Задача A3. HyperMegaLogLog Pro Max++

Исследование алгоритма HyperLogLog

1 Инфраструктура

1.1 Генератор потока `RandomStreamGen`

Для экспериментов нужен генератор случайных строк. Класс `RandomStreamGen` порождает строки длиной от 1 до 30 символов из алфавита: строчные и прописные латинские буквы, цифры 0–9, дефис — итого 63 символа. Генератор использует `std::mt19937_64` с фиксированным seed для воспроизводимости.

Метод `generate(n)` возвращает вектор из n строк. Чтобы разбить поток на части (моделирование моментов t), есть статический метод `getPartitions(total, step_pct)`, который возвращает индексы, соответствующие 5%, 10%, …, 100% потока.

```
class RandomStreamGen {
public:
    RandomStreamGen(uint64_t seed = 42) : rng_(seed) {}

    std::vector<std::string> generate(size_t n) {
        std::vector<std::string> stream;
        stream.reserve(n);
        std::uniform_int_distribution<int> len_dist(1, 30);
        std::uniform_int_distribution<int> char_dist(0, 62);
        for (size_t i = 0; i < n; ++i) {
            int len = len_dist(rng_);
            std::string s(len, 'x');
            for (int j = 0; j < len; ++j)
                s[j] = alphabet_[char_dist(rng_)];
            stream.push_back(std::move(s));
        }
        return stream;
    }

    static std::vector<size_t> getPartitions(
            size_t total, int step_pct) {
        std::vector<size_t> parts;
        for (int pct = step_pct; pct <= 100; pct += step_pct)
            parts.push_back((size_t)(pct / 100.0 * total));
        return parts;
    }

private:
```

```
    std::mt19937_64 rng_;
    const std::string alphabet_ =
        "abcdefghijklmnopqrstuvwxyz"
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        "0123456789-";
};
```

1.2 Генератор хеш-функций HashFuncGen

Хеш-функция $h : U \to \{0, \ldots, 2^{32}-1\}$ реализована на основе MurmurHash3 (32-битная версия) — популярная некриптографическая хеш-функция с хорошим распределением и быстрой работой.

Почему MurmurHash3: (1) распределение хеш-значений близко к равномерному, (2) хорошее перемешивание бит (avalanche effect), (3) можно параметризовать через seed и получить семейство хеш-функций.

```
class HashFuncGen {
public:
    HashFuncGen(uint32_t seed = 0) : seed_(seed) {}

    uint32_t hash(const std::string& key) const {
        const uint8_t* data = (const uint8_t*)key.data();
        int len = (int)key.size();
        uint32_t h1 = seed_;
        const uint32_t c1 = 0xcc9e2d51, c2 = 0x1b873593;

        for (int i = 0; i < len / 4; i++) {
            uint32_t k1 = ((const uint32_t*)data)[i];
            k1 *= c1; k1 = rotl32(k1, 15); k1 *= c2;
            h1 ^= k1; h1 = rotl32(h1, 13);
            h1 = h1 * 5 + 0xe6546b64;
        }
        // ... tail ...
        h1 ^= (uint32_t)len;
        h1 = fmix32(h1);
        return h1;
    }

    static std::vector<HashFuncGen> generateMultiple(
            int count, uint32_t base_seed = 0) {
        std::vector<HashFuncGen> funcs;
        for (int i = 0; i < count; ++i)
            funcs.emplace_back(base_seed + i * 0x9E3779B9u);
        return funcs;
    }

private:
    uint32_t seed_;
};
```

Чтобы получить несколько хеш-функций, берём разные seed'ы, разнесённые на $\varphi \cdot 2^{32}$ (золотое сечение) — так корреляции минимальны.

Тест равномерности

Чтобы убедиться, что хеш работает корректно, проведём χ^2-тест. Хешируем 10^6 уникальных строк (`"str_0"`, `"str_1"`, ...) и распределяем по 256 корзинам (по остатку от деления). При равномерном распределении ожидаем ≈ 3906 строк на корзину.

Результат: $\chi^2 = 259.20$ при 255 степенях свободы. Критическое значение на уровне $\alpha = 0.05$ — примерно 293.25. Раз $259.20 < 293.25$, гипотеза о равномерности не отвергается. Хеш-функция распределяет значения равномерно.

Распределение по регистрам (субпотокам)

Отдельно проверим, как первые B бит хеша (индекс регистра) распределяются для разных B. Хешируем 10^6 уникальных строк, считаем попадания в каждый из $m = 2^B$ регистров и проводим χ^2-тест:

B	m	Ожид.	σ_{counts}	min	max	χ^2 (результат)
4	16	62500	189.6	62035	62796	9.20 (PASS)
8	256	3906	61.3	3760	4079	245.95 (PASS)
10	1024	977	30.9	883	1069	1000.67 (PASS)
14	16384	61	7.8	34	100	16498.59 (PASS)

Для всех B тест проходит: элементы распределяются по регистрам равномерно. Значит, выбранная хеш-функция корректно разбивает поток на m субпотоков, и можно переходить к HyperLogLog.

2 Стандартный HyperLogLog

2.1 Описание алгоритма

HyperLogLog оценивает число уникальных элементов F_0^t в потоке данных, используя лишь $O(m)$ памяти, где $m = 2^B$ — число регистров.

Работает так. Каждый элемент хешируется в 32-битное число. Первые B бит определяют номер регистра j, а по оставшимся $32 - B$ битам вычисляется $\rho(w)$ — позиция первого единичного бита (считая с 1). В регистре $M[j]$ хранится максимум всех наблюдённых $\rho(w)$.

Оценка строится как:

$$N_t = \alpha_m \cdot m^2 \cdot \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1},$$

где α_m — корректирующая константа:

$$\alpha_m = \begin{cases} 0.673, & m = 16, \\ 0.697, & m = 32, \\ 0.709, & m = 64, \\ \frac{0.7213}{1+1.079/m}, & m \geq 128. \end{cases}$$

Если $N_t \leq 2.5m$ и есть пустые регистры, включается коррекция Linear Counting:

$$N_t = m \cdot \ln \frac{m}{V},$$

где V — число пустых регистров. Для очень больших значений ($N_t > 2^{32}/30$) тоже есть коррекция, но в наших экспериментах до неё дело не доходит.

2.2 Выбор параметра B

Теоретическое стандартное отклонение оценки:

$$\sigma \approx \frac{1.04}{\sqrt{m}} = \frac{1.04}{\sqrt{2^B}}.$$

Чтобы проверить это на практике и подобрать хороший B, запустим эксперименты с $B \in \{4, 8, 10, 14\}$:

B	$m = 2^B$	$\sigma_{\text{теор}}$ $(1.04/\sqrt{m})$	Память (байт)
4	16	26.0%	16
8	256	6.5%	256
10	1024	3.25%	1024
14	16384	0.81%	16384

В основных экспериментах берём $B = 14$ — это стандартное значение в промышленных реализациях (Redis, BigQuery). При 16 КБ памяти получаем ошибку меньше 1%. Но для наглядности покажем результаты и для других B.

2.3 Реализация

```
class HyperLogLog {
public:
    HyperLogLog(int B, const HashFuncGen& hasher)
        : B_(B), m_(1u << B), hasher_(hasher),
          registers_(1u << B, 0)
    {
        if (m_ == 16) alpha_ = 0.673;
        else if (m_ == 32) alpha_ = 0.697;
        else if (m_ == 64) alpha_ = 0.709;
        else alpha_ = 0.7213 / (1.0 + 1.079 / (double)m_);
    }

    void add(const std::string& element) {
        uint32_t h = hasher_.hash(element);
        uint32_t idx = h >> (32 - B_);
        uint32_t w = h << B_;
        int r = rho(w);
        if (r > registers_[idx])
            registers_[idx] = (uint8_t)r;
    }

```

```
    double estimate() const {
        double Z = 0.0;
        for (uint32_t j = 0; j < m_; ++j)
            Z += std::pow(2.0, -(double)registers_[j]);
        double E = alpha_ * m_ * m_ / Z;

        if (E <= 2.5 * m_) {
            int V = 0;
            for (uint32_t j = 0; j < m_; ++j)
                if (registers_[j] == 0) V++;
            if (V > 0)
                E = m_ * std::log((double)m_ / V);
        }
        if (E > 4294967296.0 / 30.0)
            E = -4294967296.0 * std::log(1.0 - E / 4294967296.0);
        return E;
    }

private:
    int B_; uint32_t m_; double alpha_;
    HashFuncGen hasher_;
    std::vector<uint8_t> registers_;

    int rho(uint32_t w) const {
        int max_r = 32 - B_ + 1;
        if (w == 0) return max_r;
        int r = 1;
        while ((w & 0x80000000u) == 0 && r < max_r)
            { w <<= 1; r++; }
        return r;
    }
};
```

Для точного подсчёта F_0^t используем `std::unordered_set<std::string>`.

3 Эмпирические результаты

3.1 Схема эксперимента

Для каждого размера $N \in \{10\,000,\ 100\,000,\ 500\,000\}$ сгенерировали 20 независимых потоков. Каждый поток обрабатываем с шагом 5%: после очередных 5% элементов фиксируем точное F_0^t и оценку N_t, итого 20 замеров на поток.

Хеш-функция одна для всех экспериментов (seed = 42), потоки различаются seed'ами генератора строк.

3.2 График №1: сравнение N_t и F_0^t

На рис. 1 — сравнение для одного потока ($N = 500\,000$) при разных B.

При $B = 4$ оценка сильно скачет — всего 16 регистров, этого мало. При $B = 8$ уже заметно лучше, а при $B = 14$ кривые F_0^t и N_t практически сливаются.

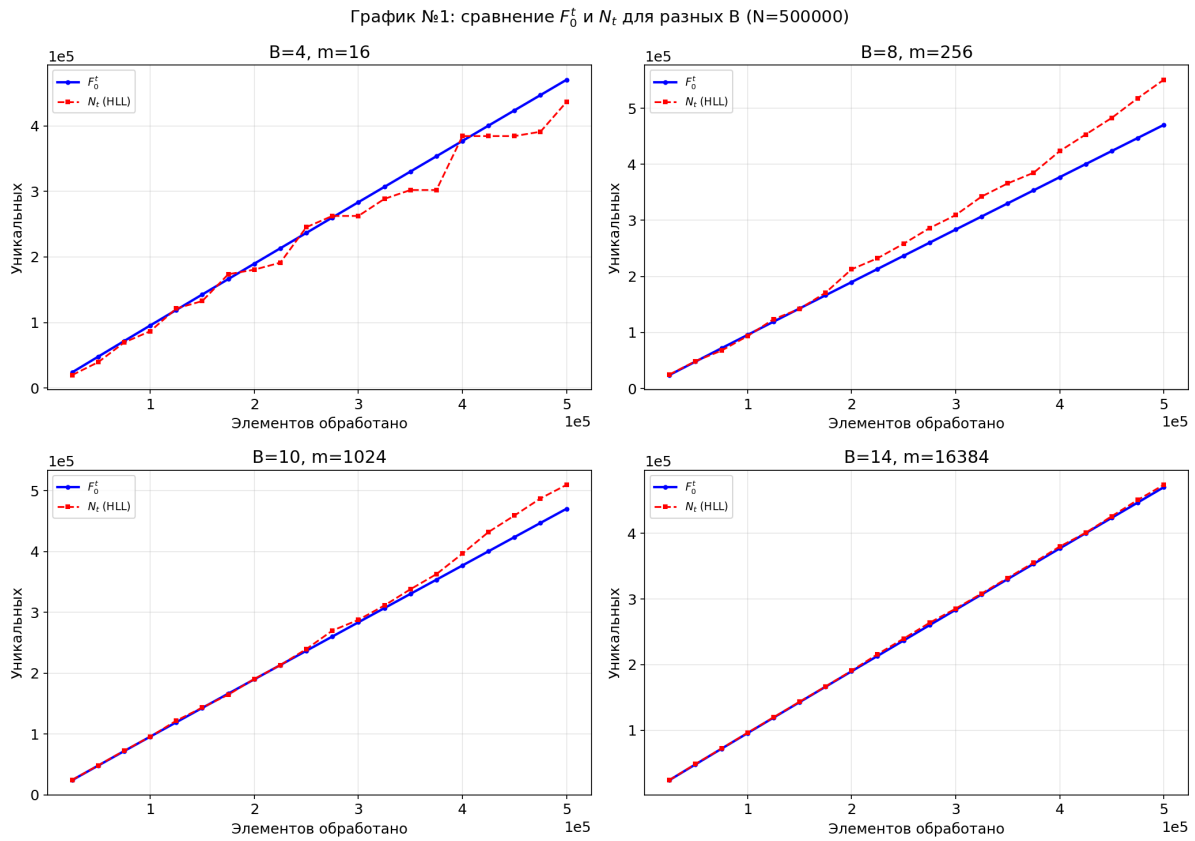

Рис. 1: Сравнение F_0^t (точное) и N_t (HyperLogLog) для одного потока при разных B.

Отдельно покажем крупно график для $B = 14$ (рис. 2).

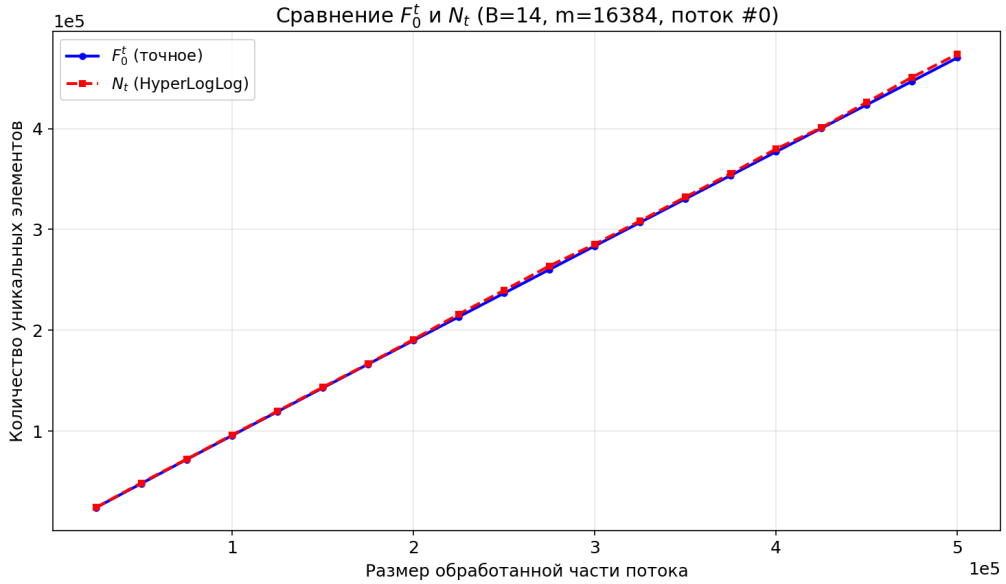

Рис. 2: F_0^t и N_t для $B = 14$, $N = 500\,000$. Кривые почти неразличимы.

3.3 График №2: статистики оценки

Рис. 3: $\mathbb{E}(N_t)$ и область $\mathbb{E}(N_t) \pm \sigma_{N_t}$, усреднённые по 20 потокам.

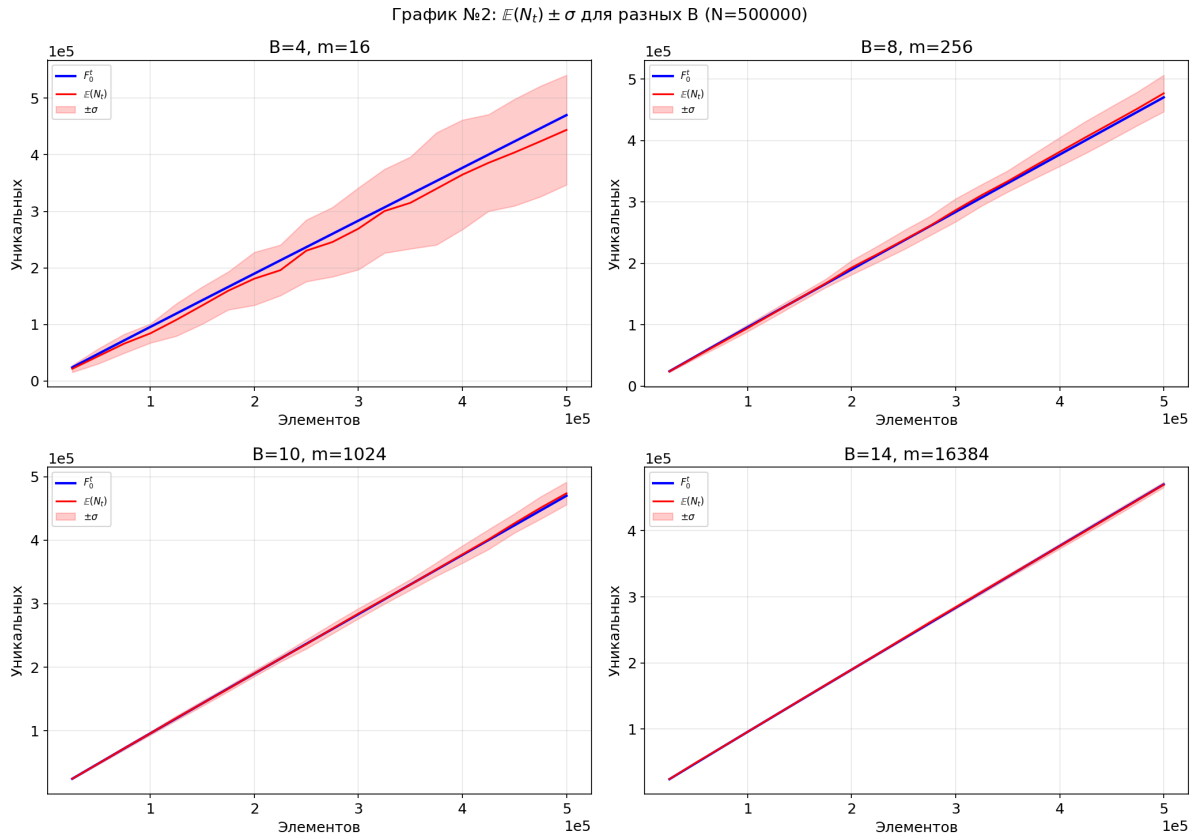

Рис. 3: $\mathbb{E}(N_t) \pm \sigma_{N_t}$ для разных B ($N = 500\,000$, 20 потоков).

При $B = 4$ коридор неопределённости широкий, среднее заметно уходит от истинного значения. С ростом B коридор сужается: при $B = 14$ полоса $\pm\sigma$ едва видна.

4 Анализ результатов

4.1 Точность оценки

Сравним эмпирическую относительную ошибку с теоретическими границами. Для каждого B считаем стандартное отклонение отношения N_t/F_0^t по 20 потокам на 100% потока ($N = 500\,000$):

B	m	$\sigma_{\text{эмп}}$	$1.04/\sqrt{m}$	$1.30/\sqrt{m}$
4	16	0.2010	0.2600	0.3250
8	256	0.0616	0.0650	0.0813
10	1024	0.0368	0.0325	0.0406
14	16384	0.0082	0.0081	0.0102

Эмпирика хорошо совпадает с теорией. При $B = 14$: $\sigma_{\text{эмп}} = 0.82\%$, а теоретическое $1.04/\sqrt{2^{14}} = 0.81\%$ — почти идеально. При $B = 10$ эмпирика чуть выше (3.68% против 3.25%), но всё равно в пределах верхней границы 4.06%.

А вот при $B = 4$ эмпирическая ошибка ниже теоретической (20.1% против 26.0%) — скорее всего из-за коррекции Linear Counting, которая при таком маленьком m срабатывает часто и подтягивает оценку.

Наглядно это видно на рис. 4.

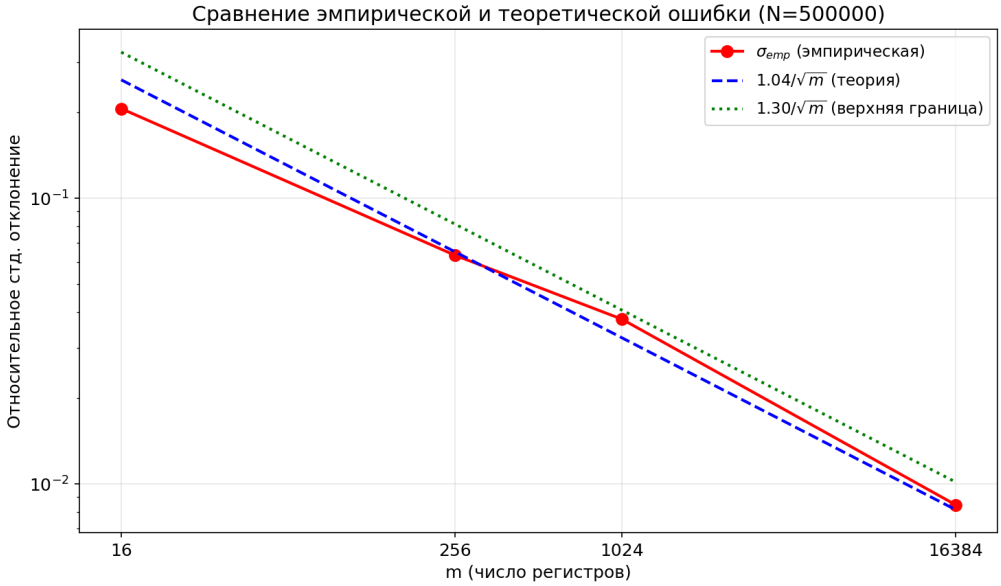

Рис. 4: Эмпирическая и теоретическая относительная ошибка в зависимости от m.

4.2 Стабильность оценки

Посмотрим на средние значения отношения N_t/F_0^t:

B	$\mathbb{E}(N_t/F_0^t)$	Смещение
4	0.9443	-5.6%
8	1.0139	$+1.4\%$
10	1.0081	$+0.8\%$
14	0.9991	-0.1%

При достаточном m алгоритм несмещённый: при $B \geq 8$ среднее отношение близко к 1. А вот при $B = 4$ среднее занижено на $\sim$5.6% — тут всего 16 регистров, коррекция Linear Counting включается слишком часто, отсюда систематическая ошибка.

4.3 Влияние параметра B

Зависимость простая: B увеличивается на 1 — регистров вдвое больше — σ в $\sqrt{2}$ раз меньше. Память растёт линейно с m. По сути, компромисс между точностью и памятью:

- $B = 8$ (256 байт) — грубая оценка с ошибкой $\sim 6\%$, годится, чтобы отличить «тысячи» от «миллионов».
- $B = 10$ (1 КБ) — ошибка $\sim 3\%$, для большинства задач хватает.

- $B = 14$ (16 КБ) — ошибка $< 1\%$. Именно это значение используют в Redis и Google BigQuery, и не случайно: 16 килобайт — копейки по нынешним меркам, а точность отличная.

5 Улучшение: HyperLogLog++

5.1 Суть модификации

У стандартного HyperLogLog две проблемы: (1) 32-битный хеш ограничивает максимальную кардинальность ($\sim 10^9$) и порождает коллизии, (2) при малых кардинальностях работает грубая коррекция Linear Counting.

HyperLogLog++ (Heule et al., 2013) предлагает:

1. **64-битный хеш.** Убирает проблему коллизий для больших множеств. Коррекция для больших значений больше не нужна.

2. **Sparse representation.** Пока элементов мало, храним точные хеш-значения в множестве (`unordered_set`). Когда множество вырастает до $6m$, переключаемся на стандартные регистры.

64-битный хеш строим конкатенацией двух независимых 32-битных MurmurHash3 (с разными seed'ами). Sparse-режим реализован упрощённо: хеши хранятся в `std::unordered_set`. В оригинальной статье Heule et al. используется более компактное представление (sorted list of encoded pairs), но для демонстрации идеи этого достаточно.

```
class HyperLogLogPlus {
public:
    HyperLogLogPlus(int B, const HashFuncGen& h1,
                    const HashFuncGen& h2)
        : B_(B), m_(1u << B), hasher1_(h1), hasher2_(h2),
          registers_(1u << B, 0), use_sparse_(true)
    {
        sparse_threshold_ = m_ * 6;
    }

    void add(const std::string& element) {
        uint64_t h = ((uint64_t)hasher1_.hash(element) << 32)
                   | hasher2_.hash(element);
        if (use_sparse_) {
            sparse_set_.insert(h);
            if (sparse_set_.size() > sparse_threshold_)
                switchToDense();
        } else {
            addToDense(h);
        }
    }

    double estimate() const { ... }

private:
    std::unordered_set<uint64_t> sparse_set_;
```

```
    bool use_sparse_;
};
```

5.2 Результаты сравнения

На рис. 5 сравниваются HLL и HLL++ при $B = 14$, $N = 500\,000$.

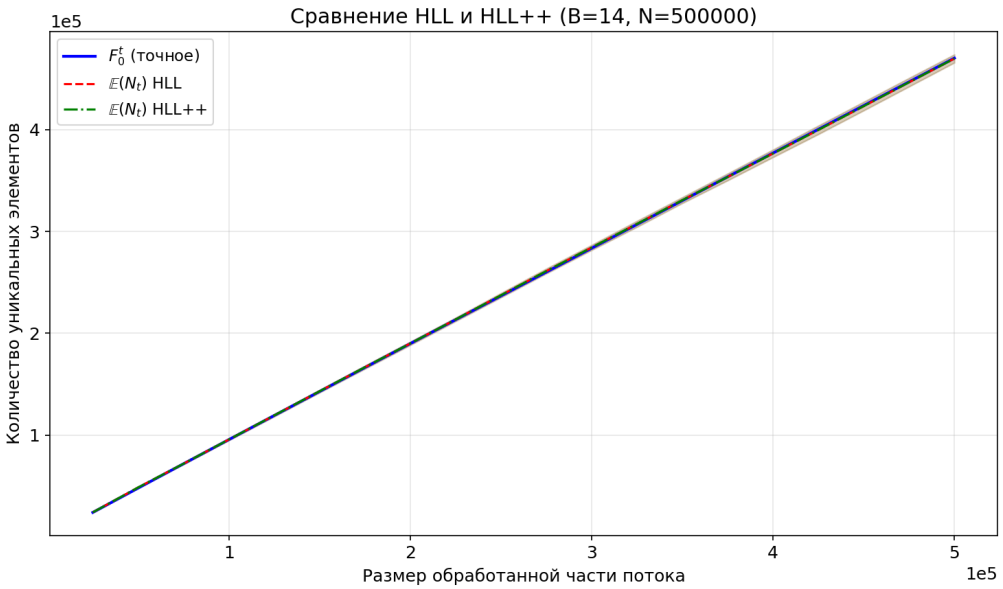

Рис. 5: Сравнение HLL и HLL++ ($B = 14$, $N = 500\,000$, 20 потоков).

Для наших размеров потоков ($\leq 500\,000$) результаты практически одинаковые. Неудивительно: при $N \ll 2^{32}$ коллизии 32-битного хеша редки, а значит 64-битный хеш ничего не даёт. Выигрыш от HLL++ проявляется в других ситуациях:

- При $N > 10^9$, когда 32-битных коллизий становится много и стандартный HLL начинает систематически ошибаться.
- При малых N — sparse-режим даёт точный результат, пока $N < 6m$.

5.3 Память

Измерим реальное потребление памяти:

B	HLL (dense)	HLL++ (пустой, sparse)	HLL++ (после 100K элем.)
10	1024 байт	56 байт	1024 байт (dense)
14	16384 байт	56 байт	∼1.5 МБ (sparse)

При $B = 10$ порог sparse-режима — $6m = 6144$, а 100K элементов это больше, так что HLL++ уже переключился на dense и занимает те же 1024 байта. При $B = 14$ порог — $6 \cdot 16384 = 98304$, и 100K элементов ещё укладываются в sparse. Но `unordered_set` с ∼100K записями по ∼16 байт каждая — это ∼1.5 МБ, заметно больше, чем 16 КБ у стандартного HLL.

Тут видно компромисс нашей упрощённой реализации: `unordered_set` расходует много памяти на overhead (указатели, хеш-таблица). В оригинальном HLL++ sparse-режим компактнее за счёт sorted list и variable-length encoding. Но при переключении на dense потребление памяти сравнивается.

Ещё один вариант оптимизации — упаковать регистры. Каждый хранит $\rho \leq 32 - B + 1$; при $B = 14$ это ≤ 19, т.е. хватит 5 бит. Если паковать по 5 бит, потребление упадёт с 16384 до $\sim$10240 байт — экономия 37%.

Заключение

HyperLogLog — красивый алгоритм: всего 16 КБ памяти, а ошибка оценки кардинальности — менее 1%. Наши эксперименты хорошо подтверждают теорию: $\sigma \approx 1.04/\sqrt{m}$.

Для практики $B = 14$ ($m = 16384$) — оптимальный выбор. HLL++ стоит использовать при кардинальностях $> 10^9$, а для обычных потоков стандартный алгоритм справляется отлично.

Исходный код, данные экспериментов и скрипты визуализации: `https://github.com/kitbuilder587/ADS_SET5_A3`.