

# SET-5 (A3)

Михайлов Артём, БПИ-243

3 февраля 2026 г.

## Задача A3. HyperMegaLogLog Pro Max++

Весь исходный код, все исходные данные, использованные при тестировании и анализе, а также данный отчёт по оценке и анализу точности реализованных алгоритмов находятся в моём репозитории на [GitHub](#). Но помимо этого, все коды и графики есть в этом отчёте.

В этом задании исследуем алгоритм HyperLogLog, который находит оценку  $N_t$  для частотного момента  $F_0^t$  - количества уникальных объектов в потоке на момент времени  $t$ . Чтобы провести успешное исследование, выполним несколько важных этапов.

### Этап 1. Создание инфраструктуры

- Класс RandomStreamGen для генерации потока данных S.

```
class RandomStreamGen {
    size_t stream_size;
    size_t ready_strings_count;
    std::mt19937 rsg;
    std::string generateOneString() {
        static const char charset[] = "0123456789
ABCDEFHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz -";
        size_t alphabet_size = 62;
        std::uniform_int_distribution dist_len(1, 30);
        std::uniform_int_distribution<> dist_char(0, alphabet_size);
        size_t length = dist_len(rsg);
        std::string result;
        result.reserve(length);
        for (size_t i = 0; i < length; ++i) {
            result += charset[dist_char(rsg)];
        }
        return result;
    }
public:
    explicit RandomStreamGen(size_t size) {
        stream_size = size;
        ready_strings_count = 0;
        std::random_device rd;
        rsg.seed(rd());
    }
    std::vector<std::string> nextPortion(double percent) {
        std::vector<std::string> portion;
        auto count_to_generate = static_cast<size_t>(stream_size * percent);

        if (ready_strings_count + count_to_generate > stream_size) {
            count_to_generate = stream_size - ready_strings_count;
        }
        if (count_to_generate == 0) {
            return portion;
        }
        for (size_t i = 0; i < count_to_generate; ++i) {
            portion.push_back(generateOneString());
        }
        ready_strings_count += count_to_generate;
        return portion;
    }
    bool isFinished() const { return ready_strings_count >= stream_size; }
};
```

Этот класс генерирует поток строк длиной до 30 символов из латинских букв (строчных и прописных), цифр 0-9 и тире. Кроме того он реализует возможность разбиения потока на части (с шагом 5%, 10%...) для моделирования момента времени t.

- Класс HashFuncGen для генерации хеш-функции  $h : U \rightarrow M = 2^{32}$ , где U - это множество строк, из которых формируется поток данных S.

```
class HashFuncGen {
    std::mt19937 rhg;

public:
    HashFuncGen() {
        std::random_device rd;
        rhg.seed(rd());
    }
    auto generateNewHashFunc() {
        std::uniform_int_distribution<unsigned int> dist(1000, 4294967295);
        unsigned int A = dist(rhg);
        A = A | 1;
        return [A](const std::string &s) -> unsigned int {
            unsigned int hash = 0;
            for (char c: s) {
                hash = hash * A + static_cast<unsigned int>(c);
            }
            return hash;
        };
    }
};
```

Хеш-функция даёт приблизительно равномерное распределение значений. Я использовал здесь полиномиальную хеш-функцию, которая опирается на код символа, его положение в строке и рандомных коэффициент A.

## Этап 2. Реализация и оценка точности стандартного алгоритма HyperLogLog

Теперь реализуем вероятностный алгоритм HyperLogLog для вычисления оценки  $N_t$ .

```
class HyperLogLog {
    size_t B;
    size_t m;
    std::vector<int> subStreams;
    std::function<unsigned int(const std::string &) > h;

public:
    HyperLogLog(size_t b, HashFuncGen &hasher) {
        B = b;
        m = 1 << B;
        subStreams = std::vector<int>(m, 0);
        h = hasher.generateNewHashFunc();
    }
    void workWithStrings(std::vector<std::string> &myStrings) {
        for (std::string str: myStrings) {
            unsigned int hash = h(str);
            size_t index = hash >> (32 - B);
            unsigned int rank = std::countl_zero(hash << B) + 1;
            if (rank > subStreams[index]) {
                subStreams[index] = rank;
            }
        }
    }
    double get_alpha(size_t m_val) {
        if (m_val == 2)
            return 0.3512;
        if (m_val == 4)
            return 0.5324;
        if (m_val == 8)
            return 0.6355;
        if (m_val == 16)
            return 0.673;
```

```

    if (m_val == 32)
        return 0.697;
    if (m_val == 64)
        return 0.709;
    if (m_val >= 128)
        return 0.7213 / (1.0 + 1.079 / m_val);
    return 0.673;
}
double approx() {
    double sum = 0;
    for (int i = 0; i < m; i++) {
        sum += std::pow(2.0, -subStreams[i]);
    }
    double res = get_alpha(m) * m * m / sum;
    if (res < 2.5 * m) {
        int v = 0;
        for (int t: subStreams) {
            if (t == 0) {
                v++;
            }
        }
        if (v == 0) {
            return res;
        }
        res = m * std::log(static_cast<double>(m) / v);
    }
    return res;
}
};


```

Для тестирования HyperLogLog я выбрал 3 разных значения  $B$  (6, 10, 14), чтобы сделать вывод о том, какое значение даёт более точные результаты.

Выбор этих значений обусловлен тем, что они покрывают разные диапазоны потребления памяти: от минимального ( $2^6 = 64$  байт) до среднего ( $2^{10} \approx 4$  КБ) и относительно большого для задачи ( $2^{14} \approx 64$  КБ), что позволит явно увидеть зависимость точности от выделенной памяти.

Предполагается, что хеш-функция распределяет значения равномерно, поэтому оставшиеся 32-Бит будут вести себя как случайные величины, что необходимо для корректной оценки ранга. Сравнение полученных результатов будет представлено в Этапе 3.

Кроме того, написал функцию для вычисления точного числа  $F_0^t$  уникальных объектов в потоке.

```

class ExactCounter {
    std::unordered_set<std::string> unique_elements;

public:
    void add(const std::vector<std::string> &portion) {
        for (const auto &str: portion) {
            unique_elements.insert(str);
        }
    }
    size_t get() const { return unique_elements.size(); }
    void clear() { unique_elements.clear(); }
};


```

Здесь я использовал `std::unordered_set` для быстрого поиска и для того, чтобы хранить только уникальные значения.

Теперь от нас требуется сгенерировать несколько потоков и:

1. Для каждой выбранной части каждого потока вычислить точное число  $F_0^t$  уникальных объектов и оценку  $N_t$ . (это эксперимент 1)

Вот написанный мной код для проведения этого тестирования:

```

int main() {
    for (int b = 6; b <= 14; b += 4) {
        std::cout << "B = " << b << "\n";
        RandomStreamGen stream(1000000);
        HashFuncGen hasher;
        HyperLogLog hyper_log_log(b, hasher);
        ExactCounter real_counter;


```

```

    double percent_counter = 0.0;
    double my_working_percent = 0.05;
    while (!stream.isFinished()) {
        auto data = stream.nextPortion(my_working_percent);
        percent_counter += my_working_percent;
        hyper_log_log.workWithStrings(data);
        real_counter.add(data);
        double approx_res = hyper_log_log.approx();
        size_t realResult = real_counter.get();
        std::cout << percent_counter * 100 << "%" << " " << realResult << " " <<
approx_res << "\n";
    }
    std::cout << "-----\n";
}
return 0;

```

То есть взята длина потока 1000000 строк, которые поступают нам порциями по 5%. Для каждого из В создаём свой поток и прогоняем для получения собственных значений.

2. Определить выборочные статистики для всех сгенерированных потоков: среднее значение оценки  $E(N_t)$ , а также стандартное отклонение (это эксперимент 2)  $\sigma_{N_t}$ , которое я впоследствии буду называть просто sigma.

Вот написанный мной код для этого тестирования:

```

std::pair<double, double> statsCounter(std::vector<double> &results) {
    size_t n = results.size();
    double sum = 0.0;
    for (double t: results) {
        sum += t;
    }
    double E = sum / static_cast<double>(n);
    double sumDelux = 0.0;
    for (double t: results) {
        sumDelux += (t - E) * (t - E);
    }
    double sigma = std::sqrt(sumDelux / (n - 1));
    return std::make_pair(E, sigma);
}

int main() {
    for (int b = 6; b <= 14; b += 4) {
        std::cout << "B = " << b << "\n";
        std::vector<std::vector<double>> statistics_data(101);
        for (int i = 0; i < 100; i++) {
            RandomStreamGen stream(1000000);
            HashFuncGen hasher;
            HyperLogLog hyper_log_log(b, hasher);
            std::unordered_set<std::string> global_set;
            double percent_counter = 0.0;
            double my_working_percent = 0.05;
            while (!stream.isFinished()) {
                auto data = stream.nextPortion(my_working_percent);
                percent_counter += my_working_percent;
                hyper_log_log.workWithStrings(data);
                double approx_res = hyper_log_log.approx();
                int index = std::round(percent_counter * 100);
                statistics_data[index].push_back(approx_res);
            }
        }
        std::vector<double> Es;
        std::vector<double> sigmas;
        for (int i = 5; i <= 100; i += 5) {
            std::pair<double, double> stats = statsCounter(statistics_data[i]);
            Es.push_back(stats.first);
            sigmas.push_back(stats.second);
        }
        for (int i = 0; i < 20; i++) {
            std::cout << 5*(i+1) << "%" << "E = " << Es[i] << " ; sigma = " << sigmas[i] << "\n";
        }
    }
}

```

```

    }
    std::cout << "-----\n";
}
return 0;
}

```

Здесь для каждого из значение В взято по 100 потоков длиной 100000 строк, которые точно так же поступают порциями по 5% от общего количества.

Результаты работы тестирующего кода можно найти в папке results/ в соответствующих файлах на Гитхабе. Соответственно results\_1.txt - 1 эксперимент, а results\_2.txt - 2 эксперимент.

Теперь на основе полученных данных построим графики для визуализации результатов анализа точности HyperLogLog.

### 1. График №1 сравнения оценки $N_t$ и $F_0^t$ . (графики 1-3)

Из графиков видно, что при  $B = 6$  оценка HyperLogLog сильно колеблется и в итоге отличается более, чем на 100000 от истинного значения уникальных элементов. При  $B = 10$  оценка намного более точная, однако всё ещё присутствуют незначительные колебания, а в итоге расхождение с истинным значением составляет примерно меньше 50000. При  $B = 14$  получаем наилучшую оценку. Колебаний нет вообще, а расхождение с истинным значением составляет около 20000, что на миллионе строк - очень хороший результат.

### 2. График №2 статистик оценки (линия $E(N_t)$ и область неопределённости в зависимости от $\sigma_{N_t}$ ). (графики 4-6)

Из графиков видно, что при  $B = 6$  дисперсия сильно увеличивается с ростом количества обработанных строк, в итоге достигая значений больше 10000 (дисперсия огромная). При  $B = 10$  дисперсия значительно уменьшается, и не так сильно растёт (в конце достигает около 3000), но при  $B = 14$  дисперсия минимальна (около 600), а значит среди рассмотренных  $B$  при значении 14 получаем минимальный разброс получаемых значений, а значит наибольшую точность.

Я построил график каждого типа для каждого из  $B$  - всего получилось 6 штук. Их можно посмотреть в конце этого файла. Кроме того все графики лежат в папке graphics на Гитхаб.

## Этап 3. Анализ результатов работы стандартного алгоритма HyperLogLog

Теперь перейдём к сравнению практических результатов работы алгоритма с теоретическими оценками для него.

### 1. Точность разработанного алгоритма.

Для анализа возьмём данные обработки 100% потока для каждого из  $B$ .

Рассчитаем относительную стандартную ошибку:  $O = \frac{\text{sigma}}{E}$

А также оценим, насколько наша практическая ошибка укладывается в рамки теоретических отклонений:

$$O_1 = \frac{1.04}{\sqrt{2^B}}$$

и

$$O_2 = \frac{1.3}{\sqrt{2^B}}$$

- $B = 6$

$$O = \frac{\text{sigma}}{E} = \frac{12404.2}{94965.8} = 0.1306$$

$$O_1 = \frac{1.04}{\sqrt{2^6}} = 0.13$$

$$O_2 = \frac{1.3}{\sqrt{2^6}} = 0.1625$$

Получаем, что при  $B = 6$  относительная стандартная ошибка составляет немного больше, чем 0.13, что практически вписывается в рамки теоретического отклонения  $O_1$  и с запасом укладывается в более консервативную оценку  $O_2$ .

- $B = 10$

$$O = \frac{\text{sigma}}{E} = \frac{3098.85}{92412.6} = 0.0335$$

$$O_1 = \frac{1.04}{\sqrt{2^{10}}} = 0.0325$$

$$O_2 = \frac{1.3}{\sqrt{2^{10}}} = 0.0406$$

Здесь получаем, что при  $B = 10$  относительная стандартная ошибка составляет примерно 0.0335, что опять же практически вписывается в рамки теоретического отклонения  $O_1$  и с запасом укладывается в более консервативную оценку  $O_2$ .

- $B = 14$
- $$O = \frac{\text{sigma}}{E} = \frac{623.17}{92213.8} = 0.00676$$
- $$O_1 = \frac{1.04}{\sqrt{2^{14}}} = 0.008125$$
- $$O_2 = \frac{1.3}{\sqrt{2^{14}}} = 0.01015$$

Здесь получаем, что при  $B = 14$  относительная стандартная ошибка составляет примерно 0.0068, что с запасом вписывается в рамки теоретических отклонений  $O_1$  и  $O_2$ . Практическая точность превосходит даже оптимистичную теоретическую оценку, что говорит о высоком качестве распределения хеш-функции.

## 2. Стабильность оценки, получаемой алгоритмом.

Здесь проанализируем параметр sigma, получаемый после обработки 100% потока строк при разных B:

- $B = 6$   
sigma = 12404.2
- $B = 10$   
sigma = 3098.85
- $B = 14$   
sigma = 623.17

Заметим, что с ростом B значение в sigma заметно уменьшается: более, чем на 75% при переходе с  $B = 6$  на  $B = 10$ , а также ещё на более чем 80% при переходе с  $B = 10$  на  $B = 14$ .

Получаем, что среди проведённых экспериментов оценка наиболее стабильна (дисперсия минимальна) при  $B = 14$ . Это означает, что алгоритм становится более стабильным и предсказуемым. При малых B оценка сильно варьируется от запуска к запуску.

## 3. Эффективность выбранных констант.

В этом алгоритме выбранной константой является B. Было проведено 3 эксперимента с разными значениями B: 6, 10, 14. Для каждого из этих значений мы провели большое количество запусков алгоритма для получения более точной и справедливой оценки работы алгоритма при таком значении параметра B.

Если анализировать влияние константы на результаты работы алгоритма, то можно с уверенностью сказать, что с увеличением B результаты работы становились всё точнее, относительная стандартная ошибка достигла значения 0.00676, что укладывается в рамки теоретических отклонений. Кроме того, стабильность оценки также сильно возросла (на 95% от начального значения) при росте B с 6 до 14.

Анализируя эффективность работы алгоритма при увеличении B можно сказать, что при больших B работа алгоритма занимает значительно больше памяти - затраты памяти растут экспоненциально ( $2^B$ ), однако в контексте современных вычислительных систем, это ничтожно мало по сравнению с тем, что точность работы алгоритма возрастает в разы.

## Этап 4. Усовершенствование HyperLogLog

Теперь, когда мы проверили эффективность оригинального алгоритма, предложим модификацию для него для достижения лучших результатов  $N_t$ .

Вот код для HyperMegaLogLogProMax:

```
class HyperMegaLogLogProMax {
    size_t B;
    size_t m;
    bool isSparse;
    std::vector<uint8_t> subStreams;
    std::map<int, uint8_t> sparseStreams;
    std::function<unsigned int(const std::string &)> h;

public:
    HyperMegaLogLogProMax(size_t b, HashFuncGen &hasher) {
        B = b;
        m = 1 << B;
        isSparse = true;
        h = hasher.generateNewHashFunc();
    }
    void workWithStrings(std::vector<std::string> &myStrings) {
        for (std::string str: myStrings) {
            if (sparseStreams.size() > m / 16) {
                isSparse = false;
            }
        }
    }
}
```

```

    subStreams.resize(m, 0);
    for (auto const &[index, value]: sparseStreams) {
        subStreams[index] = value;
    }
    sparseStreams.clear();
}
unsigned int hash = h(str);
size_t index = hash >> (32 - B);
uint8_t rank = std::countl_zero(hash << B) + 1;
if (isSparse) {
    if (rank > sparseStreams[index]) {
        sparseStreams[index] = rank;
    }
} else {
    if (rank > subStreams[index]) {
        subStreams[index] = rank;
    }
}
}
}

double get_alpha(size_t m_val) {
    if (m_val == 2)
        return 0.3512;
    if (m_val == 4)
        return 0.5324;
    if (m_val == 8)
        return 0.6355;
    if (m_val == 16)
        return 0.673;
    if (m_val == 32)
        return 0.697;
    if (m_val == 64)
        return 0.709;
    if (m_val >= 128)
        return 0.7213 / (1.0 + 1.079 / m_val);
    return 0.673;
}

double approx() {
    double sum = 0;
    if (isSparse) {
        for (auto const &[index, value]: sparseStreams) {
            sum += std::pow(2.0, -value);
        }
        size_t empty_count = m - sparseStreams.size();
        sum += empty_count * 1.0;
    } else {
        for (int i = 0; i < m; i++) {
            sum += std::pow(2.0, -subStreams[i]);
        }
    }
    double res = get_alpha(m) * m * m / sum;
    if (res < 2.5 * m) {
        int v = 0;
        if (isSparse) {
            int non_zeros = 0;
            for (auto const &[index, value]: sparseStreams) {
                if (value > 0)
                    non_zeros++;
            }
            v = m - non_zeros;
        } else {
            for (uint8_t t: subStreams) {
                if (t == 0)
                    v++;
            }
        }
        if (v == 0) {
            return res;
        }
    }
}

```

```

        res = m * std::log(static_cast<double>(m) / v);
    }
    return res;
}

```

Здесь в качестве модификации было предложено 2 решения для снижения потребления памяти:

- Использовать `std::vector<uint8_t>` вместо `std::vector<int>`. Так как мы храним в векторе только ранги, значения которых вариируются от 0 до 32 (в зависимости от  $B$ ), а значит, что нам не нужен тип `int`, а хватит даже `uint8_t`
- Поначалу, пока количество поступивших строк  $\leq \frac{m}{16}$  мы используем не вектор, а `std::map<int, uint8_t>`, тем самым мы не создаём сразу массив на  $m$  элементов, а создаем map и в него отдельно для каждого индекса записываем его ранг, который занимает меньше памяти. Использование `std::map` оправдано только на малых объемах данных (Sparse mode), где накладные расходы на структуру дерева перекрываются выгодой от того, что мы не храним тысячи пустых нулей.

После того, как количество вставленных элементов превысит  $\frac{m}{16}$  мы вставляем все уже обработанные элементы в `std::vector<uint8_t>`, и дальнее работа алгоритма продолжается в том же порядке, что и в стандартном HyperLogLog.

- Используем  $B=14$ , так как поняли, что это даёт нам лучший результат по точности по сравнению с другими разобранными  $B$ .

Теперь для того, чтобы подтвердить, что модифицированный алгоритм действительно потребляет меньше памяти, рассчитаем потребление памяти для стандартного и доработанного алгоритма.

## 1. Потребление памяти стандартным алгоритмом.

Поскольку  $B = 14$ , то  $m = 2^B = 2^{14} = 16384$ .

Значит у нас с самого начала работы алгоритма будет создан `std::vector<int>(m, 0)`.

`int` весит 4 байта, а значит получаем:

Занятая память = 4 байта \* 16384 = 65536 байт = 64КБ - с первой секунды работы алгоритма.

## 2. Потребление памяти модифицированным алгоритмом.

До переключения на `std::vector<uint8_t>` мы используем `std::map<int, uint8_t>`.

Я посмотрел, один узел в таком случае занимает 40-48 байт. Возьмём 48 байт для надёжности. Поскольку мы добавляем элемент в map только когда он приходит к нам, то занимать в памяти будет он будет ( $48 * \text{кол-во элементов}$ ) байт.

0 элементов = 0 байт

10 элементов = 480 байт

1000 элементов = 48000 байт

Понимаем, что переход осуществляется при кол-ве элементов  $> \frac{m}{16} = \frac{16384}{16} = 1024 = 48\text{KB}$

Значит прямо перед самым переключением std::map занимает всё ещё меньше места, чем с самого начала занимал `std::vector<int>` в стандартном алгоритме.

После переключения на `std::vector<uint8_t>` мы переносим все ранги в него из map, а map после этого очищаем.

`uint_8` весит 1 байт, а значит получаем:

Занятая память = 1 байт \* 16384 = 16384 байт = 16КБ - после переключения на `std::vector<uint_8>`.

Таким образом, даже в самом худшем случае модифицированный алгоритм занимает 48КБ, что составляет 75% от первоначального использования памяти стандартным алгоритмом(64КБ). Однако, этот худший случай длится лишь одно мгновение перед переходом. После него потребление мгновенно падает до 16 КБ и остается таким до конца работы, независимо от того, сколько еще придет строк.

Кроме того, я провёл тестирование разработанного алгоритма, повторив этапы 2 и 3.

Результаты работы алгоритма находятся в папке `results/` в соответствующих файлах на Гитхабе. Соответственно `results_improvement_1.txt` - 1 эксперимент, а `results_improvement_2.txt` - 2 эксперимент. Также были построены графики обоих типов на основе полученных данных. (графики 7 и 8 в конце файла, также в папке `graphics` на Гитхаб)

Как видим, точность алгоритма никак не изменилась, так как мы не предпринимали никаких действий для этого. Точность осталась ровно на том же уровне, что и была при  $B = 14$ . Колебания оценки минимальны, сама итоговая оценка отличается менее, чем на 30000, а дисперсия растёт совсем медленно и не достигает даже 700.

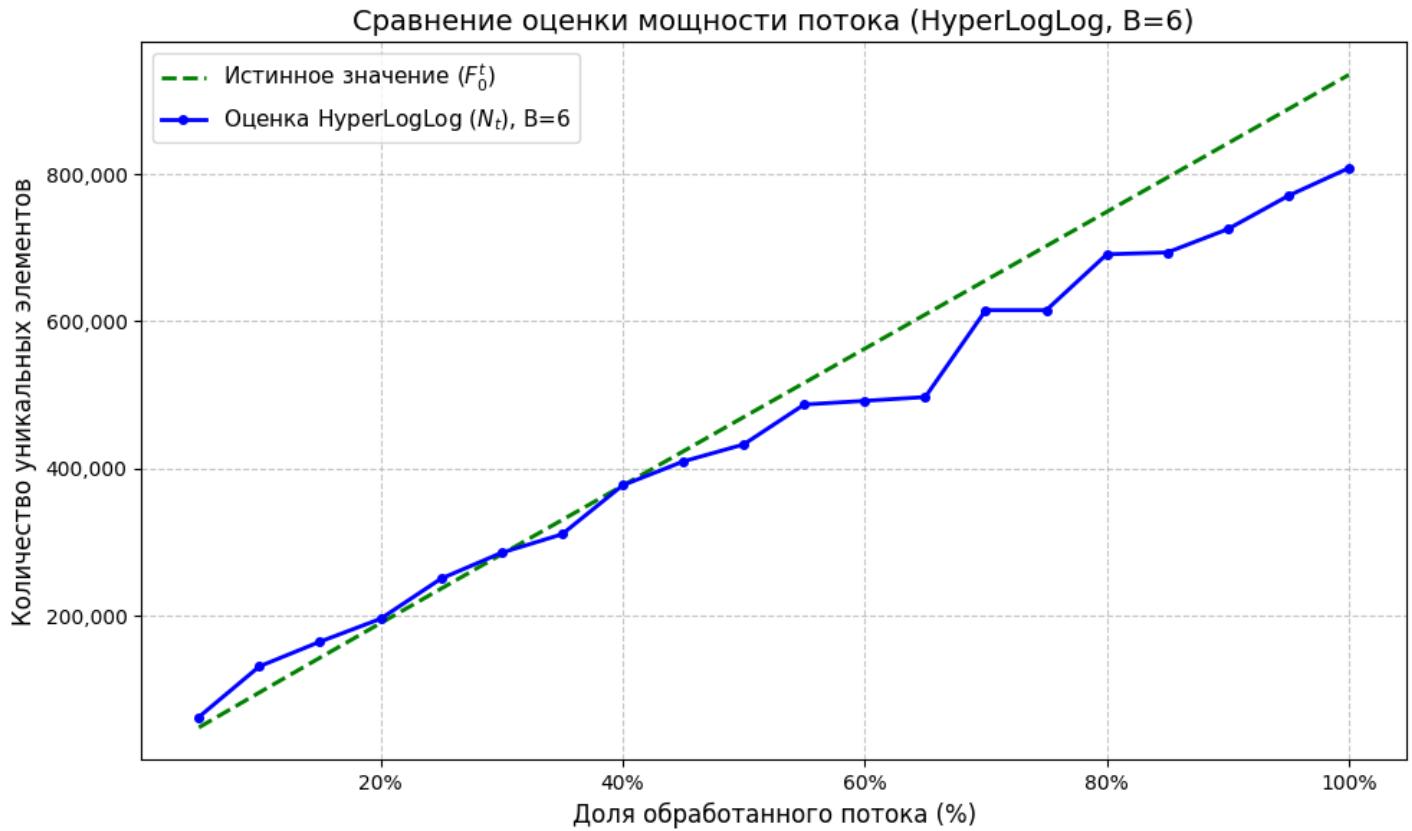


Рис. 1: График-1

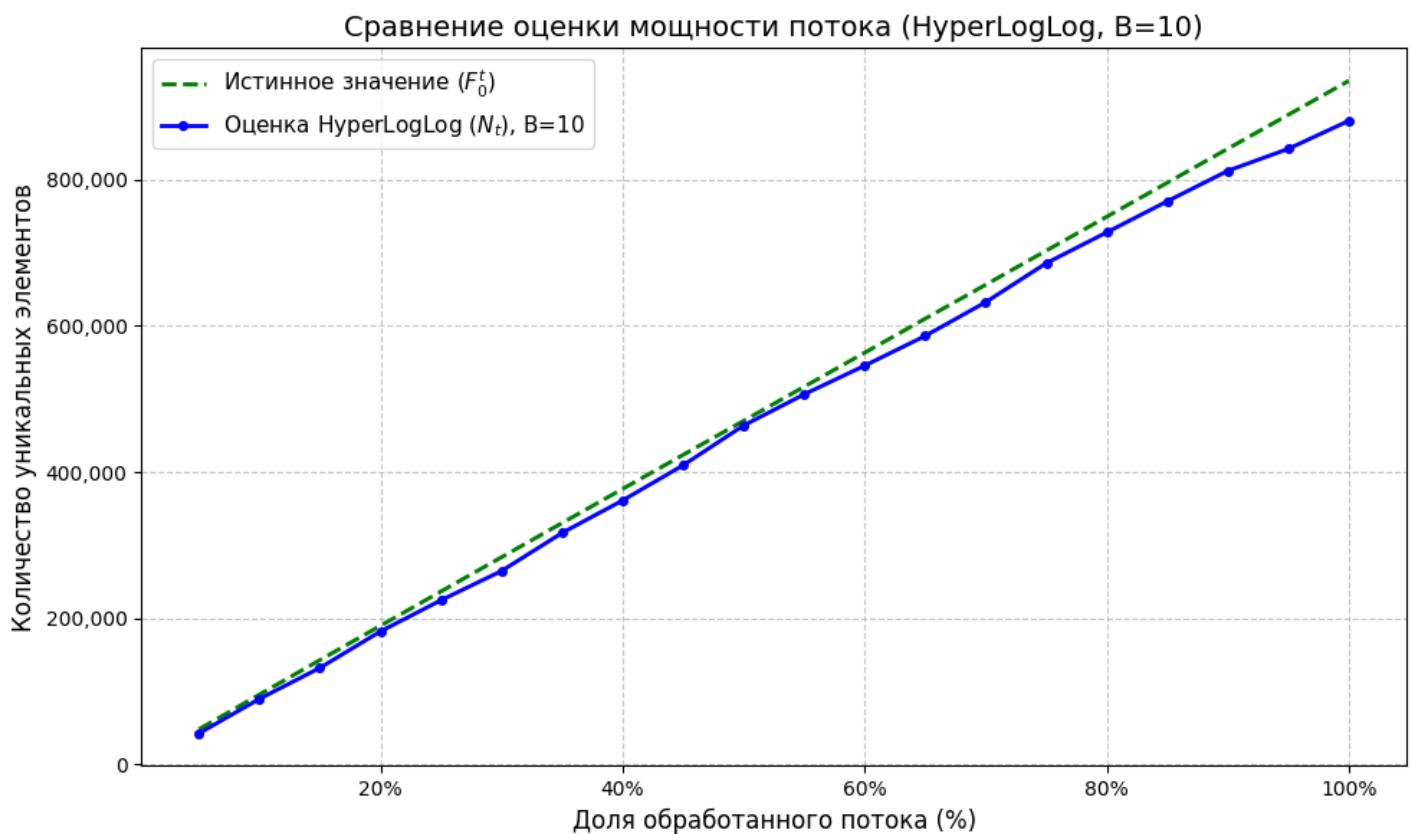


Рис. 2: График-2

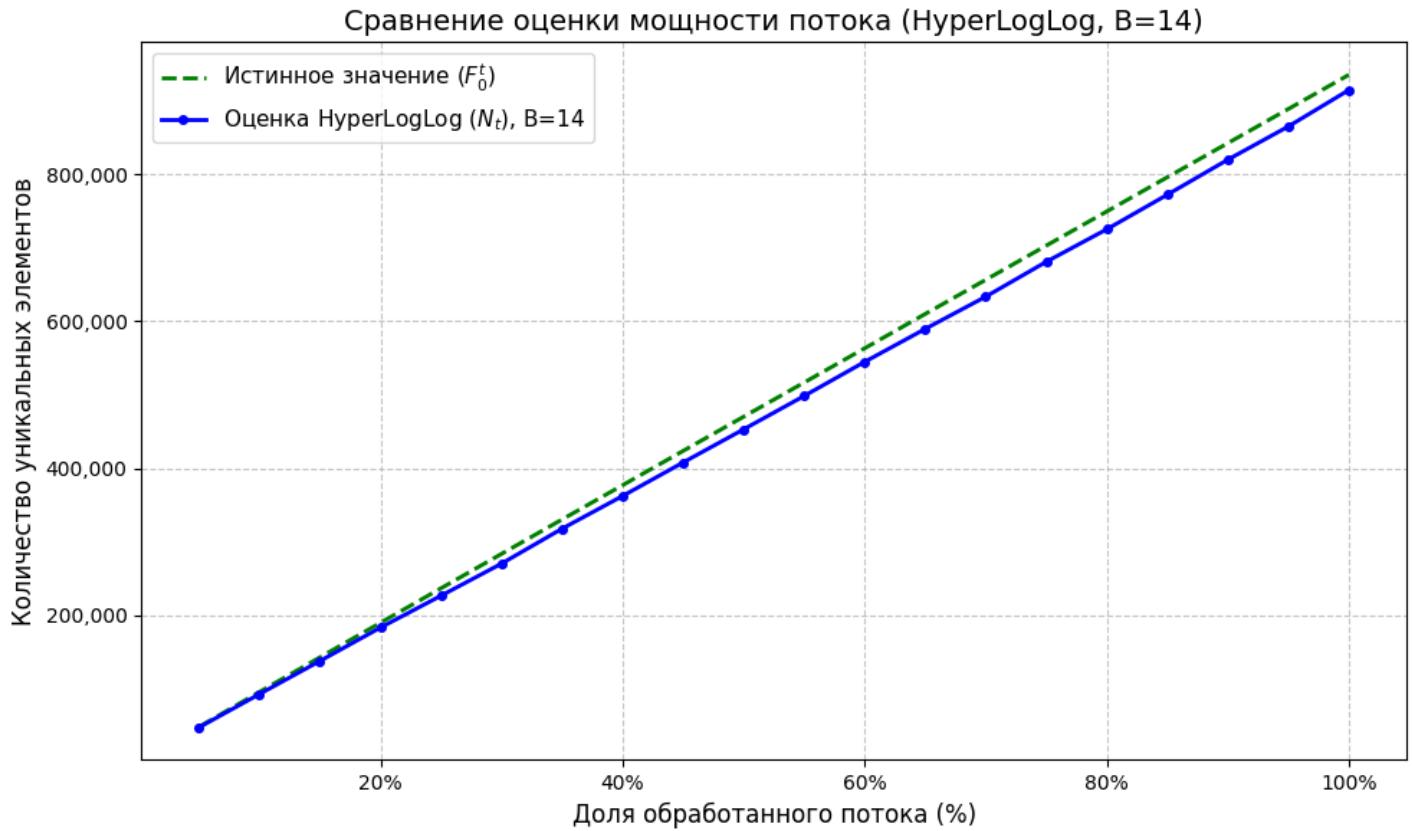


Рис. 3: График-3

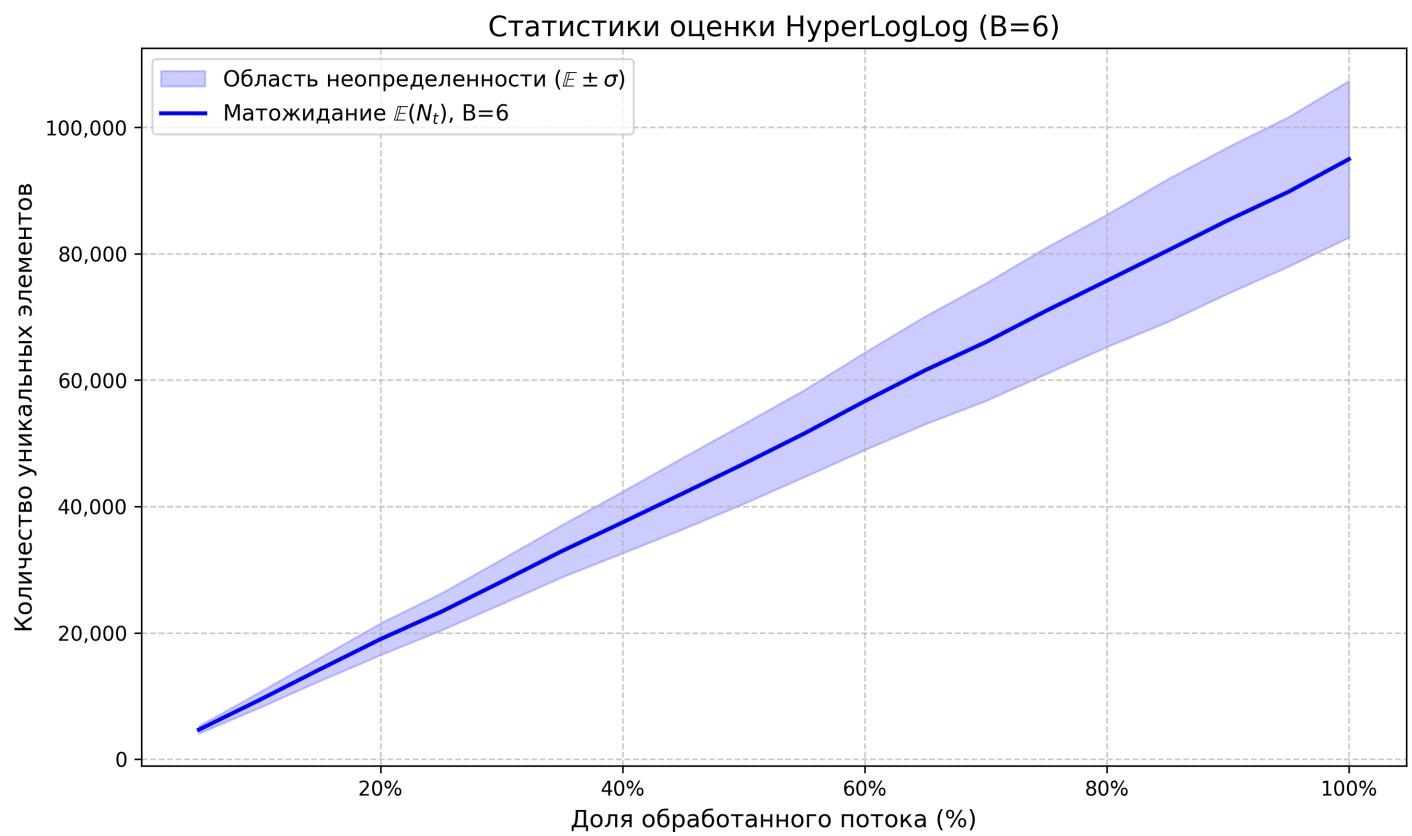


Рис. 4: График-4

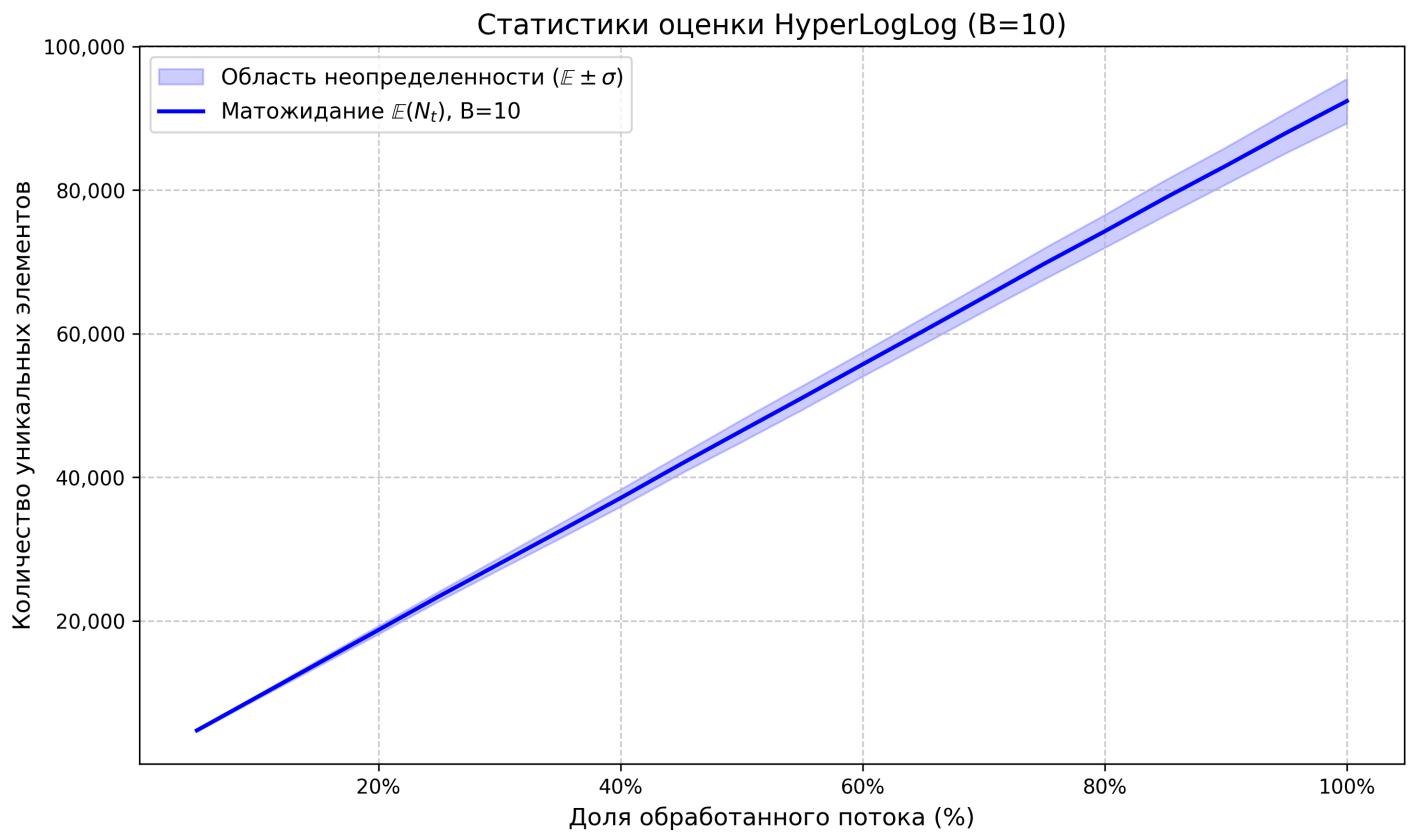


Рис. 5: График-5

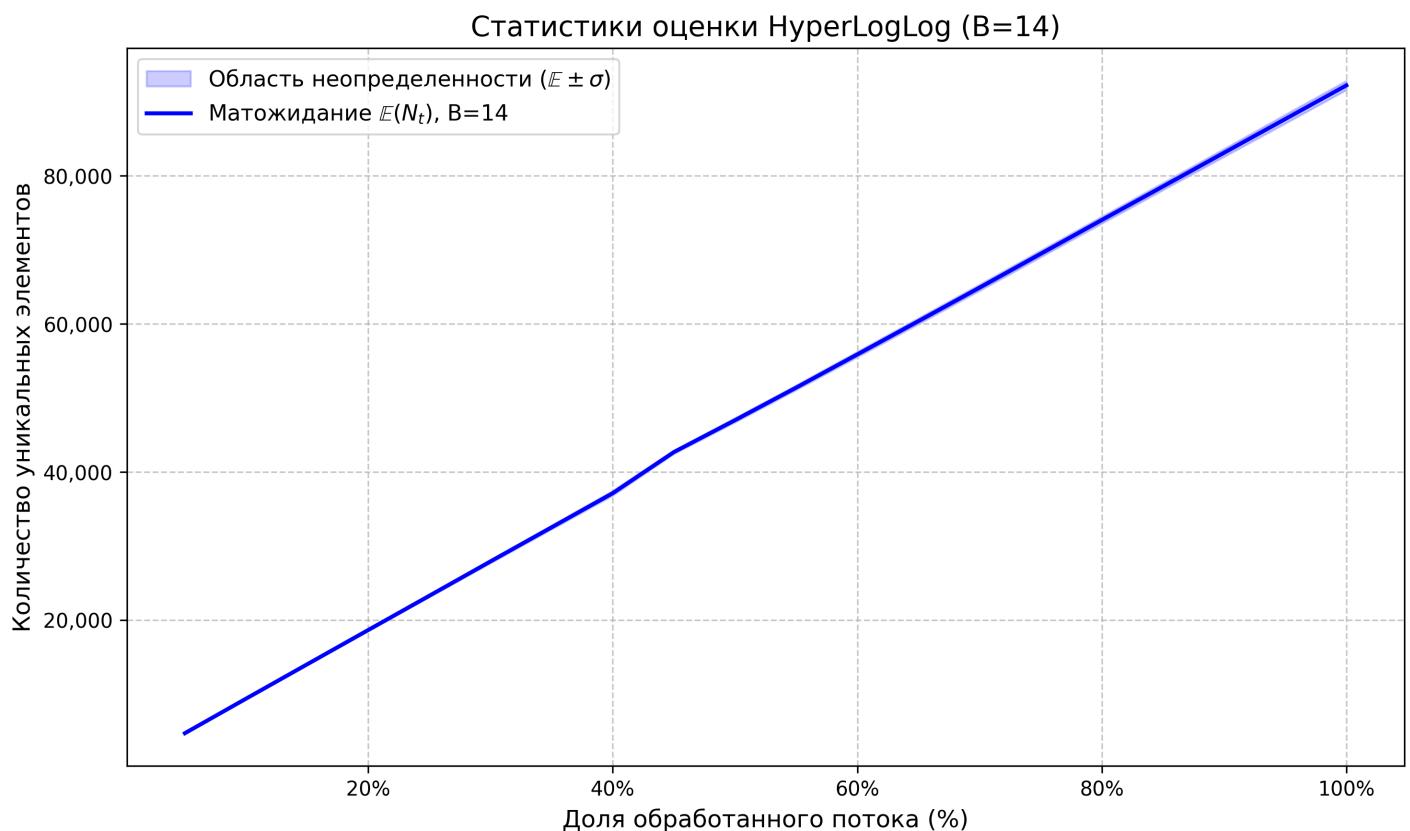


Рис. 6: График-6

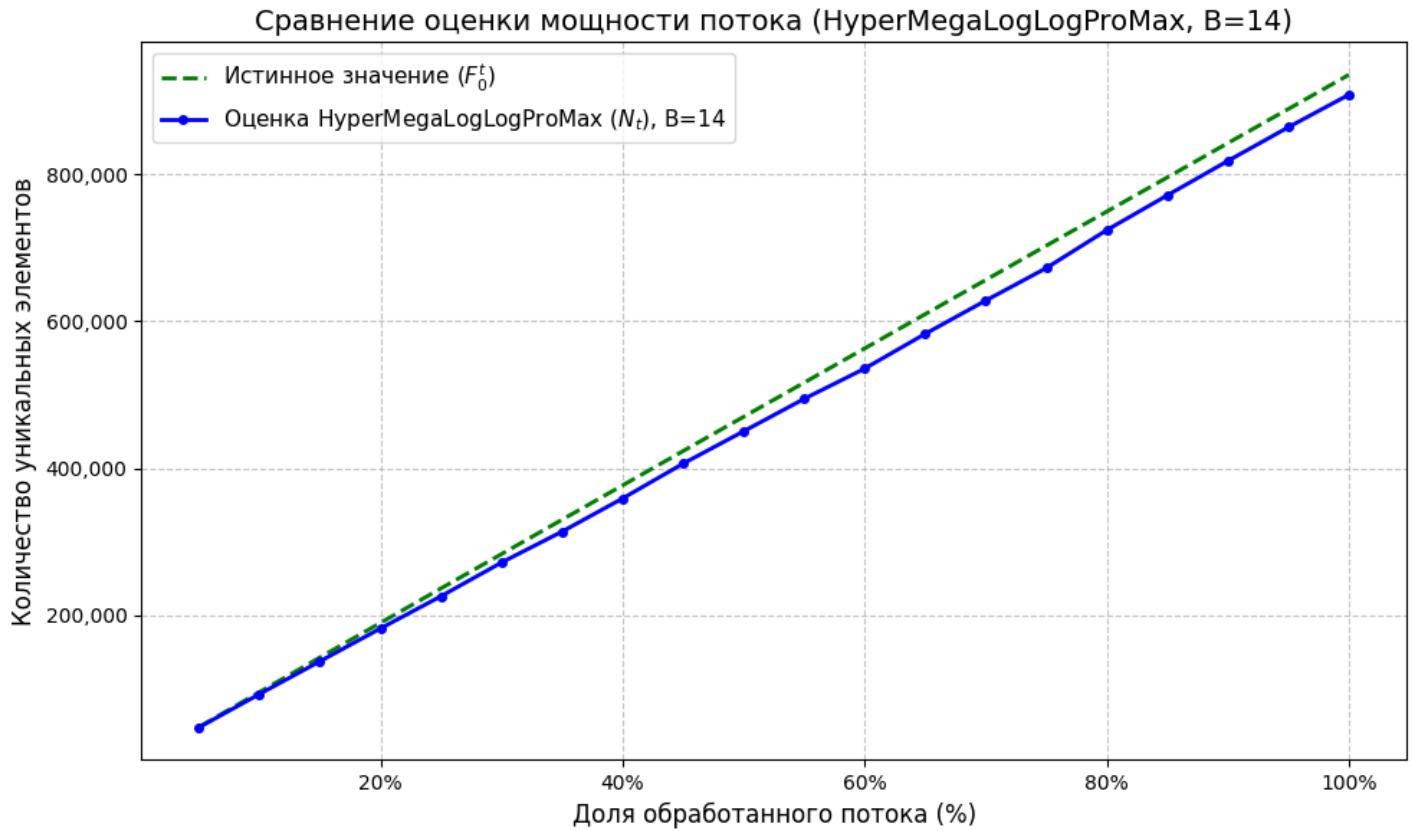


Рис. 7: График-7

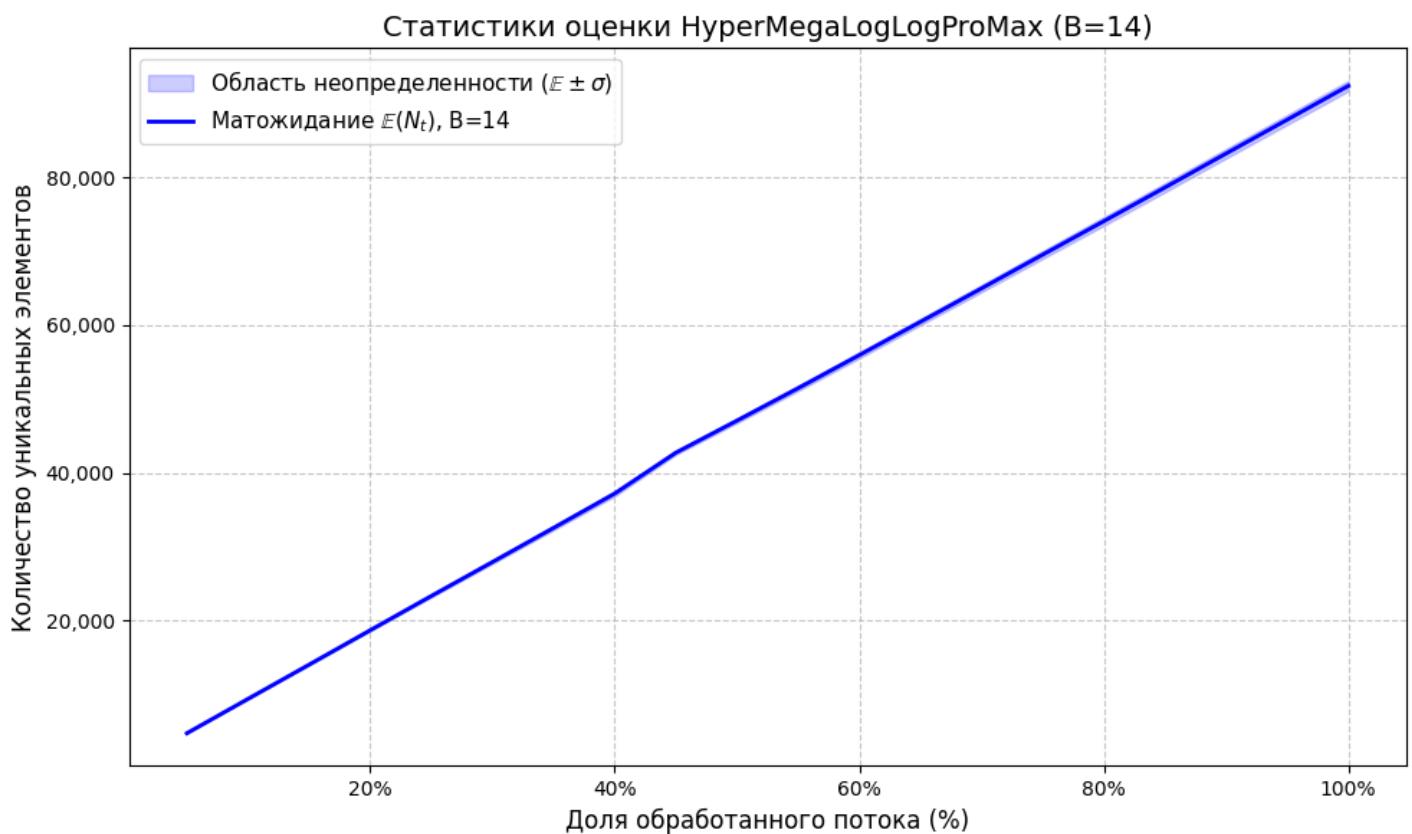


Рис. 8: График-8