Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного программирования

Ершов Александр Владимирович

Высокоскоростной алгоритм хэширования для распределённой системы хранения

Курсовая работа

Научный руководитель:
Разработчик исследовательской лаборатории RAIDIX Маров А. В.

Санкт-Петербург
2015

Оглавление

Введение **3**

0.1. Дедупликация 3

1. Архитектура СХД **5**

1.1. Запись 5

1.2. Чтение 6

2. Постановка задачи **7**

3. Обзор существующих хэш функций **8**

4. Инструменты **9**

4.1. Язык 9

4.2. Векторные вычисления 9

5. Реализация **10**

5.1. SHA-256 10

5.2. Алгоритм 10

5.3. Разница 11

6. Тестирование результатов **12**

Заключение **14**

Список литературы **15**

Введение

Направление высокопроизводительного анализа данных (High Perfomance Data Analysis) в современное время очень актуально. Потребность в высокопроизводительных вычислениях (High Perfomance Computing) растет в разных сферах - моделирование различных физических процессов, аналитика больших данных, криптография, медицина и так далее. Есть несколько основных причин, которые двигают эту область. Возможность увеличения мощности HPC систем. Распространение все более больших и сложных научных систем и инструментов от умных сетей электроснабжения до Большого адронного коллайдера. Новые методы аналитики больших данных : MapReduce/Hadoop, graph analytics. Необходимость выполнять некоторые вычисления со скоростью, близкой к реальному времени.

При этом в среднем примерно 30% данных являются дубликатами. Борьба с дубликатами позволяет решить две проблемы - уменьшить объем занимаемого дискового пространства и количество перезаписей.

0.1. Дедупликация

Дедупликация - это процесс обработки данных, позволяющий избавится от хранения дубликатов. В общем смысле дедупликацию можно разбить на File-level deduplication и Block-level deduplication. В file-level единицей данных является файл, в Block-level - блок данных произвольнной длины. Далее рассматривается Block-level deduplication. В общем смысле алгоритм можно разбить на четыре части:

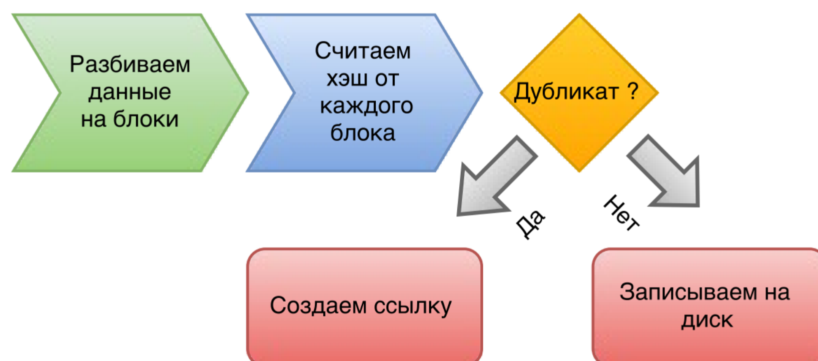


Рис. 1: Дедупликация

- Разбить входные данные на блоки
- Посчитать хэш для каждого блока
- По хэшу определить, был ли такой блок уже записан
- Если данные уже были записаны, то новые данные не записывать, а просто создавать ссылку на старые

По тому, кто выполняет дедупликацию, сам процесс дедупликации делится на source / target - обработка на клиенте или на СХД. По времени выполнения - inline / post-process. В случае inline дедупликация идет в реальном времени, в случае post-process данные сначала записываются , а потом происходит дедупликация.

1. Архитектура СХД

В данной системе используется inline target deduplication. Для чтения и записи используются таблицы, сопоставляющие LBA с хэшем и хэш с физическим местоположением на SSD.

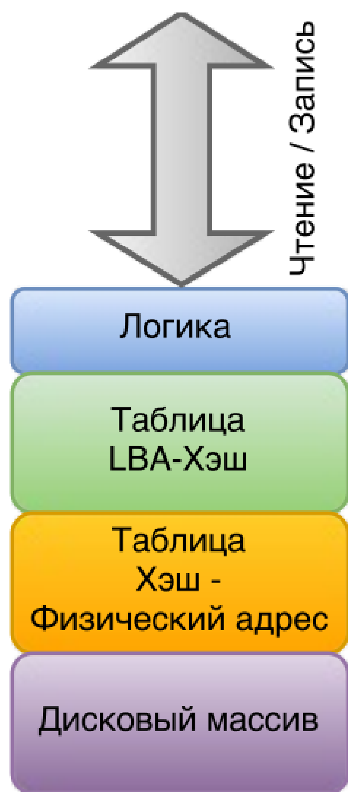


Рис. 2: Архитектура СХД

1.1. Запись

- Данные разбиваются на блоки по 4кб
- Для каждого блока считается хэш
- Для каждого блока по хэшу проверяется, был ли он уже записан
- Если блок уже был записан, система записывает новое сопоставление LBA с хэшем и увеличивает на 1 количество ссылок на данный физический блок. Если блок записан не был, то система выбирает место записи на физическом носителе, создает сопоставление между хэшем и физическим адресом и между LBA и хэшем

1.2. Чтение

По LBA делается сопоставление хэшу, а потом хэш сопоставляются физическому адресу ,а дальше данные извлекаются из их физического местоположения.

2. Постановка задачи

Моей задачей является реализовать хэш функцию с использованием векторных вычислений. Так как СХД[12] оперирует блоками определённого размера (4kb), следует разработать хэш функцию, оптимизированную под эти размеры. Цели оптимизации две : высокая скорость расчёта (10 млн расчётов хэшей 4 ядрами) и минимизация вероятности коллизии, так как при коллизии данные теряются. Используется 256-битный ключ.

3. Обзор существующих хэш функций

В обзоре представлены популярные криптографические хэш функции. Все они обрабатывают данные раундами. Размер блока - количество данных, обрабатываемых за цикл. Размер слова - размер используемых переменных.

Функции	Размер блока	Размер слова	Кол-во раундов	Вывод	Год	Найдены коллизии
MD4	512	32	48	128	1990	+
MD5	512	32	64	128	1992	+
SHA-1	512	32	80	160	1995	+
SHA-256	512	32	64	256	2002	-
SHA-384	1024	64	80	384	2002	-
SHA-512	1024	64	80	512	2002	-

Рис. 3: Обзор хэш функций

Из обзора было решено в качестве основы для алгоритма взять SHA-256, так как в ней не были найдены коллизии, и она - 256-битная.

4. Инструменты

4.1. Язык

Написание кода производилось на языке C[6]. Он был выбран вместо языка Ассемблера по нескольким причинам:

- Писать и отлаживать код на нём проще
- Его можно компилировать под разные архитектуры
- Компиляторы языка C, такие как gcc, выполняют оптимизации кода (например ключи $-O2 - O3$), что может значительно повысить скорость работы

4.2. Векторные вычисления

При реализации использовались технологии SSE[9], AVX[10] и AVX2. Это векторные расширения системы команд x86 для процессоров. Ниже дано описание для SSE, AVX и Intel Intrinsics instructions[2].

SSE (Streaming SIMD Extension) – 128-битная векторная арифметика. Она включает 8 (или 16 для 64-битных систем) XMM0 - XMM7 (XMM15) 128-битных регистров и набор инструкций, работающих со скалярными и упакованными типами данных. Эта технология позволяет распараллелить вычисление, поделив один 128-битный регистр на 8, 16, 32 или 64-битные части, и обрабатывать весь вектор за одну инструкцию.

AVX - это расширение, появившееся после SSE и дополняющее векторы XMM0-XMM15 до 256-битных YMM0-YMM15. Целочисленные вычисления появились только в AVX2. В отличие от SSE, оно не требует выравнивания памяти и использует трёхоперандный синтаксис.

Intel Intrinsics instructions - это набор функций, которые позволяют использовать ассемблерные инструкции из SSE, AVX, MMX без необходимости написания кода на языке Ассемблера. Каждая функция представляет собой одну инструкцию. Полное описание всех функций доступно на Intel Intrinsics Guide.

5. Реализация

5.1. SHA-256

Ниже представлен схема алгоритма SHA-256.

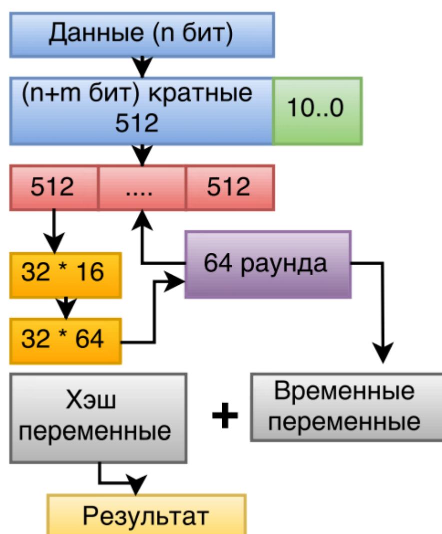


Рис. 4: SHA-256

Рассмотрим, как работает SHA-256 : сначала она дополняет сообщение, чтобы оно было кратно 512. Потом 512-битный блок представляется в виде массива из 16 32-битных переменных, из него делается дополнительно 48 переменных, и таким образом их становится 64. Это делается для увеличения лавинного эффекта. Потом с этими данными проводятся 64 раунда, и результат прибавляется к прошлому. Дальше обрабатывается следующий блок, и так до конца.

SHA-256 была взята за основу моего алгоритма. Но её реализация строго по RFC[1] не подходит по скорости. Поэтому основной цикл был взят из неё, а количество блоков и их размер был изменён. Входные данные 4кб разбиваются на 128-битные блоки, так как регистры XMM являются 128 битными. В оригинальном алгоритме каждый 512-битный блок разбивается на 16 32-битных, и из них создается дополнительно еще 48 32-битных блоков, таким образом количество входных данных увеличивается в 4 раза. В теории это должно влиять на лавинный эффект. Я провел тестирование на 10 миллионах элементов, постоянно уменьшая количество избыточных блоков, и коллизии не были найдены даже при отсутствие этих блоков, поэтому от избыточных блоков было решено отказаться. В итоге получившийся алгоритм представлен ниже.

5.2. Алгоритм

- Разбиваем входные данные 4 кб на блоки по 128 бит, так как регистры XMM

являются 128-битными. Этих блоков получается 256

- Определяем восемь 128-битных переменных a, b, c, d, e, f, g, h с начальным значением и массив K размера 40, также состоящий из 128-битных переменных, где находятся константы

- Определяем 6 логических функций в соответствие с RFC SHA-256

  - $CH(x, y, z) = (x \wedge y) \oplus ((\neg x) \wedge z)$
  - $MAJ(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
  - $BSIG_0(x) = \sigma(x, 2) \oplus \sigma(x, 13) \oplus \sigma(x, 22)$
  - $BSIG_1(x) = \sigma(x, 6) \oplus \sigma(x, 11) \oplus \sigma(x, 25)$
  - $SSIG_0(x) = \sigma(x, 7) \oplus \sigma(x, 18) \oplus (x \gg 3)$
  - $SSIG_1(x) = \sigma(x, 17) \oplus \sigma(x, 19) \oplus (x \gg 10)$

  Где $\sigma(x, n)$ - это циклический сдвиг x вправо на n.

- Основной цикл:

```
for t = 1 to 256 do
    T1 = h + BSIG1(e) + CH(e, f, g) + Kt + Wt
    T2 = BSIG0(a) + MAJ(a, b, c)
    h = g
    g = f
    f = e
    e = d + T1
    d = c
    c = b
    b = a
    a = T1 + T2
    end
```

  где $K_t = K[t \bmod 40]$, $W_t = W[t]$, если $t < 256$, или $W_t = SSIG_1(W(t-64)) + W(t-128) + SSIG0(t-192) + W(t-256)$ для оставшихся t, где $W[i] - i-$я часть входных данных.

- Результат $a \oplus c \oplus e \oplus g$ APPEND $b \oplus d \oplus f \oplus h$, где A APPEND B = AB

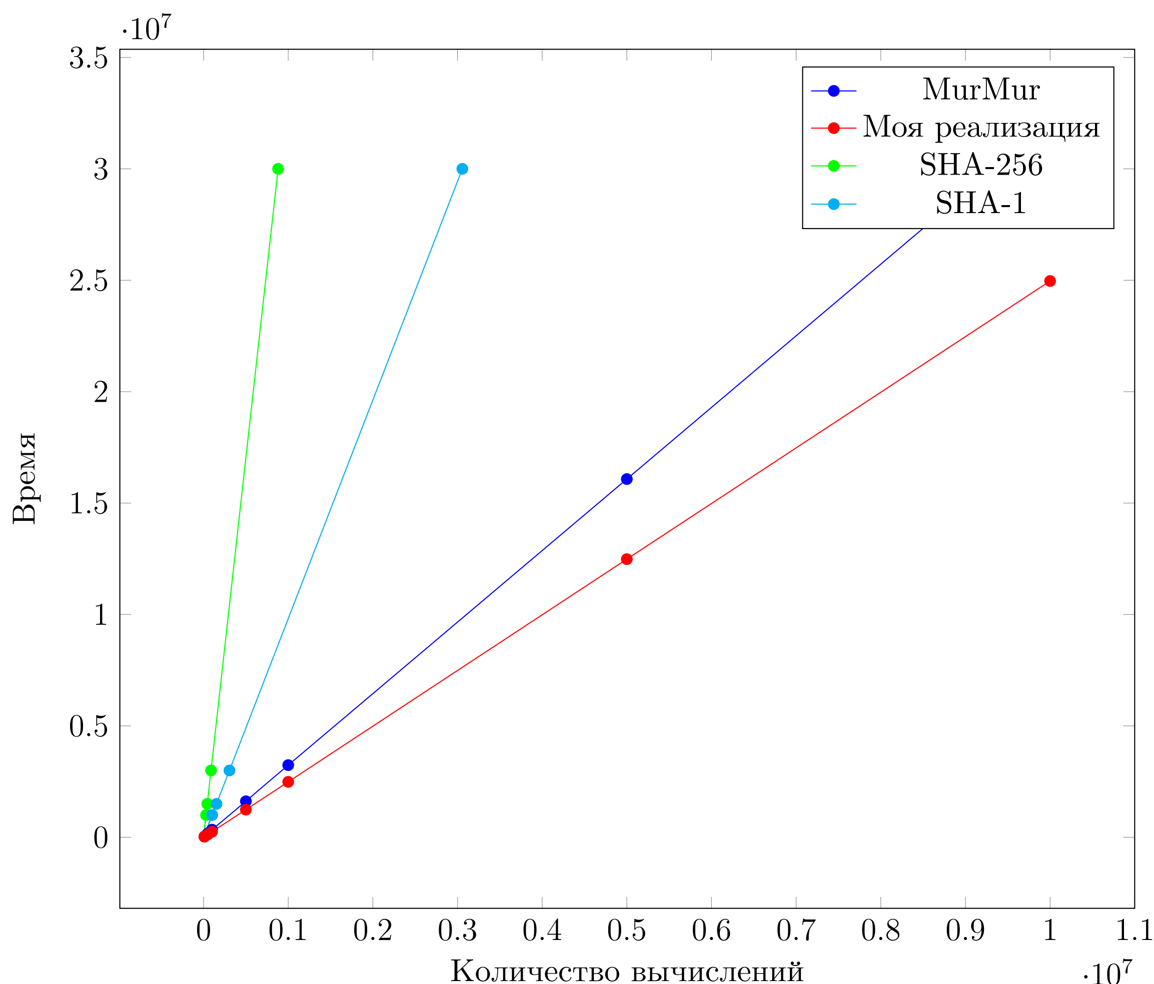
5.3. Разница

Hash	Размер блока	Размер слова (бит)	Количество раундов	Количество дополни - тельных блоков	Вывод (бит)
Sha-256	512 бит	32	64	48	256
Моя реализация	4 кб	128	256	0	256

Рис. 5: SHA-256

6. Тестирование результатов

Тестирование скорости проводилось на ноутбуке с процессором AMD A10 2,3 Ггц и с 6 Гб оперативной памяти. Операционная система Ubuntu. Для сравнения были выбраны три функции : MurMur[7], SHA-1 и SHA-256. По оси x отложено количество вычислений функций от блоков по 4кб. По оси y - время в микросекундах. Отклонение в измерениях в среднем состовляет 15 %.



Сравнение показывает, что благодаря отказу от избыточных блоков и использованию векторных вычислений, скорость моего алгоритма выросла более чем в 10 раз по сравнению с SHA-256.

Заключение

В результате этой работы было сделано следующее:

- Сделан обзор существующих хэш функций
- Реализован свой алгоритм
- Проведено тестирование на скорость в сравнении с существующими хэш функциями

Список литературы

[1] IETF. RFC. — URL: https://www.ietf.org/rfc.html.

[2] Intel. Intel intrinsic guide. — URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[3] Open SSl. — URL: "https://www.openssl.org".

[4] RFC. SHA. — URL: https://tools.ietf.org/html/rfc4634.

[5] Redirect on write. — URL: https://storagegaga.wordpress.com/tag/redirect-on-write.

[6] Wikipedia. C. — URL: http://en.wikipedia.org/wiki/C_(programming_language).

[7] Wikipedia. MurMur. — URL: http://en.wikipedia.org/wiki/MurmurHash.

[8] Wikipedia. SHA криптоанализ. — URL: https://ru.wikipedia.org/wiki/SHA-2.

[9] Wikipedia. SSE. — URL: https://ru.wikipedia.org/wiki/SSE.

[10] Wikipedia. SSE. — URL: https://ru.wikipedia.org/wiki/AVX.

[11] Wikipedia. UUID. — URL: https://ru.wikipedia.org/wiki/UUID.

[12] Wikipedia. СХД. — URL: https://en.wikipedia.org/wiki/Storage_area_network.

[13] Оценка на коллизии известных не криптографических алгоритмов. — URL: http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed.