Ермаков С. Г., Забродин А. В., Костин М. А.
S. G. Yermakov, A. V. Zabrodin, M. A. Kostin

ПРИМЕНЕНИЕ ДИНАМИЧЕСКОГО ФИЛЬТРА БЛУМА ДЛЯ ОБРАБОТКИ БОЛЬШИХ ДАННЫХ В СИСТЕМАХ РЕАЛЬНОГО ВРЕМЕНИ

APPLICATION OF THE DYNAMIC BLOOM FILTER FOR PROCESSING BIG DATA IN REAL-TIME SYSTEMS

Ермаков Сергей Геннадьевич – доктор технических наук, профессор, заведующий кафедрой «Информатика и вычислительная техника» Петербургского государственного университета путей сообщения Императора Александра I (Россия, Санкт-Петербург); 190031, Россия, Санкт-Петербург, Московский пр., д. 9. E-mail: Ermakov@pgups.ru.
Sergey G. Yermakov – Doctor of Technical Sciences, Professor, Head of the Department of «Computer Science and Computer Engineering», Emperor Alexander I St. Petersburg State Transport University (Russia, Saint-Petersburg); 190031, Russia, Saint Petersburg, 9, Moskovsky Ave. E-mail: Ermakov@pgups.ru.

Забродин Андрей Владимирович – кандидат исторических наук, доцент кафедры «Информатика и вычислительная техника» Петербургского государственного университета путей сообщения Императора Александра I (Россия, Санкт-Петербург); 190031, Россия, Санкт-Петербург, Московский пр., д. 9. E-mail: Zabrodin@pgups.ru.
Andrej V. Zabrodin – PhD in History, Associate Professor, Computer Science and Computer Engineering Department, Emperor Alexander I St. Petersburg State Transport University (Russia, Saint-Petersburg); 190031, Russia, Saint Petersburg, 9, Moskovsky Ave. E-mail: Zabrodin@pgups.ru.

Костин Максим Андреевич – студент 3-го курса Петербургского государственного университета путей сообщения Императора Александра I (Россия, Санкт-Петербург); 190031, Россия, Санкт-Петербург, Московский пр., д. 9. E-mail: m.kkostin@yandex.ru.
Maksim A. Kostin – 3rd Year Student, Emperor Alexander I St. Petersburg State Transport University (Russia, Saint-Petersburg); 190031, Russia, Saint Petersburg, 9, Moskovsky Ave. E-mail: m.kkostin@yandex.ru.

Аннотация. В современных системах реального времени существует потребность в эффективных алгоритмах обработки больших данных с минимальным временем отклика. Целью исследования являлась разработка модифицированного динамического фильтра Блума, обеспечивающего оптимальный баланс между скоростью, точностью и потреблением памяти. Основной гипотезой выступало предположение о возможности создания вероятностной структуры данных, сохраняющей константное время поиска при динамическом изменении параметров. Для достижения поставленной цели были сформулированы задачи: разработать алгоритм автоматического подбора размера фильтра и числа хеш-функций, реализовать математическую модель и проверить её эффективность на реальных данных. Методология работы основана на вероятностных подходах и экспериментальном тестировании с использованием MurmurHash32 и простых хеш-функций. Экспериментальные исследования показали, что предложенная реализация динамического фильтра Блума обеспечивает высокую производительность при обработке больших наборов данных с переменной структурой, минимизируя вероятность ложноположительных срабатываний и оптимизируя использование памяти.

Summary. In modern real-time systems, there is a need for efficient algorithms for processing big data with minimal response time. The aim of the study was to develop a modified dynamic Bloom filter that provides an optimal balance between speed, accuracy, and memory consumption. The main hypothesis was the possibility of creating a probabilistic data structure that preserves constant search time with dynamic parameter changes. To achieve this goal, the tasks were formulated: to develop an algorithm for automatically selecting the filter size and the number of hash functions, to implement a mathematical model and test its effectiveness on real data. The methodology of the work is based on probabilistic approaches and experimental testing using MurmurHash32 and simple hash functions. Experimental studies have shown that the proposed implementation of the dynamic Bloom filter provides high performance when processing large datasets with variable structure, minimizing the likelihood of false positives and optimizing memory usage.

Ключевые слова: функция, хеширование, класс, переменная, структура данных, битовый массив, аналитика данных, большие данные, минимизация, оптимизация данных, вероятностные структуры данных.

Key words: function, hashing, class, variable, data structure, bit array, data analysis, big data, minimization, data optimization, probabilistic data structures.

УДК 004.041:004.056

Введение. Темой исследования является разработка динамической структуры данных на основе фильтра Блума для быстрого определения принадлежности элемента к множеству с минимальным временем отклика. Вероятностные структуры данных, такие как фильтр Блума, благодаря своей простоте и эффективности широко применяются для проверки принадлежности элементов. Однако классическая версия фильтра Блума имеет ограничения: фиксированный размер структуры и низкую адаптивность к динамическим данным. В данной работе предпринята попытка преодолеть указанные ограничения путём создания динамической структуры данных, которая способна автоматически адаптироваться к изменяющимся условиям. Это достигается за счёт реализации алгоритма, который позволяет фильтру эффективно перераспределять память и настраивать параметры, такие как количество хеш-функций, в зависимости от объёма и свойств данных. Работа концентрируется на теоретическом обосновании и практической реализации предложенного подхода. Теоретическая значимость исследования заключается в разработке новой математической модели, которая служит основой для функционирования динамической структуры данных. Практическая значимость работы заключается в тестировании реализованного алгоритма на реальных данных, что позволяет оценить его эффективность при обработке больших массивов информации. Полученные результаты подтверждают пригодность фильтра для решения задач анализа сетевых данных, мониторинга и оптимизации процессов. В частности, фильтр может быть успешно использован в таких областях, как сетевой мониторинг на железных дорогах, обнаружение аномалий и сбоев в системах транспорта в режиме реального времени, благодаря эффективному использованию памяти и минимальному времени отклика. Теоретическая значимость работы заключается в формировании новой математической модели, которая будет являться основой для работы структуры данных. Практическая значимость разрабатываемого динамического фильтра Блума состоит в широкой возможности применения в различных областях, включая сетевой мониторинг на железных дорогах, обнаружение аномалий, сбоев в системе работы транспорта в режиме реального времени за счёт эффективного использования памяти и небольшого времени отклика.

Фильтр Блума – одна из множества вероятностных структур данных, позволяющих незамедлительно проверять принадлежность элемента к множеству данных. Существует возможность получения ложноположительного срабатывания – получения сообщения о наличии элемента во множестве при его фактическом отсутствии. Динамический фильтр Блума, рассматриваемый в статье, представляет собой изменённую реализацию классического фильтра Блума. Главными особенностями новой структуры данных являются: автоматическое определение оптимальной ёмкости фильтра (битового массива), благодаря которому динамический фильтр может адаптироваться к любому количеству элементов, которые нужно поместить в него; выбор оптимального количества хеш-функций, которые необходимы для минимизации вероятности ложноположительного срабатывания, но при этом не влияют на скорость проверки принадлежности элемента; оптимизация использования памяти в связи с использованием минимального количества битов.

Математическая модель алгоритма. Пусть размер битового массива m, а k – количество хеш-функций

$$p(h_i(x) \neq j) = 1 - \frac{1}{m}, \quad (1)$$

где $h_i(x) \neq j$ – вероятность того, что j-й бит останется нулевым после применения i-й хеш-функции.

Вероятность события (обозначим событие A), что в j-й бит не будет записана единица i-й хеш-функцией при вставке очередного элемента, рассчитывается по формуле (1) [1, 109]. При этом следует отметить, что событие A можно считать локальным, т. к. оно связано с каждым конкретным элементом и хеш-функцией.

Для упрощения анализа можно предположить, что значения хеш-функций – это независимые случайные величины. Вероятность того, что j-й бит останется нулевым после добавления очередного элемента и применения всех k хеш-функций, определится по формуле

$$p(h_i(x) \neq j, \forall i \in \{1, k\}) = \left(1 - \frac{1}{m}\right)^k, \quad (2)$$

где $p(h_i(x) \neq j, \forall i \in \{1, k\})$ – вероятность того, что j-й бит не будет установлен в 1 после применения всех k хеш-функций.

Следует отметить, что предположение о независимости хеш-функций является упрощением в рамках нашего исследования, которое значительно облегчает анализ и вычисления. В реальных условиях это допущение может быть не всегда справедливо, поскольку хеш-функции могут быть зависимыми, что, в свою очередь, потребует более сложных моделей для оценки [8].

После вставки n различных элементов в пустой фильтр вероятность того, что j-й бит будет равен нулю (обозначим как событие B) при использовании k хеш-функций, рассчитывается по формуле [1, 111]

$$p(B) = \left(1 - \frac{1}{m}\right)^{kn}. \quad (3)$$

Событие B можно считать глобальным, т. к. оно описывает вероятность для всего множества элементов, вставляемых в фильтр.

Здесь важно отметить, что прямое использование выражения (3) может быть не совсем удобным для практических вычислений, особенно при больших значениях m и n, вследствие того, что оно быстро становится громоздким и сложным для дальнейшего анализа. Для упрощения расчётов в случае, когда размер хеш-таблицы m достаточно велик, можно воспользоваться приближением с использованием второго замечательного предела. Это позволяет выразить вероятность более простым способом, заменив сложную степень на экспоненциальную функцию, в результате мы получим формулу

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}}. \quad (4)$$

Приведённая формула (4) предоставляет более удобную форму для вычислений и анализа при больших значениях m и n. Опираясь на полученную вероятность, рассмотрим ложноположительное срабатывание – ситуацию, когда фильтр ошибочно утверждает, что элемент присутствует, хотя на самом деле он не был добавлен. Ложноположительное срабатывание произойдёт тогда, когда для несуществующего элемента все k бит окажутся ненулевыми и фильтр Блума ответит, что он входит в число вставленных элементов. Вероятность такого события будет определяться по формуле

$$p(C) = \left(1 - e^{\frac{-kn}{m}}\right)^k. \quad (5)$$

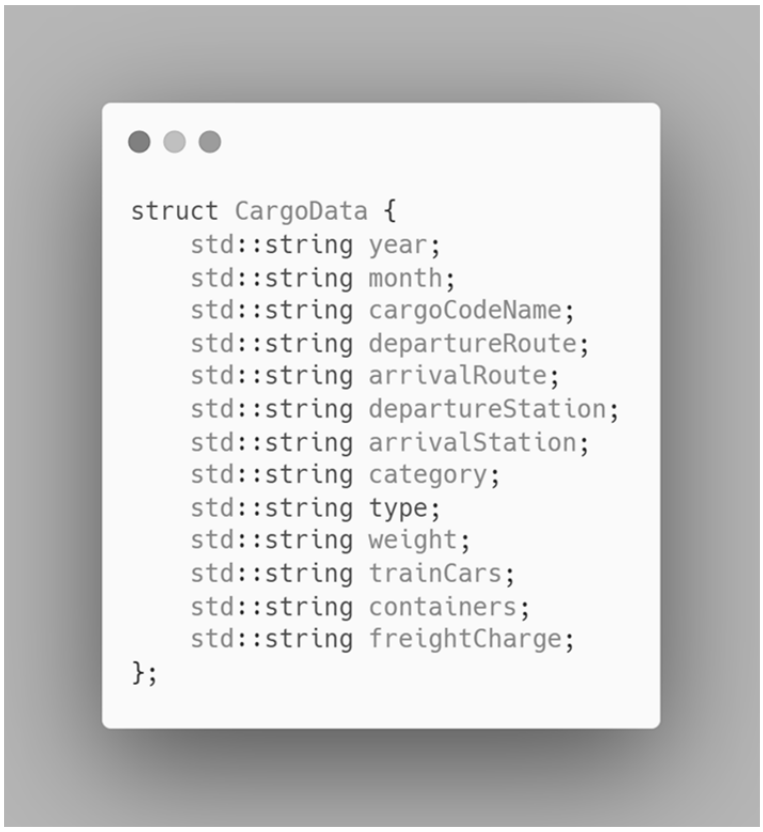
Оптимальный размер динамического фильтра Блума m определяется исходя из количества элементов n и желаемой вероятности ложноположительного срабатывания p:

$$m = \frac{-n * \ln p}{(\ln 2)^2}. \quad (6)$$

На основе полученного значения m оптимальное количество хеш-функций k рассчитывается по формуле (6) так, чтобы минимизировать вероятность ложноположительных срабатываний [2, 171]. В программной реализации эти вычисления выполняются в конструкторе.

$$k = \frac{m}{n}\ln 2 \approx 0.6931\frac{m}{n}. \quad (7)$$

Апробация математической модели фильтра Блума проводилась на реальных данных, что позволило оценить практическую применимость и точность алгоритма в условиях, приближённых к реальным. Тестирование работы алгоритма проводилось на данных, отражающих деятельность транспортной инфраструктуры. На рис. 1 представлена структура данных, которая будет хранить одну запись из обрабатываемого массива. Таким образом, фильтр Блума должен в качестве элемента получать набор таких строк, формировать для них индексы и записывать в битовый массив. После записи всех строк происходит проверка наличия одной заданной заранее записи.



```
struct CargoData {
    std::string year;
    std::string month;
    std::string cargoCodeName;
    std::string departureRoute;
    std::string arrivalRoute;
    std::string departureStation;
    std::string arrivalStation;
    std::string category;
    std::string type;
    std::string weight;
    std::string trainCars;
    std::string containers;
    std::string freightCharge;
};
```

Рис. 1. Структура для хранения записей из .csv-таблиц отчётов

Вычисления по формулам (1) – (7) производятся в конструкторе динамического фильтра Блума. Основа структуры данных, необходимой для решения поставленной проблемы, это класс, содержащий конструктор и необходимые поля для хранения битового массива с индексами, массива хеш-функций, необходимых для получения индексов, а также словарь для отслеживания коллизий [3, 129]. Под коллизией подразумевается случай совпадения индексов для некоторых элементов в битовом массиве. Подробно структуры для хранения вышеперечисленных инструментов показаны на рис. 2.

Одна из главных хеш-функций, применённых в реализации структуры данных, это «MurmurHash». Так как MurmurHash очень быстрый алгоритм хеширования, он очень важен для эффективной работы фильтра Блума. MurmurHash обеспечивает хорошее равномерное распределение хеш-значений, что снижает вероятность коллизий в фильтре Блума. К тому же алгоритм хеш-функции оптимизирован для эффективного использования памяти, что важно для больших фильтров Блума [8]. Пример программной реализации алгоритма MurmurHash32 приведён на рис. 3.

```
class DynBloomFilter
{
private:
    std::vector<bool> data; //битовый массив
    std::vector<std::function<size_t(const std::string&)>> hashFunctions; //вектор хеш-функций
    std::unordered_map<size_t, size_t> collisionMap; // Для подсчета коллизий
    std::unordered_map<std::string, std::vector<size_t>> elementIndices; // Для отслеживания индексов
элементов и абсолютных коллизий

    int hash_f_counter;
    size_t size;

    void resetBit(size_t index)
    {
        if (index < data.size())
        {
            data[index] = false;
            collisionMap[index]--;
        }
    }

public:
    DynBloomFilter(int numElements, double falsePositiveRate)
    {
        size_t m = static_cast<size_t>(-numElements * log(falsePositiveRate) / (log(2) * log(2)));

        int k = static_cast<int>(std::log(2) * m / numElements);

        // Динамическое увеличение размера
        data.resize(m, 0);

        //Добавление хеш-функций

        for (int i = 0; i < k; ++i)
        {
            if (i == 0)
            {
                addHashFunction([this](const std::string& name) { return this->hashFunction1(name); });
            }
            else if (i == 1)
            {
                addHashFunction([this](const std::string& name) { return this->hashFunction2(name); });
            }
            else if (i == 2)
            {
                addHashFunction([this](const std::string& name) { return this->murmurHash(name); });
            }
            else if (i > 2)
            {
                auto hashFunction = [i](const std::string& name) {
                    return std::hash<std::string>{}(name)+i;
                    };
                addHashFunction(hashFunction);
            }
        }
    }
```

Рис. 2. Поля структуры данных «Динамический фильтр Блума» и конструктор с вычислениями

Отличительная черта динамического фильтра Блума, которая полезна для его использования в сложных серверных системах реального времени или для работы с большими данными, это выполнение проверки принадлежности элемента к множеству за O(1), следовательно, эффективность этой операции не зависит от количества данных и размера фильтра Блума, что обеспечивается работой нескольких хеш-функций, распределяющих индексы по битовому массиву (см. рис. 4), где $\{x,y,z\}$ – входные данные, которым по хеш-функциям предоставляются индексы в битовом массиве, причём каждый элемент получит 3 индекса, т. к. используются 3 хеш-функции, и элемент w, наличие которого проверяется.

```cpp
uint32_t murmur3_32(const std::string& myString, uint32_t len, uint32_t seed)
{
    const char* key = myString.c_str();

    static const uint32_t c1 = 0xcc9e2d51;
    static const uint32_t c2 = 0x1b873593;
    static const uint32_t r1 = 15;
    static const uint32_t r2 = 13;
    static const uint32_t m = 5;
    static const uint32_t n = 0xe6546b64;

    uint32_t hash = seed;

    const int nblocks = len / 4;
    const uint32_t* blocks = (const uint32_t*)key;
    int i;
    for (i = 0; i < nblocks; i++) {
        uint32_t k = blocks[i];
        k *= c1;
        k = (k << r1) | (k >> (32 - r1));
        k *= c2;

        hash ^= k;
        hash = ((hash << r2) | (hash >> (32 - r2))) * m + n;
    }

    const uint8_t* tail = (const uint8_t*)(key + nblocks * 4);
    uint32_t k1 = 0;

    switch (len & 3) {
    case 3:
        k1 ^= tail[2] << 16;
    case 2:
        k1 ^= tail[1] << 8;
    case 1:
        k1 ^= tail[0];

        k1 *= c1;
        k1 = (k1 << r1) | (k1 >> (32 - r1));
        k1 *= c2;
        hash ^= k1;
    }

    hash ^= len;
    hash ^= (hash >> 16);
    hash *= 0x85ebca6b;
    hash ^= (hash >> 13);
    hash *= 0xc2b2ae35;
    hash ^= (hash >> 16);

    return hash;
}

// Адаптированная функция хеширования MurmurHash для использования в BloomFilter
size_t murmurHash(const std::string& str) {
    uint32_t len = static_cast<uint32_t>(str.length());
    uint32_t seed = 0;
    uint32_t hash = murmur3_32(str, len, seed);
    return static_cast<size_t>(hash);
}
```

Рис. 3. Реализация MurmurHash32

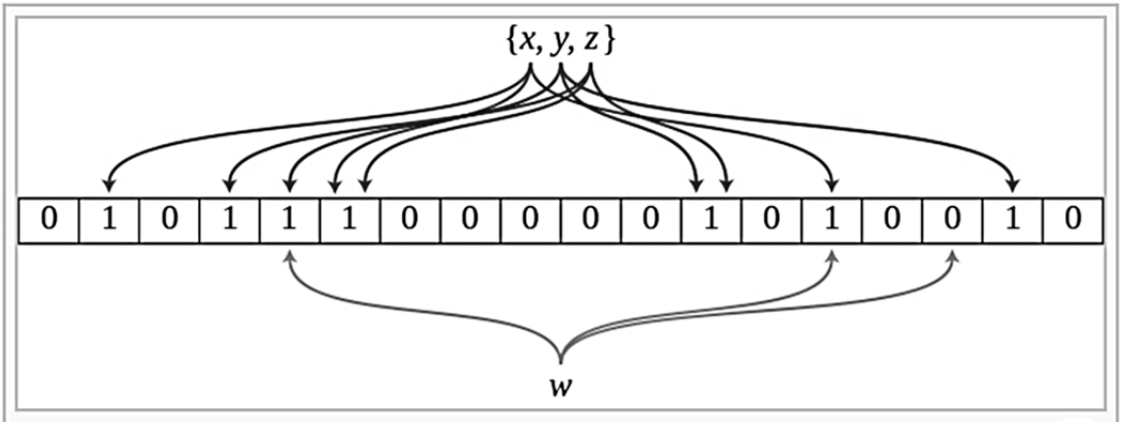


Рис. 4. Пример работы фильтра Блума

В ходе разработки, как было сказано выше, использовалась хеш-функция MurmurHash32, а также 2 простые хеш-функции на основе простых чисел (см. рис. 5).

```
1     size_t hashFunction1(const std::string& str)
2     {
3         size_t hash = 0;
4         for (char c : str) {
5             hash = hash * 31 + c; // Пример хэш-функции для строки
6         }
7         return hash;
8     }
9
10    size_t hashFunction2(const std::string& str) {
11        size_t hash = 0;
12        for (char c : str) {
13            hash = hash * 37 + c;
14        }
15        return hash;
16    }
```

Рис. 5. Реализация хеш-функции с использованием простых чисел

Количество хеш-функций в фильтре Блума определяет его эффективность: увеличение их числа снижает вероятность ложноположительных срабатываний, но одновременно возрастает вычислительная нагрузка [5, 376]. Поэтому в рамках предлагаемого исследования одним из ключевых усовершенствований стала возможность гибко изменять количество хеш-функций по мере необходимости. Для этого в конструкторе (см. рис. 2) задействованы лямбда-функция и стандартный метод хеширования, позволяющие при необходимости добавлять новые хеш-функции и, как следствие, поддерживать оптимальный баланс между точностью и производительностью. Для тестирования динамического фильтра Блума были использованы данные, извлечённые из электронных таблиц с широким набором операционных параметров (уникальные идентификаторы, числовые характеристики, коды локаций и т. д.) и отражающие реальные процессы. Многообразие

этих полей не влияет на работоспособность структуры, поскольку предметная область не накладывает ограничений на внутренние механизмы фильтра. На рис. 6 показан результат процесса поиска одной из записей в первом наборе данных (были использованы пять таблиц, каждая содержит около миллиона записей). В консоли выводится общий объём занимаемой памяти (включая битовый массив и хеш-функции), число хеш-функций, а также результат проверки на наличие тестируемой записи. Полученные данные подтверждают способность фильтра эффективно обнаруживать заданную запись, что свидетельствует о правильности его функционирования при заданных условиях.

```
Size DFB =9585058
HF count: 3
1000001
Cargo data exists: Yes
1000001
Cargo data exists: No
1000001
Cargo data exists: No
1000001
Cargo data exists: No
1000001
Cargo data exists: No
Время выполнения алгоритма: 35 секунд
Размер объекта DynBloomFilter: 192 байт
```

Рис. 6. Результат работы алгоритма на 5 миллионах записей
с проверкой принадлежности записи после обработки каждой таблицы

Исследование времени работы метода для проверки принадлежности записи к общему множеству показало, что этот показатель практически не изменяется при увеличении исходного набора с одного до пяти миллионов записей. Результаты, выводимые в консоль, свидетельствуют о том, что длительность поиска в битовом массиве не зависит ни от размера самого массива, ни от общего объёма обработанных данных [6, 18].

Динамический фильтр Блума, как и классическая модель фильтра, позволяет интерпретировать любые данные как битовый массив [7, 52], обеспечивая тем самым практически мгновенный поиск при наличии некоторой вероятности ложноположительных срабатываний. Благодаря индексации и эффективному поиску в больших наборах данных, где точное количество элементов неизвестно заранее, динамический фильтр Блума может успешно применяться при мониторинге систем и сетевого трафика (к примеру, для отслеживания уникальных доменов или запросов в реальном времени), при анализе логов для выявления уникальных ошибок при работе сервера, а также в рекомендательных системах, где необходимо быстро определить уникальные элементы пользовательской деятельности. Динамический фильтр Блума особенно полезен там, где требуется баланс между скоростью и точностью и при этом невозможно точно предсказать объём обрабатываемых данных заранее. Способность автоматически адаптироваться к изменениям в данных делает динамический фильтр Блума универсальным инструментом для обработки неструктурированных или эволюционирующих наборов данных.

Выводы. При разработке динамического фильтра Блума был реализован механизм вычисления необходимого числа хеш-функций для минимизации коллизий и устранения эффекта ложноположительного срабатывания. Динамический фильтр Блума демонстрирует значительные

преимущества в обработке больших данных в системах реального времени, обеспечивая баланс между скоростью, точностью и эффективностью использования ресурсов. Предложенное решение имеет потенциал стать мощным инструментом в современных системах анализа данных и обработки информации в режиме реального времени. В ходе исследования были выполнены главные критерии для использования динамического фильтра Блума в системах реального времени – константное время выполнения поиска элемента при динамически подобранных параметрах (размер фильтра Блума и количество хеш-функций).

ЛИТЕРАТУРА

1. Demetrescu C. Experimental Algorithms: Proceedings of the International Workshop on Experimental Algorithms – 2007. Algorithm Engineering Research Collection. Rome: La Sapienza University, 2007. – 458 p.
2. Patgiri R., Nayak S., Muppalaneni N. B. Bloom Filter: A Data Structure for Networking, Big Data, Machine Learning, Cloud Computing, Internet of Things, Bioinformatics, and Beyond. 1st ed. Cambridge: Academic Press, 2023. – 228 p.
3. Рафгарден, Т. Совершенный алгоритм. Графовые алгоритмы и структуры данных / Т. Рафгарден. – СПб.: Питер, 2019. – 256 с.
4. Burton B. H. Space/time trade-offs in hash coding with allowable errors // Communications of the ACM T. – 1970. – Vol. 13. – No. 7. – P. 426.
5. Fan L., Cao P., Almeida J., Broder A. Summary cache: A scalable wide-area Web cache sharing protocol // IEEE/ACM Transactions on Networking. – 2000. – Vol. 8. – No. 3. – P. 281-293.
6. Zhang Y., Wu Y. and Yang G. Droplet: A distributed solution of data deduplication // ACM/IEEE 13th International Conference on Grid Computing. – 2012. – P. 114-121.
7. Patgiri R., Nayak S., Borgohain S. K. Role of Bloom Filter in Big Data Research: A Survey // International Journal of Advanced Computer Science and Applications (IJACSA). – 2018.– Vol. 9. – No 11. – P. 650-658.
8. Сайт конспектов лекций университета ИТМО: сайт. – Санкт-Петербург, 2023. – URL: https://neerc.ifmo.ru (дата обращения: 21.04.2024). – Текст: электронный.