce_remove(NodeKey<KeyType> key, int
18    storage_id) = 0;
19    // запрашивает у source вершины, соседствующие с target
20    virtual std::set<Node<KeyType>> ask_neighbours_to_storage(int
21    source, int target) = 0;

```

```

14     // запрашивает у source рёбра, идущие в target
15     virtual std::set<Edge<KeyType>> ask_edges_to_storage(int
16     source, int target) = 0;

```

Класс шины SimpleBus (bus.hpp)

Простая синхронная однопоточная реализация методов IBus. Релевантный код слишком длинный для вставки в основную часть, поэтому представлен в приложении А.

Класс хранилища (storage.hpp)

Класс Storage представляет собой хранилище вершин графа и реализует логику управления внутренними и внешними связями.

1.3.3 Структура класса Storage

Листинг 5: Базовая структура класса Storage

```

1 template <typename KeyType>
2 class Storage {
3 private:
4     typedef Node<KeyType> StorageNode;
5     typedef NodeKey<KeyType> Key;
6     int storage_id;
7     std::map<Key, StorageNode> nodes;
8     // external_edges[storage edge go to][external node][local node] = edge
9     std::map<int, std::map<Key, std::map<Key, Edge<KeyType>>>>
    external_edges;
10    IBus<KeyType> *bus = nullptr;
11
12 public:
13    Storage(int id)
14        :storage_id(id) {}
15    Storage(int id, std::map<Key, StorageNode> _nodes)
16        :storage_id(id), nodes(_nodes) {}
17
18    int get_id() const { return storage_id; }
19    void connect_to_bus(IBus<KeyType>* _bus) { bus = _bus; }

```

Хранилище идентифицируется уникальным storage_id и содержит вершины в виде хэш-таблицы для обеспечения быстрого доступа по ключу.

Добавление вершин с автоматическим созданием связей

Листинг 6: Методы добавления вершин класса Storage

```

1 std::optional<std::set<Edge<KeyType>>> add_node(const
StorageNode& node){
2     Key key = node.get_key();
3     std::cout << storage_id << " adding " << key.key_value <<
std::endl;

```

```

4   typename std::map<Key, StorageNode>::iterator it =
5     nodes.find(key);
6   if (it != nodes.end()) {
7     return std::nullopt;
8   }
9   nodes[key] = node;
10
11  std::set<Edge<KeyType>> external_edges_to_announce;
12  for (typename std::map<Key, Edge<KeyType>>::const_iterator
13    edge_it = node.edges.begin(); edge_it != node.edges.end();
14    ++edge_it) {
15    const Key& neighbor_key = edge_it->first;
16    const Edge<KeyType>& edge = edge_it->second;
17    std::cout << storage_id << " looks for " <<
18    neighbor_key.key_value << std::endl;
19    // Ищем соседа в текущем хранилище
20    typename std::map<Key, StorageNode>::iterator it2 =
21      nodes.find(neighbor_key);
22    if (it2 != nodes.end()) {
23      std::cout << storage_id << " node " <<
24      neighbor_key.key_value << " is inside, no asking" <<
25      std::endl;
26      // Сосед найден в этом же хранилище - добавляем обратное ребро
27      it2->second.add_edge(edge);
28    } else {
29      std::cout << storage_id << " node " <<
30      neighbor_key.key_value << " is outside, asking" << std::endl;
31      // Сосед не найден - спрашиваем у шины, где он находится
32      int neighbours_storage_id =
33        bus->ask_who_has(storage_id, neighbor_key);
34      std::cout << storage_id << " asked for " <<
35      neighbor_key.key_value << " answer: " << neighbours_storage_id
36      << std::endl;
37      if (neighbours_storage_id != -1) {
38        // Сохраняем внешнее ребро
39
40        external_edges[neighbours_storage_id][neighbor_key][key] =
41          edge;
42        external_edges_to_announce.insert(edge);
43      }
44      // Если сосед нигде не найден - игнорируем (возможно, будет добавлен
45      позже)
46    }
47  }
48
49  return external_edges_to_announce;

```

```

36 };
37
38 bool add_node_and_announce(const StorageNode& node) {
39     if (bus == nullptr) {
40         return false;
41     }
42     std::optional<std::set<Edge<KeyType>>> external_edges =
43         add_node(node);
44     if (!external_edges.has_value()) {
45         return false;
46     }
47     bus->announce_add(node.get_key(), storage_id,
48         external_edges.value());
49     return true;
50 }

```

Метод `add_node` не только добавляет вершину в хранилище, но и автоматически создает связи с её соседями, что обеспечивает целостность графовой структуры. Это требуется для будущей реализации потокового распределения. Также метод `add_node_and_announce` позволяет при добавлении объявить о добавлении новой вершины.

Удаление вершин с автоматическим удалением связей

Листинг 7: Метод удаления вершин класса Storage

```

1 bool remove_node(const Key& key) {
2     if (!has_node(key)) {
3         return false;
4     }
5     StorageNode node = nodes.find(key)->second;
6     nodes.erase(key);
7
8     for (typename std::map<NodeKey<KeyType>,
9         Edge<KeyType>>::iterator edge_it = node.edges.begin(); edge_it
10        != node.edges.end(); ++edge_it) {
11         Key other_key = edge_it->second.get_other(key);
12         typename std::map<Key, StorageNode>::iterator it =
13             nodes.find(other_key);
14         if (it != nodes.end()) {
15             it->second.remove_edge_to(key);
16         }
17     }
18
19     return true;
20 }
21
22 bool remove_node_and_announce(const Key& key) {

```

```

20     if (remove_node(key)) {
21         bus->announce_remove(key, storage_id);
22         return true;
23     }
24     return false;
25 };

```

Методы `remove_node` и `remove_node_and_announce` аналогично их `add` версиям автоматически следят за целостностью графа как локально, так и внешне. Это значительно снижает вычислительную сложность, так как исключает из рассмотрения вершины, не имеющие внешних связей.

Методы для работы с граничными вершинами

Для реализации граничного алгоритма Кернигана-Лина требуется уметь получать вершины, граничащие с другим хранилищем, чем занимается метод `get_nodes_with_neighbors_in` используя мапу внешних соседей.

Листинг 8: Метод для получения граничных вершин

```

1   std::set<StorageNode> result;
2
3   typename std::map<int, std::map<Key, std::map<Key,
4     Edge<KeyType>>>::const_iterator storage_it =
5     external_edges.find(target_storage_id);
6   if (storage_it == external_edges.end()) {
7     return result;
8   }
9
10  const std::map<Key, std::map<Key, Edge<KeyType>>>& node_edges =
11    storage_it->second;
12  for (typename std::map<Key, std::map<Key,
13    Edge<KeyType>>>::const_iterator node_edges_it =
14    node_edges.begin(); node_edges_it != node_edges.end();
15    ++node_edges_it) {
16    const std::map<Key, Edge<KeyType>>& local_node_edges =
17      node_edges_it->second;
18    for (typename std::map<Key,
19      Edge<KeyType>>::const_iterator local_node_edges_it =
20      local_node_edges.begin(); local_node_edges_it !=
21      local_node_edges.end(); ++local_node_edges_it) {
22      Key local_key = local_node_edges_it->first;
23      typename std::map<Key, StorageNode>::const_iterator
24      node_it = nodes.find(local_key);
25      if (node_it != nodes.end()) {
26        result.insert(node_it->second);
27      }
28    }
29  }

```

```
19
20     return result;
```

В листинге 9 представлен метод получения рёбер между хранилищами, что позволяет несколько упростить дальнейшие вычисления.

Листинг 9: Метод получения рёбер в другое хранилище

```
1 std::set<Edge<KeyType>> get_all_edges_to_storage(int
2     target_storage_id) const {
3     std::set<Edge<KeyType>> result;
4     typename std::map<int, std::map<Key, std::map<Key,
5         Edge<KeyType>>>>::const_iterator storage_it =
6         external_edges.find(target_storage_id);
7     if (storage_it == external_edges.end()) {
8         return std::set<Edge<KeyType>>();
9     } else {
10        const std::map<Key, std::map<Key, Edge<KeyType>>>&
11        external_nodes_map = storage_it->second;
12        for (typename std::map<Key, std::map<Key,
13             Edge<KeyType>>>::const_iterator node_edges_it =
14             external_nodes_map.begin(); node_edges_it !=
15             external_nodes_map.end(); ++node_edges_it) {
16            const std::map<Key, Edge<KeyType>>& node_edge_map =
17            node_edges_it->second;
18            for (typename std::map<Key,
19                 Edge<KeyType>>::const_iterator edges_it =
20                 node_edge_map.begin(); edges_it != node_edge_map.end();
21                 ++edges_it) {
22                result.insert(edges_it->second);
23            }
24        }
25    }
26    return result;
27 }
```

1.4 Класс оптимизатора (optimizer.hpp)

Класс `ExternalStorageOptimizer` является центральным компонентом реализации алгоритма Кернигана-Лина и отвечает за расчёт метрики улучшения g_v .

1.4.1 Структура класса оптимизатора

Листинг 10: Класс StorageOptimizer

```
1 template <typename KeyType>
```

```

2 class ExternalStorageOptimizer {
3 private:
4 IBus<KeyType>* bus;
5 public:
6 ExternalStorageOptimizer(IBus<KeyType>* _bus)
7     : bus(_bus) {}
8 }

```

1.4.2 Основной метод расчёта метрик

Метод `calculate_gvs()` демонстрирует практическую реализацию итерации Boundary KL (алгоритм работы только с граничными вершинами). Он получает граничные вершины из обоих хранилищ и вычисляет для каждой из них метрику улучшения g_v , а метод `calculate_gv` занимается непосредственно рассчётом метрики.

Листинг 11: Методы расчёта g_v

```

1 float calculate_gv(const Node<KeyType>& node,
2                     std::set<Edge<KeyType>> boundary_edges) const {
3     NodeKey<KeyType> this_key = node.get_key();
4     float internal_edges_weight = 0;
5     float external_edges_weight = 0;
6
7     std::map<NodeKey<KeyType>, Edge<KeyType>> boundary_edges_map;
8     for (typename std::set<Edge<KeyType>>::const_iterator edge_it
9          = boundary_edges.begin();
10         edge_it != boundary_edges.end(); ++edge_it) {
11         NodeKey<KeyType> other = edge_it->get_other(this_key);
12         if (!other == this_key) {
13             boundary_edges_map[other] = *edge_it;
14         }
15     }
16     for (typename std::map<NodeKey<KeyType>,
17           Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
18         edge_it != node.edges.end(); ++edge_it) {
19         const NodeKey<KeyType>& neighbor_key = edge_it->first;
20         const Edge<KeyType>& edge = edge_it->second;
21         if (boundary_edges_map.find(neighbor_key) !=
22             boundary_edges_map.end()) {
23             external_edges_weight += edge.get_weight();
24         } else {
25             internal_edges_weight += edge.get_weight();
26         }
27     }
28 }

```

```

25     return internal_edges_weight - external_edges_weight;
26 }
27
28 std::map<int, std::map<Node<KeyType>, float>> calculate_gvs(int
29     storage1, int storage2) const {
30     std::map<int, std::map<Node<KeyType>, float>> result;
31     // Получаем граничные вершины для обоих хранилищ
32     std::set<Node<KeyType>> boundary_nodes1 =
33     bus->ask_neighbours_to_storage(storage1, storage2);
34     std::set<Node<KeyType>> boundary_nodes2 =
35     bus->ask_neighbours_to_storage(storage2, storage1);
36
37     std::set<Edge<KeyType>> boundary_edges1 =
38     bus->ask_edges_to_storage(storage1, storage2);
39     std::set<Edge<KeyType>> boundary_edges2 =
40     bus->ask_edges_to_storage(storage2, storage1);
41
42     result[storage1] = std::map<Node<KeyType>, float>();
43     typename std::set<Node<KeyType>>::const_iterator it;
44     for (it = boundary_nodes1.begin(); it != boundary_nodes1.end(); ++it) {
45         const Node<KeyType>& node = *it;
46         result[storage1][node.get_key()] = calculate_gv(node,
47             boundary_edges1);
48     }
49
50     result[storage2] = std::map<Node<KeyType>, float>();
51     for (it = boundary_nodes2.begin(); it != boundary_nodes2.end(); ++it) {
52         const Node<KeyType>& node = *it;
53         result[storage2][node.get_key()] = calculate_gv(node,
54             boundary_edges2);
55     }
56     return result;
57 };

```

1.4.3 Метод оптимизации

Метод `optimize()` демонстрирует практическую реализацию оптимизации KL. Он итеративно пользуется алгоритмом Кернигана-Лина, получает вершины с негативным g_v , после чего перемещает их в другое хранилище.

Листинг 12: Метод оптимизации

```

1 void optimize(int storage1, int storage2, int iterations_limit =
5) {

```

```

2     if (iterations_limit == 0) return;
3
4     int iteration = 0;
5     std::map<int, std::set<Node<KeyType>>> negative_gvs =
6         get_negative_gvs(calculate_gvs(storage1, storage2));
7     do {
8         typename std::map<int, std::set<Node<KeyType>>>::iterator
9         map_it;
10        for (map_it = negative_gvs.begin(); map_it != negative_gvs.end(); ++map_it) {
11            int this_storage = map_it->first;
12            int other_storage = this_storage == storage1 ?
13                storage2 : storage1;
14            std::set<Node<KeyType>>& nodes = map_it->second;
15            typename std::set<Node<KeyType>>::const_iterator
16            set_it;
17            for (set_it = nodes.begin(); set_it != nodes.end(); ++set_it) {
18                const Node<KeyType>& node = *set_it;
19                Node<int> removed =
20                    bus->request_node(node.get_key());
21                bus->send_remove_node(removed.get_key(),
22                    this_storage);
23                bus->send_add_node(removed, other_storage);
24            }
25        }
26        ++iteration;
27        negative_gvs = get_negative_gvs(calculate_gvs(storage1,
28            storage2));
29    } while (iteration < iterations_limit &&
30        !negative_gvs.empty());
31 }
```

1.5 Тестирование

Для тестирования создаётся простой граф показанный на рисунке 2

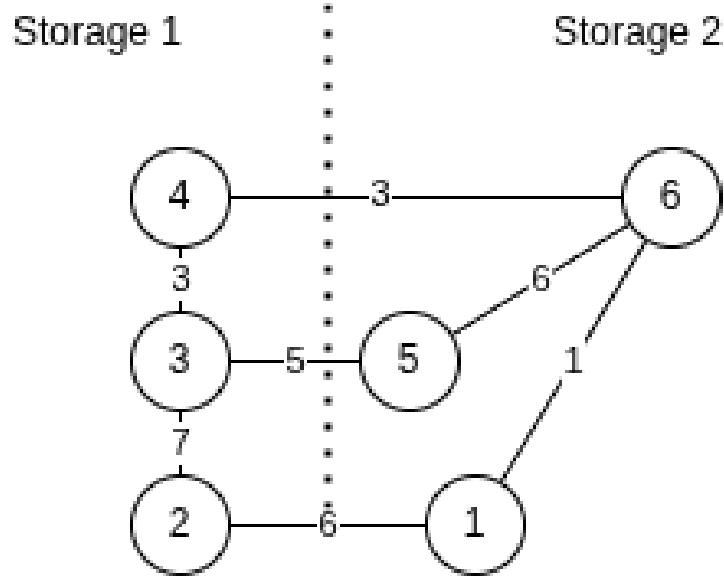


Рис. 2: Граф для тестирования

После оптимизации должно оказаться, что у всех вершин $g_v > 0$

Листинг 13: Вывод тестовой программы

```

storage: 1 node: Node(key: 1) gv: 5
storage: 1 node: Node(key: 3) gv: 0
storage: 2 node: Node(key: 4) gv: 1
storage: 2 node: Node(key: 5) gv: 1
storage: 2 node: Node(key: 6) gv: 8
Storage(id: 1, nodes: 3 {
    Node(key: 1, edges: [Edge(from: 1, to: 2, weight: 6), Edge(from: 1, to:
        ↪ 6, weight: 1)]),
    Node(key: 2, edges: [Edge(from: 1, to: 2, weight: 6), Edge(from: 2, to:
        ↪ 3, weight: 7)]),
    Node(key: 3, edges: [Edge(from: 2, to: 3, weight: 7), Edge(from: 3, to:
        ↪ 4, weight: 2), Edge(from: 3, to: 5, weight: 5)])
}, external_edges: {
    to storage 2: [
        external node 4: [
            local node 3 -> Edge(from: 3, to: 4, weight: 2)
        ],
        external node 5: [
            local node 3 -> Edge(from: 3, to: 5, weight: 5)
        ],
        external node 6: [
            local node 1 -> Edge(from: 1, to: 6, weight: 1)
        ]
    }
}

```

```

})
Storage(id: 2, nodes: 3 {
    Node(key: 4, edges: [Edge(from: 3, to: 4, weight: 2), Edge(from: 4, to:
        ↪ 6, weight: 3)]),
    Node(key: 5, edges: [Edge(from: 3, to: 5, weight: 5), Edge(from: 5, to:
        ↪ 6, weight: 6)]),
    Node(key: 6, edges: [Edge(from: 1, to: 6, weight: 1), Edge(from: 4, to:
        ↪ 6, weight: 3), Edge(from: 5, to: 6, weight: 6)])
}, external_edges: {
    to storage 1: [
        external node 1: [
            local node 6 -> Edge(from: 1, to: 6, weight: 1)
        ],
        external node 3: [
            local node 4 -> Edge(from: 3, to: 4, weight: 2),
            local node 5 -> Edge(from: 3, to: 5, weight: 5)
        ]
    ]
}
})

```

В результате получаем следующий граф:

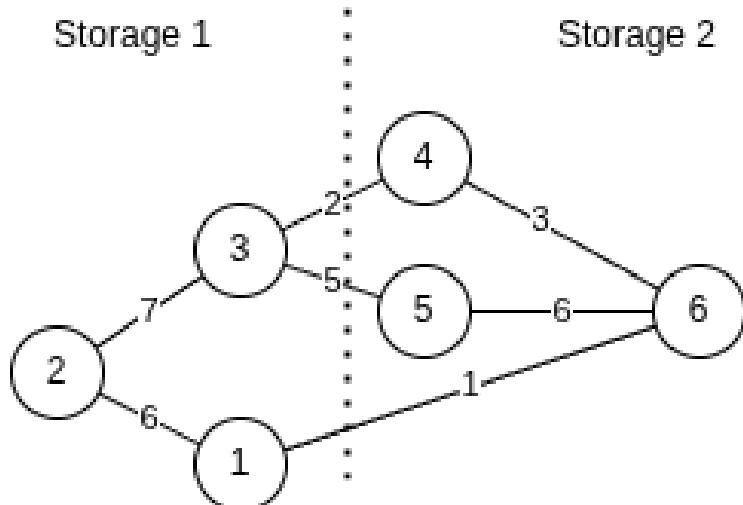


Рис. 3: Оптимизированный граф

Можно убедиться, что g_v действительно неотрицателен у всех и расчёты программы верны, граф успешно оптимизирован.

Заключение

В ходе выполнения курсового проекта была успешно исследована задача распределения вершин графа по гомогенным хранилищам и реализован ключевой компонент алгоритма Кернигана-Лина для оптимизации такого распределения.

На теоретическом уровне проведён анализ современных подходов к разбиению графов, включая как потоковые методы (Fennel, Streaming Graph Partitioning), так и методы оптимизации распределения (библиотека METIS). Установлено, что алгоритм Кернигана-Лина, несмотря на свою классическую природу, остаётся эффективным инструментом для уточнения разбиения графов, особенно в его оптимизированной граничной версии (BKL), которая значительно снижает вычислительные затраты.

На практическом уровне реализована вычислительная модель для расчёта метрики улучшения распределения g_v , являющейся основой алгоритма Кернигана-Лина. Разработаны:

- Гибкая система представления графовых структур с поддержкой различных типов ключей вершин
- Классы для управления вершинами и их связями в рамках отдельных хранилищ
- Оптимизатор, вычисляющий метрику g_v только для граничных вершин (реализация подхода Boundary KL)
- Демонстрационный пример, подтверждающий корректность работы реализованных компонентов

Ключевым достижением работы является реализация оптимизации Boundary KL, позволяющей работать только с вершинами, имеющими внешние связи, что значительно снижает вычислительную сложность алгоритма при сохранении качества оптимизации.

Полученные результаты подтверждают эффективность комбинированного подхода, при котором потоковые методы используются для начального распределения вершин, а алгоритм Кернигана-Лина — для последующей оптимизации. Такая стратегия позволяет достичь оптимального баланса между скоростью распределения и качеством разбиения графа, что особенно важно для распределённых графовых баз данных, работающих с большими объёмами данных в реальном времени.

Реализованная система может быть расширена для поддержки большего количества хранилищ, добавления полного цикла итераций алгоритма Кернигана-Лина и интеграции с реальными системами управления графовыми базами данных.

Список использованных источников

Приложение А

Полные листинги файлов программы.

Листинг 14: graph.hpp

```
1 #ifndef VKR_GRAPH
2 #define VKR_GRAPH
3
4 #include <stdlib.h>
5 #include <map>
6 #include <string>
7 #include <vector>
8 #include <numeric>
9 #include <utility>
10 #include <stdexcept>
11 #include <iostream>
12 #include <sstream>
13
14 template <typename KeyType>
15 class NodeKey {
16 public:
17     KeyType key_value;
18
19     NodeKey(): key_value(KeyType()) {};
20     NodeKey(const KeyType& key): key_value(key) {};
21
22     // Оператор присваивания
23     NodeKey<KeyType>& operator=(const NodeKey<KeyType>& other) {
24         if (this != &other) {
25             key_value = other.key_value;
26         }
27         return *this;
28     };
29
30     // Дружественные операторы сравнения
31     friend bool operator<(const NodeKey<KeyType>& lhs, const
32                             NodeKey<KeyType>& rhs) {
33         return lhs.key_value < rhs.key_value;
34     }
35     friend bool operator>(const NodeKey<KeyType>& lhs, const
36                             NodeKey<KeyType>& rhs) {
37         return rhs < lhs;
38     }
39     friend bool operator==(const NodeKey<KeyType>& lhs, const
```

```

        NodeKey<KeyType>& rhs) {
40    return lhs.key_value == rhs.key_value;
41 }
42
43 friend bool operator!=(const NodeKey<KeyType>& lhs, const
44     NodeKey<KeyType>& rhs) {
45     return !(lhs == rhs);
46 }
47
48
49 class Parameter{
50 private:
51     std::vector<std::byte> data;
52 public:
53     void set_data(const std::vector<std::byte>& _data) {
54         data = _data;
55     }
56     std::vector<std::byte> get_data() {
57         return data;
58     };
59 };
60
61 template <typename KeyType>
62 class Edge {
63 private:
64     float weight;
65     bool directional;
66     std::map<std::string, Parameter> parameters;
67     NodeKey<KeyType> from;
68     NodeKey<KeyType> to;
69 public:
70     Edge() : weight(1), directional(false) {}
71     Edge(const NodeKey<KeyType>& from_node, const
72           NodeKey<KeyType>& to_node)
73         : weight(1), directional(false), from(from_node),
74           to(to_node) {}
75
76     Edge(const NodeKey<KeyType>& from_node, const
77           NodeKey<KeyType>& to_node, int edge_weight)
78         : weight(edge_weight), directional(false),
79           from(from_node), to(to_node) {}
80
81     Edge(const NodeKey<KeyType>& from_node, const
82           NodeKey<KeyType>& to_node,
83           float edge_weight, bool is_directional)

```

```

79      : weight(edge_weight), directional(is_directional),
80      from(from_node), to(to_node) {}
81
82      Edge(const NodeKey<KeyType>& from_node, const
83      NodeKey<KeyType>& to_node,
84      float edge_weight, bool is_directional, const
85      std::map<std::string, Parameter>& params)
86      : weight(edge_weight), directional(is_directional),
87      parameters(params),
88      from(from_node), to(to_node) {}
89
90      Edge(const Edge& other)
91      : weight(other.weight), directional(other.directional),
92      parameters(other.parameters), from(other.from),
93      to(other.to) {}
94
95      float get_weight() const {
96          return weight;
97      }
98      bool is_directional() const {
99          return directional;
100     }
101     const std::map<std::string, Parameter>& get_parameters()
102     const {
103         return parameters;
104     }
105
106     Edge& get_reverted() {
107         return Edge<KeyType>(from, to, weight, directional,
108         parameters);
109     }
110
111     Edge& operator=(const Edge& other) {
112         if (this != &other) {
113             weight = other.weight;
114             directional = other.directional;
115             parameters = other.parameters;
116             from = other.from;
117             to = other.to;
118         }
119     }

```

```

118     return *this;
119 }
120
121 bool operator<(const Edge& other) const {
122     if (from != other.from) return from < other.from;
123     if (to != other.to) return to < other.to;
124     if (weight != other.weight) return weight < other.weight;
125     return directional < other.directional;
126 }
127
128 // doesn't check extra parameters
129 bool operator==(const Edge& other) {
130     if (other.directional != directional) return false;
131
132     if (directional) {
133         if (other.to != to || other.from != from) return
134         false;
135     } else {
136         if ((other.to != to && other.from != from) ||
137             (other.from != to && other.to != from)) return
138         false;
139     }
140
141     if (other.weight != weight) return false;
142
143     return true;
144 }
145
146 NodeKey<KeyType> get_other(const NodeKey<KeyType>& node)
147 const {
148     if (node == to) {
149         return from;
150     } else if (node == from) {
151         return to;
152     }
153
154     return node;
155 }
156
157 NodeKey<KeyType> get_other(NodeKey<KeyType>* node) const {
158     if (*node == to) {
159         return from;
160     } else if (*node == from) {
161         return to;
162     } else {
163         return *node;
164     }
165 }
```

```

161         }
162     }
163
164     friend std::ostream& operator<<(std::ostream& os, const
165     Edge<KeyType>& edge) {
166         os << "Edge(from: " << edge.get_from().key_value
167         << ", to: " << edge.get_to().key_value
168         << ", weight: " << edge.get_weight();
169
170         if (edge.is_directional()) {
171             os << ", directional";
172         }
173
174         // Добавляем информацию о параметрах, если они есть
175         const std::map<std::string, Parameter>& params =
176         edge.get_parameters();
177         if (!params.empty()) {
178             os << ", parameters: {";
179             typename std::map<std::string,
180             Parameter>::const_iterator it = params.begin();
181             if (it != params.end()) {
182                 os << it->first;
183                 ++it;
184             }
185             for (; it != params.end(); ++it) {
186                 os << ", " << it->first;
187             }
188             os << "}";
189         }
190     }
191 }
192
193 template <typename KeyType>
194 class Node {
195 private:
196     NodeKey<KeyType> key;
197 public:
198     std::map<std::string, Parameter> parameters;
199     std::map<NodeKey<KeyType>, Edge<KeyType>> edges;
200
201     Node():
202         key(NodeKey<KeyType>()) {};
203

```

```

204     Node( KeyType _key) :
205         key( NodeKey<KeyType>(_key)) {};
206
207     Node( NodeKey<KeyType> _key) :
208         key( _key) {};
209
210     Node( NodeKey<KeyType> _key, std :: map<NodeKey<KeyType>,
211           Edge<KeyType>>> _edges):
212         key( _key), edges(_edges) {};
213
214     Node( const Node& other)
215         : key(other.key),
216           parameters(other.parameters),
217           edges(other.edges) {};
218
219     NodeKey<KeyType> get_key() const {
220         return key;
221     }
222
223     void add_edge(Edge<KeyType> new_edge) {
224         NodeKey<KeyType> current_key = this->get_key();
225         edges[new_edge.get_other(current_key)] = new_edge;
226     }
227
228     void add_edges(std :: map<NodeKey<KeyType>, Edge<KeyType>>>
229                   new_edges) {
230         edges.merge(new_edges);
231     }
232
233     bool remove_edge_to(NodeKey<KeyType> neighbour_key) {
234         return edges.erase(neighbour_key);
235     }
236
237     bool remove_edge(Edge<KeyType> removing_edge) {
238         NodeKey<KeyType> current_key = this->get_key();
239         return edges.erase(removeing_edge.get_other(current_key));
240     }
241
242     Node& operator=(const Node& other) {
243         if (this != &other) {
244             key = other.key;
245             parameters = other.parameters;
246             edges = other.edges;
247         }
248         return *this;
249     }

```

```

248
249     bool operator<(const Node<KeyType>& other) const {
250         return key < other.key;
251     }
252
253     friend std::ostream& operator<<(std::ostream& os, const
254     Node<KeyType>& node) {
255         os << "Node(key: " << node.get_key().key_value;
256
257         // Выводим параметры узла
258         if (!node.parameters.empty()) {
259             os << ", parameters: {";
260             typename std::map<std::string,
261             Parameter>::const_iterator param_it = node.parameters.begin();
262             if (param_it != node.parameters.end()) {
263                 os << param_it->first;
264                 ++param_it;
265             }
266             for (; param_it != node.parameters.end(); ++param_it)
267             {
268                 os << ", " << param_it->first;
269             }
270             os << "}";
271
272         // Выводим ребра узла
273         if (!node.edges.empty()) {
274             os << ", edges: [";
275             typename std::map<NodeKey<KeyType>,
276             Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
277             if (edge_it != node.edges.end()) {
278                 os << edge_it->second;
279                 ++edge_it;
280             }
281             for (; edge_it != node.edges.end(); ++edge_it) {
282                 os << ", " << edge_it->second;
283             }
284             os << "]";
285         }
286     }
287 }
288
289 #endif // VKR_GRAPH

```

Листинг 15: bus.hpp

```
1 #ifndef VKR_BUS
2 #define VKR_BUS
3
4 #include "interface_bus.hpp"
5 #include "storage.hpp"
6 #include <map>
7 #include <set>
8
9 template <typename KeyType>
10 class SimpleBus : public IBus<KeyType> {
11 private:
12 std::map<int, Storage<KeyType>*> storages;
13 public:
14 SimpleBus(): storages() {};
15
16 int connect_storage(Storage<KeyType>* storage) {
17     if (storages.find(storage->get_id()) != storages.end()) {
18         return -1;
19     }
20     storages[storage->get_id()] = storage;
21     storage->connect_to_bus(this);
22     return storage->get_id();
23 };
24
25 Node<KeyType> request_node(const NodeKey<KeyType>& node) override
26 {
27     for (typename std::map<int, Storage<KeyType>*>::iterator it =
28         storages.begin(); it != storages.end(); ++it) {
29         Node<KeyType>* node_pointer = it->second->get_node(node);
30         if (node_pointer != nullptr) {
31             return Node<KeyType>(*node_pointer);
32             break;
33         }
34     }
35     return Node<KeyType>();
36 }
37 int send_add_node(const Node<KeyType>& node) override {
38     (void)node;
39     // No autosending
40     return -1;
41 }
```

```

42 bool send_add_node(const Node<KeyType>& node, int storage_id)
43     override {
44         if (storages.find(storage_id) == storages.end()) {
45             return false;
46         }
47         return storages[storage_id]→add_node_and_announce(node);
48     };
49
50 bool send_remove_node(const NodeKey<KeyType>& node) override {
51     bool success = false;
52     for (typename std::map<int, Storage<KeyType*>*>::iterator it =
53           storages.begin(); it != storages.end(); ++it) {
54         if (it→second→remove_node_and_announce(node)) {
55             success = true;
56             break;
57         };
58     }
59     return success;
60 };
61
62 bool send_remove_node(const NodeKey<KeyType>& node, int
63     storage_id) override {
64     if (storages.find(storage_id) == storages.end()) {
65         return false;
66     }
67     return storages[storage_id]→remove_node_and_announce(node);
68 };
69
70 int ask_who_has(int asker_id, NodeKey<KeyType> key) override{
71     for (typename std::map<int, Storage<KeyType*>*>::iterator it =
72           storages.begin(); it != storages.end(); ++it) {
73         if (it→second != nullptr && it→second→has_node(key)) {
74             return it→first;
75         };
76     }
77     return -1;
78 };
79
80 void announce_add(NodeKey<KeyType> key, int storage_id,
81     std::set<Edge<KeyType>> edges) override {
82     for (typename std::map<int, Storage<KeyType*>*>::iterator it =
83           storages.begin(); it != storages.end(); ++it) {
84         if (it→second != nullptr) {
85             it→second→get_add_announcement(key, storage_id,
86                 edges);
87         };
88     };

```

```

81     }
82 };
83
84 void announce_remove(NodeKey<KeyType> key, int storage_id)
85     override {
86     for (typename std::map<int, Storage<KeyType>*>::iterator it =
87         storages.begin(); it != storages.end(); ++it) {
88         if (it->second != nullptr) {
89             it->second->get_remove_announcement(key, storage_id);
90         }
91     }
92 // запрашивает у source вершины, соседствующие с target
93 std::set<Node<KeyType>> ask_neighbours_to_storage(int source, int
94     target) {
95     if (storages.find(source) == storages.end()) {
96         return std::set<Node<KeyType>>();
97     }
98     return
99         storages[source]->get_nodes_with_neighbors_in_storage(target);
100    }
101   // запрашивает у source рёбра, идущие в target
102   std::set<Edge<KeyType>> ask_edges_to_storage(int source, int
103     target) {
104     if (storages.find(source) == storages.end()) {
105         return std::set<Edge<KeyType>>();
106     }
107     return storages[source]->get_all_edges_to_storage(target);
108 }
109
110 #endif // VKR_BUS

```

Листинг 16: storage.hpp

```

1 #ifndef VKR_COURSE_STORAGE
2 #define VKR_COURSE_STORAGE
3
4 #include "graph.hpp"
5 #include "interface_bus.hpp"
6 #include <set>
7 #include <optional>
8 #include <vector>

```

```

9 #include <algorithm>
10 #include <iostream>
11 #include <map>
12
13 template <typename KeyType>
14 class Storage {
15 private:
16     typedef Node<KeyType> StorageNode;
17     typedef NodeKey<KeyType> Key;
18     int storage_id;
19     std::map<Key, StorageNode> nodes;
20     // external_edges[storage edge go to][external node][local node] = edge
21     std::map<int, std::map<Key, std::map<Key, Edge<KeyType>>>>
22         external_edges;
23     IBus<KeyType> *bus = nullptr;
24
25 public:
26     Storage(int id)
27         : storage_id(id) {}
28     Storage(int id, std::map<Key, StorageNode> _nodes)
29         : storage_id(id), nodes(_nodes) {}
30     int get_id() const { return storage_id; }
31     void connect_to_bus(IBus<KeyType>* _bus) { bus = _bus; }
32
33     void get_add_announcement(Key key, int announcer_id,
34         std::set<Edge<KeyType>> edges) {
35         if (announcer_id == storage_id) return;
36         for (typename std::set<Edge<KeyType>>::iterator edge_it =
37             edges.begin(); edge_it != edges.end(); ++edge_it) {
38             Key other_key = edge_it->get_other(key);
39             typename std::map<Key, StorageNode>::iterator it =
40                 nodes.find(other_key);
41             if (it != nodes.end()) {
42                 it->second.add_edge(*edge_it);
43                 typename std::map<Key, std::map<Key,
44                     Edge<KeyType>>>::iterator it2 =
45                     external_edges[announcer_id].find(key);
46                     if (it2 == external_edges[announcer_id].end()){
47                         external_edges[announcer_id][key] = std::map<Key,
48                             Edge<KeyType>>();
49                     }
50                     external_edges[announcer_id][key][other_key] =
51                         *edge_it;
52                 }
53             }

```

```

47     }
48 };
49
50 void get_remove_announcement(Key key, int announcer_id) {
51     if (announcer_id == storage_id) return;
52
53     typename std::map<int, std::map<Key, std::map<Key,
54         Edge<KeyType>>>>::iterator storage_it =
55         external_edges.find(announcer_id);
56     if (storage_it == external_edges.end()) {
57         return;
58     }
59
60     std::map<Key, std::map<Key, Edge<KeyType>>>& external_map =
61     storage_it->second;
62     typename std::map<Key, std::map<Key,
63         Edge<KeyType>>>::iterator external_node_it =
64         external_map.find(key);
65     if (external_node_it == external_map.end()) {
66         return;
67     }
68
69     // Удаляем рёбра из локальных узлов
70     for (typename std::map<Key, Edge<KeyType>>::iterator
71         local_nodes_it = local_to_ext_map.begin();
72         local_nodes_it != local_to_ext_map.end();
73         ++local_nodes_it) {
74         Key local_key = local_nodes_it->first;
75         typename std::map<Key, StorageNode>::iterator node =
76         nodes.find(local_key);
77         if (node != nodes.end()) {
78             node->second.remove_edge_to(key);
79         }
80
81     // Удаляем запись из external_edges
82     external_map.erase(external_node_it);
83
84     // Если для этого хранилища не осталось внешних узлов, удаляем запись
85     if (external_map.empty()) {
86         external_edges.erase(storage_it);
87     }
88 }
```

```

86
87 std::optional<std::set<Edge<KeyType>>> add_node(const
88 StorageNode& node){
89     Key key = node.get_key();
90     std::cout << storage_id << " adding " << key.key_value <<
91     std::endl;
92     typename std::map<Key, StorageNode>::iterator it =
93     nodes.find(key);
94     if (it != nodes.end()) {
95         return std::nullopt;
96     }
97     nodes[key] = node;
98
99     std::set<Edge<KeyType>> external_edges_to_announce;
100    for (typename std::map<Key, Edge<KeyType>>::const_iterator
101        edge_it = node.edges.begin(); edge_it != node.edges.end();
102        ++edge_it) {
103        const Key& neighbor_key = edge_it->first;
104        const Edge<KeyType>& edge = edge_it->second;
105        std::cout << storage_id << " looks for " <<
106        neighbor_key.key_value << std::endl;
107        // Ищем соседа в текущем хранилище
108        typename std::map<Key, StorageNode>::iterator it2 =
109        nodes.find(neighbor_key);
110        if (it2 != nodes.end()) {
111            std::cout << storage_id << " node " <<
112            neighbor_key.key_value << " is inside, no asking" <<
113            std::endl;
114            // Сосед найден в этом же хранилище - добавляем обратное ребро
115            it2->second.add_edge(edge);
116        } else {
117            std::cout << storage_id << " node " <<
118            neighbor_key.key_value << " is outside, asking" << std::endl;
119            // Сосед не найден - спрашиваем у шины, где он находится
120            int neighbours_storage_id =
121            bus->ask_who_has(storage_id, neighbor_key);
122            std::cout << storage_id << " asked for " <<
123            neighbor_key.key_value << " answer: " << neighbours_storage_id
124            << std::endl;
125            if (neighbours_storage_id != -1) {
126                // Сохраняем внешнее ребро
127                external_edges[neighbours_storage_id][neighbor_key][key] =
128                edge;
129                external_edges_to_announce.insert(edge);
130            }

```

```

117         // Если сосед нигде не найден - игнорируем (возможно, будет добавлен
118         // позже)
119     }
120
121     return external_edges_to_announce;
122 };
123
124 bool add_node_and_announce(const StorageNode& node) {
125     if (bus == nullptr) {
126         return false;
127     }
128     std::optional<std::set<Edge<KeyType>>> external_edges =
129     add_node(node);
130     if (!external_edges.has_value()) {
131         return false;
132     }
133     bus->announce_add(node.get_key(), storage_id,
134     external_edges.value());
135     return true;
136 }
137
138 bool remove_node(const Key& key) {
139     if (!has_node(key)) {
140         return false;
141     }
142     StorageNode node = nodes.find(key)->second;
143     nodes.erase(key);
144
145     for (typename std::map<NodeKey<KeyType>,
146     Edge<KeyType>::iterator edge_it = node.edges.begin(); edge_it
147     != node.edges.end(); ++edge_it) {
148         Key other_key = edge_it->second.get_other(key);
149         typename std::map<Key, StorageNode>::iterator it =
150         nodes.find(other_key);
151         if (it != nodes.end()) {
152             it->second.remove_edge_to(key);
153         }
154     }
155
156     // Удаляем все внешние рёбра, связанные с этим узлом
157     typename std::map<int, std::map<Key, std::map<Key,
158     Edge<KeyType>>>::iterator ext_storage_it;
159     for (ext_storage_it = external_edges.begin(); ext_storage_it
160     != external_edges.end(); ) {

```

```

155     std::map<Key, std::map<Key, Edge<KeyType>>>& ext_nodes =
156     ext_storage_it->second;
157
158     // Удаляем записи, где наш узел является локальным
159     typename std::map<Key, std::map<Key,
160     Edge<KeyType>>>::iterator ext_node_it;
161     for (ext_node_it = ext_nodes.begin(); ext_node_it != ext_nodes.end(); ) {
162         std::map<Key, Edge<KeyType>>& local_edges =
163         ext_node_it->second;
164
165         // Удаляем все рёбра, где локальный ключ - наш удаляемый узел
166         typename std::map<Key, Edge<KeyType>>::iterator
167         edge_it = local_edges.find(key);
168         if (edge_it != local_edges.end()) {
169             local_edges.erase(edge_it);
170         }
171
172         // Если после удаления для этого внешнего узла не осталось рёбер,
173         // удаляем запись
174         if (local_edges.empty()) {
175             ext_nodes.erase(ext_node_it++);
176         } else {
177             ++ext_node_it;
178         }
179     }
180
181     // Если для этого хранилища не осталось внешних узлов, удаляем запись
182     if (ext_nodes.empty()) {
183         external_edges.erase(ext_storage_it++);
184     } else {
185         ++ext_storage_it;
186     }
187
188 }
189
190 bool remove_node_and_announce(const Key& key) {
191     if (remove_node(key)) {
192         bus->announce_remove(key, storage_id);
193         return true;
194     }

```

```

195     return false ;
196 };
197
198 StorageNode* get_node(const Key& key) {
199     typename std::map<Key, StorageNode>::iterator it =
200     nodes.find(key);
201     if (it != nodes.end()) {
202         return &(it->second);
203     }
204     return nullptr;
205 };
206
207 bool has_node(const Key& key) const {
208     return nodes.find(key) != nodes.end();
209 }
210
211 const std::map<Key, StorageNode>& get_all_nodes() const {
212     return nodes;
213 }
214
215 size_t size() const {
216     return nodes.size();
217 }
218
219 void clear() {
220     nodes.clear();
221 }
222
223 //
224 // Получение наборов связанных с другим хранилищем
225 //
226
227 // Получить набор узлов, имеющих соседей в указанном хранилище (копии)
228 std::set<StorageNode> get_nodes_with_neighbors_in_storage(int
229     target_storage_id) const {
230     std::set<StorageNode> result;
231
232     typename std::map<int, std::map<Key, std::map<Key,
233         Edge<KeyType>>>>::const_iterator storage_it =
234         external_edges.find(target_storage_id);
235     if (storage_it == external_edges.end()) {
236         return result;
237     }
238
239     const std::map<Key, std::map<Key, Edge<KeyType>>>& node_edges

```

```

= storage_it->second;
237   for (typename std::map<Key, std::map<Key,
238     Edge<KeyType>>>::const_iterator node_edges_it =
239     node_edges.begin(); node_edges_it != node_edges.end();
240     ++node_edges_it) {
241     const std::map<Key, Edge<KeyType>>& local_node_edges =
242     node_edges_it->second;
243     for (typename std::map<Key,
244       Edge<KeyType>>::const_iterator local_node_edges_it =
245       local_node_edges.begin(); local_node_edges_it !=
246       local_node_edges.end(); ++local_node_edges_it) {
247       Key local_key = local_node_edges_it->first;
248       typename std::map<Key, StorageNode>::const_iterator
249       node_it = nodes.find(local_key);
250       if (node_it != nodes.end()) {
251         result.insert(node_it->second);
252       }
253     }
254   }
255
256   return result;
257 }
258
259 std::set<Edge<KeyType>> get_all_edges_to_storage(int
260   target_storage_id) const {
261   std::set<Edge<KeyType>> result;
262   typename std::map<int, std::map<Key, std::map<Key,
263     Edge<KeyType>>>::const_iterator storage_it =
264     external_edges.find(target_storage_id);
265   if (storage_it == external_edges.end()) {
266     return std::set<Edge<KeyType>>();
267   } else {
268     const std::map<Key, std::map<Key, Edge<KeyType>>&
269     external_nodes_map = storage_it->second;
270     for (typename std::map<Key, std::map<Key,
271       Edge<KeyType>>>::const_iterator node_edges_it =
272       external_nodes_map.begin(); node_edges_it !=
273       external_nodes_map.end(); ++node_edges_it) {
274       const std::map<Key, Edge<KeyType>>& node_edge_map =
275       node_edges_it->second;
276       for (typename std::map<Key,
277         Edge<KeyType>>::const_iterator edges_it =
278         node_edge_map.begin(); edges_it != node_edge_map.end();
279         ++edges_it) {
280           result.insert(edges_it->second);
281         }
282     }
283   }
284 }
```

```

263         }
264     }
265     return result;
266 }
267
268 typename std::map<Key, StorageNode>::iterator begin() {
269     return nodes.begin();
270 }
271
272 typename std::map<Key, StorageNode>::iterator end() {
273     return nodes.end();
274 }
275
276 typename std::map<Key, StorageNode>::const_iterator begin() const
277 {
278     return nodes.begin();
279 }
280
281 typename std::map<Key, StorageNode>::const_iterator end() const {
282     return nodes.end();
283 }
284
285 typename std::map<Key, StorageNode>::const_iterator cbegin() const {
286     return nodes.cbegin();
287 }
288
289 typename std::map<Key, StorageNode>::const_iterator cend() const {
290     return nodes.cend();
291 }
292
293 bool empty() const {
294     return nodes.empty();
295 }
296
297 friend std::ostream& operator<<(std::ostream& os, const
298     Storage<KeyType>& storage) {
299     os << "Storage(id: " << storage.get_id();
300
301     if (storage.empty()) {
302         os << ", empty";
303     } else {
304         os << ", nodes: " << storage.size() << " {";
305
306         typename std::map<Key, StorageNode>::const_iterator it =
307             storage.begin();

```

```

305     if (it != storage.end()) {
306         os << std::endl << " " << it->second;
307         ++it;
308     }
309     for (; it != storage.end(); ++it) {
310         os << ", " << std::endl << " " << it->second;
311     }
312     os << std::endl << "}";
313 }
314
315 // Вывод внешних рёбер
316 if (!storage.external_edges.empty()) {
317     os << ", external_edges: {";
318
319     typename std::map<int, std::map<Key, std::map<Key,
320 Edge<KeyType>>>::const_iterator storage_it =
321         storage.external_edges.begin();
322
323     if (storage_it != storage.external_edges.end()) {
324         os << std::endl << " to storage " <<
325         storage_it->first << ": {";
326
327         const std::map<Key, std::map<Key, Edge<KeyType>>&
328         ext_nodes = storage_it->second;
329         typename std::map<Key, std::map<Key,
330 Edge<KeyType>>>::const_iterator node_it = ext_nodes.begin();
331
332         if (node_it != ext_nodes.end()) {
333             os << std::endl << "      external node " <<
334             node_it->first.key_value << ": [";
335
336             const std::map<Key, Edge<KeyType>>& local_edges =
337             node_it->second;
338             typename std::map<Key,
339 Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
340
341             if (edge_it != local_edges.end()) {
342                 os << std::endl << "      local node " <<
343                 edge_it->first.key_value
344                     << " -> " << edge_it->second;
345                 ++edge_it;
346             }
347             for (; edge_it != local_edges.end(); ++edge_it) {
348                 os << ", " << std::endl << "      local node "
349                 << edge_it->first.key_value
350                     << " -> " << edge_it->second;

```

```

342         }
343         os << std::endl << "      ] ";
344         ++node_it;
345     }
346     for ( ; node_it != ext_nodes.end(); ++node_it) {
347         os << "," << std::endl << "      external node " <<
348         node_it->first.key_value << ": [ ";
349
350         const std::map<Key, Edge<KeyType>>& local_edges =
351         node_it->second;
352         typename std::map<Key,
353         Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
354
355         if (edge_it != local_edges.end()) {
356             os << std::endl << "      local node " <<
357             edge_it->first.key_value
358             << " -> " << edge_it->second;
359             ++edge_it;
360         }
361         for ( ; edge_it != local_edges.end(); ++edge_it) {
362             os << "," << std::endl << "      local node "
363             << edge_it->first.key_value
364             << " -> " << edge_it->second;
365         }
366         os << std::endl << "    ] ";
367     }
368     os << std::endl << "  } ";
369
370     const std::map<Key, std::map<Key, Edge<KeyType>>>&
371     ext_nodes = storage_it->second;
372     typename std::map<Key, std::map<Key,
373     Edge<KeyType>>>::const_iterator node_it = ext_nodes.begin();
374
375     if (node_it != ext_nodes.end()) {
376         os << std::endl << "      external node " <<
377         node_it->first.key_value << ": [ ";
378
379         const std::map<Key, Edge<KeyType>>& local_edges =
380         node_it->second;

```

```

377             typename std::map<Key,
378             Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
379
380             if (edge_it != local_edges.end()) {
381                 os << std::endl << "    local node " <<
382                 edge_it->first.key_value
383                     << " -> " << edge_it->second;
384                     ++edge_it;
385             }
386             for (; edge_it != local_edges.end(); ++edge_it) {
387                 os << "," << std::endl << "    local node "
388                 << edge_it->first.key_value
389                     << " -> " << edge_it->second;
390             }
391             os << std::endl << "]";
392             ++node_it;
393         }
394         for (; node_it != ext_nodes.end(); ++node_it) {
395             os << "," << std::endl << "    external node " <<
396             node_it->first.key_value << ":" [";
397
398             const std::map<Key, Edge<KeyType>>& local_edges =
399             node_it->second;
400             typename std::map<Key,
401             Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
402
403             if (edge_it != local_edges.end()) {
404                 os << std::endl << "    local node " <<
405                 edge_it->first.key_value
406                     << " -> " << edge_it->second;
407                     ++edge_it;
408             }
409             for (; edge_it != local_edges.end(); ++edge_it) {
410                 os << "," << std::endl << "    local node "
411                 << edge_it->first.key_value
412                     << " -> " << edge_it->second;
413             }
414             os << std::endl << "}";
415         }
416         os << ")" ;

```

```

415     return os;
416 }
417 };
418
419 #endif // VKR_COURSE_STORAGE

```

Листинг 17: optimizer.hpp

```

1 #ifndef VKR_OPTIMIZER
2 #define VKR_OPTIMIZER
3
4 #include "interface_bus.hpp"
5
6 #include <stdlib.h>
7 #include <iostream>
8 #include <map>
9
10 template <typename KeyType>
11 class ExternalStorageOptimizer {
12 private:
13 IBus<KeyType>* bus;
14
15 float calculate_gv(const Node<KeyType>& node,
16 std::set<Edge<KeyType>> boundary_edges) const {
17 NodeKey<KeyType> this_key = node.get_key();
18 float internal_edges_weight = 0;
19 float external_edges_weight = 0;
20
21 std::map<NodeKey<KeyType>, Edge<KeyType>> boundary_edges_map;
22 for (typename std::set<Edge<KeyType>>::const_iterator edge_it
23 = boundary_edges.begin();
24 edge_it != boundary_edges.end(); ++edge_it) {
25     NodeKey<KeyType> other = edge_it->get_other(this_key);
26     if (!other == this_key) {
27         boundary_edges_map[other] = *edge_it;
28     }
29 }
30
31 for (typename std::map<NodeKey<KeyType>,
32 Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
33 edge_it != node.edges.end(); ++edge_it) {
34     const NodeKey<KeyType>& neighbor_key = edge_it->first;
35     const Edge<KeyType>& edge = edge_it->second;
36     if (boundary_edges_map.find(neighbor_key) !=
37 boundary_edges_map.end()) {
38         external_edges_weight += edge.get_weight();

```

```

34         std :: cout << "node: " << node.get_key().key_value <<
35         " external neighbour: " << neighbor_key.key_value << std :: endl;
36     } else {
37         internal_edges_weight += edge.get_weight();
38     }
39 }
40 return internal_edges_weight - external_edges_weight;
41 }
42
43 std :: map<int , std :: set<Node<KeyType>>>
44     get_negative_gvs(std :: map<int , std :: map<Node<KeyType>, float>>>
45     full_map) {
46     std :: map<int , std :: set<Node<KeyType>>> result;
47     typename std :: map<int , std :: map<Node<KeyType>,
48         float>>::const_iterator it;
49     for (it = full_map.begin(); it != full_map.end(); ++it) {
50         result [it->first] = std :: set<Node<KeyType>>();
51         const std :: map<Node<KeyType>, float>& nodes = it->second;
52         typename std :: map<Node<KeyType>, float>::const_iterator
53         it2;
54         for (it2 = nodes.begin(); it2 != nodes.end(); ++it2) {
55             std :: cout << "storage: " << it->first << " node: " <<
56             it2->first << " gv: " << it2->second << std :: endl;
57             if (it2->second < 0) {
58                 result [it->first].insert(it2->first);
59             }
60         }
61         if (result [it->first].empty()) {
62             result .erase(it->first);
63         }
64     }
65     return result;
66 }
67 public:
68 ExternalStorageOptimizer(IBus<KeyType>* _bus)
69     : bus(_bus) {}
70
71 std :: map<int , std :: map<Node<KeyType>, float>> calculate_gvs(int
72     storage1, int storage2) const {
73     std :: map<int , std :: map<Node<KeyType>, float>> result;
74     // Получаем граничные вершины для обоих хранилищ
75     std :: set<Node<KeyType>> boundary_nodes1 =
76     bus->ask_neighbours_to_storage(storage1, storage2);
77     std :: set<Node<KeyType>> boundary_nodes2 =

```

```

bus->ask_neighbours_to_storage(storage2, storage1);
72
73     std::set<Edge<KeyType>> boundary_edges1 =
bus->ask_edges_to_storage(storage1, storage2);
74     std::set<Edge<KeyType>> boundary_edges2 =
bus->ask_edges_to_storage(storage2, storage1);
75
76     result[storage1] = std::map<Node<KeyType>, float>();
77     typename std::set<Node<KeyType>>::const_iterator it;
78     for (it = boundary_nodes1.begin(); it != boundary_nodes1.end(); ++it) {
79         const Node<KeyType>& node = *it;
80         result[storage1][node.get_key()] = calculate_gv(node,
boundary_edges1);
81     }
82
83     result[storage2] = std::map<Node<KeyType>, float>();
84     for (it = boundary_nodes2.begin(); it != boundary_nodes2.end(); ++it) {
85         const Node<KeyType>& node = *it;
86         result[storage2][node.get_key()] = calculate_gv(node,
boundary_edges2);
87     }
88     return result;
89 };
90
91 void optimize(int storage1, int storage2, int iterations_limit =
5) {
92     if (iterations_limit == 0) return;
93
94     int iteration = 0;
95     std::map<int, std::set<Node<KeyType>>> negative_gvs =
get_negative_gvs(calculate_gvs(storage1, storage2));
96     do {
97         std::cout << "iteration " << iteration << std::endl;
98         typename std::map<int, std::set<Node<KeyType>>>::iterator
map_it;
99         for (map_it = negative_gvs.begin(); map_it != negative_gvs.end();
++map_it) {
100             int this_storage = map_it->first;
101             int other_storage = this_storage == storage1 ?
storage2 : storage1;
102             std::set<Node<KeyType>>& nodes = map_it->second;
103             typename std::set<Node<KeyType>>::const_iterator
set_it;
104             for (set_it = nodes.begin(); set_it != nodes.end());

```

```

    ++set_it) {
105        const Node<KeyType>& node = *set_it;
106        Node<int> removed =
107            bus->request_node(node.get_key());
108            bus->send_remove_node(removed.get_key());
109            bus->send_add_node(removed, other_storage);
110    }
111    ++iteration;
112    negative_gvs = get_negative_gvs(calculate_gvs(storage1,
storage2));
113 } while (iteration < iterations_limit &&
114 !negative_gvs.empty());
115 }
116
117 #endif // VKR_OPTIMIZER

```

Листинг 18: main.cpp (Демонстрация работы)

```

1 #include "include/graph.hpp"
2 #include "include/bus.hpp"
3 #include "include/storage.hpp"
4 #include "include/optimizer.hpp"
5
6 #include <stdlib.h>
7 #include <iostream>
8 #include <vector>
9 #include <utility>
10
11 int main() {
12     SimpleBus<int> bus;
13     Storage<int> storage1(1);
14     Storage<int> storage2(2);
15
16     bus.connect_storage(&storage1);
17     bus.connect_storage(&storage2);
18
19     Node<int> node1(1);
20     Node<int> node2(2);
21     Node<int> node3(3);
22     Node<int> node4(4);
23     Node<int> node5(5);
24     Node<int> node6(6);
25
26     Edge<int> edge1_2(1, 2, 6); node1.add_edge(edge1_2);

```

```

node2.add_edge(edge1_2);
27   Edge<int> edge1_6(1, 6, 1); node1.add_edge(edge1_6);
node6.add_edge(edge1_6);
28   Edge<int> edge2_3(2, 3, 7); node2.add_edge(edge2_3);
node3.add_edge(edge2_3);
29   Edge<int> edge3_4(3, 4, 2); node3.add_edge(edge3_4);
node4.add_edge(edge3_4);
30   Edge<int> edge4_6(4, 6, 3); node4.add_edge(edge4_6);
node6.add_edge(edge4_6);
31   Edge<int> edge3_5(3, 5, 5); node3.add_edge(edge3_5);
node5.add_edge(edge3_5);
32   Edge<int> edge5_6(5, 6, 6); node5.add_edge(edge5_6);
node6.add_edge(edge5_6);

33
34   bus.send_add_node(node1, 2);
35   bus.send_add_node(node2, 1);
36   bus.send_add_node(node3, 1);
37   bus.send_add_node(node4, 1);
38   bus.send_add_node(node5, 2);
39   bus.send_add_node(node6, 2);

40
41   std::cout << storage1 << std::endl;
42   std::cout << storage2 << std::endl;

43
44   std::cout << "Neighbours of 1:" << std::endl;
45   for (const auto& element : bus.ask_neighbours_to_storage(1,
2)) {
46       std::cout << element << " ";
47   }
48   std::cout << std::endl;

49
50   std::cout << "Neighbours of 2:" << std::endl;
51   for (const auto& element : bus.ask_neighbours_to_storage(2,
1)) {
52       std::cout << element << " ";
53   }
54   std::cout << std::endl;

55
56   std::cout << "edges between 1 to 2:" << std::endl;
57   for (const auto& element : bus.ask_edges_to_storage(1, 2)) {
58       std::cout << element << " ";
59   }
60   std::cout << std::endl;

61
62   std::cout << "edges between 2 to 1:" << std::endl;
63   for (const auto& element : bus.ask_edges_to_storage(2, 1)) {

```

```
64         std::cout << element << " " ;
65     }
66     std::cout << std::endl ;
67
68     /*Node<int> removed = bus.request_node(2) ;
69     bus.send_remove_node(2) ;
70
71     std::cout << storage1 << std::endl ;
72     std::cout << storage2 << std::endl ;
73
74     bus.send_add_node(removed, 2) ;
75
76     std::cout << storage1 << std::endl ;
77     std::cout << storage2 << std::endl ;*/
78
79     ExternalStorageOptimizer<int> optimizer(&bus) ;
80
81     optimizer.optimize(1, 2, 10) ;
82
83     std::cout << storage1 << std::endl ;
84     std::cout << storage2 << std::endl ;
85
86     return 0;
87 }
```
