Основы построения файловых систем

Напоминание: B-деревья, LSM-деревья, B^{ϵ}-деревья

- Бинарные деревья поиска.
  Каждый узел содержит один ключ и имеет только двух потомков
  - - имеют большую глубину,
  - - генерируют много случайного ввода-вывода.

Напоминание: B-деревья, LSM-деревья, B^{ϵ}-деревья

- Бинарные деревья поиска.
  Каждый узел содержит один ключ и имеет только двух потомков
  - - имеют большую глубину,
  - - генерируют много случайного ввода-вывода.

- B-деревья.
  Каждый узел – блок на диске. В нём умещается несколько пар ключ-значение.
  - + меньше глубина по сравнению с бинарным деревом,
  - + чтение одного узла линейное и даёт сразу много пар ключ-значение,
  - - узлы дерева заполнятся не полностью,
  - - удаление создаёт мусор и уменьшает долю полезного занятого места.

Напоминание: B-деревья, LSM-деревья, B^{ϵ}-деревья

- B-деревья.
  Каждый узел – блок на диске. В нём умещается несколько пар ключ-значение.
  - + меньше глубина по сравнению с бинарным деревом,
  - + чтение одного узла линейное и даёт сразу много пар ключ-значение,
  - - узлы дерева заполнятся не полностью,
  - - удаление создаёт мусор и уменьшает долю полезного занятого места.

- LSM-деревья.
  Иерархия полностью заполненных B-деревьев.
  - + нет незанятого места в узлах,
  - + как следствие, у узлов больше потомков (больше fan-out factor),
  - + большее число потомков у узлов влечёт меньшую глубину дерева,
  - - поиск требует поиска в нескольких деревьях или применения фильтра Блума,
  - - сильно неравномерное распределение времен вставок элементов,
  - - write amplification: вставка элемента может привести к переписыванию многих деревьев.

Напоминание: B-деревья, LSM-деревья, B^ϵ-деревья

- LSM-деревья.
  Иерархия полностью заполненных B-деревьев.
  - + нет незанятого места в узлах,
  - + как следствие, у узлов больше потомков (больше fan-out factor),
  - + большее число потомков у узлов влечёт меньшую глубину дерева,
  - - поиск требует поиска в нескольких деревьях или применения фильтра Блума,
  - - сильно неравномерное распределение времен вставок элементов,
  - - write amplification: вставка элемента может привести к переписыванию многих деревьев.

- B^ϵ-деревья.
  Одно B-дерево, где в каждом узле, помимо пар ключ-значение, есть журнал изменений в поддереве.
  - + вставки делаются только в журнал, который оказывается в памяти,
  - + запись в журнал на диске линейная и делается только в один блок,
  - + при переполнении журнала части его выталкиваются в журналы наиболее изменённых потомков, не приводя к переписыванию всего дерева,
  - + поддерживает range queries – запросы на получение значений, соответствующих ключам из диапазона.

Напоминание: Bloom filters

Поиск в LSM-дереве приходится реализовывать как несколько поисков по его составляющим разных уровней.

Можно избежать поиска во многих деревьях T_i, если научиться быстро определять, что искомого ключа в T_i не содержится. Это делает фильтр Блума, вероятностная структура данных, которая по множеству и ключу может выдавать ответы

- элемента в множестве нет,
- элемент в множестве может присутствовать.

Конструкция фильтра Блума: пусть имеется битовый массив длиной **m** и **k** независимых хеш-функций $\mathbf{f_i}$, принимающих значения в диапазоне [0, m-1).

- При вставке элемента **x** установим в 1 биты, стоящие на местах $f_1(x)$, $f_2(x)$, … $f_k(x)$,
- Для проверки отсутствия элемента **y** проверим, установлены ли биты на позициях $f_1(y)$, $f_2(y)$, …, $f_k(y)$.

Фильтр Блума полезен в ситуациях, когда надо быстро определить, что элемент в множество не входит, чтобы избежать дорогостоящего поиска по множеству.

Power of two choices

Зачем в фильтре Блума использовать несколько независимых хешей?

Power of two choices

Зачем в фильтре Блума использовать несколько независимых хешей?

Факт 1: Пусть дана хеш-таблица, где элементы размещаются по хешу в N списков. Вставим в неё N случайных элементов. Какова будет длина максимально заполненного списка? С вероятностью >= 1 – O(1/N) она будет равна

$$\frac{\log N}{\log \log N}$$

Power of two choices

Зачем в фильтре Блума использовать несколько независимых хешей?

Факт 1: Пусть дана хеш-таблица, где элементы размещаются по хешу в N списков. Вставим в неё N случайных элементов. Какова будет длина максимально заполненного списка? С вероятностью >= 1 – O(1/N) она будет равна

$$\frac{\log N}{\log \log N}$$

Факт 2: Пусть дана хеш-таблица, где элементы размещаются в N списков, но правило размещения таково: при вставке элемента посчитаем для него d независимых хешей и выберем самый короткий список, соответствующий одному из полученных хешей.

Вставим N случайных элементов в такую хеш-таблицу. Какова будет длина максимально заполненного списка на этот раз? С вероятностью >= 1 - O(1/N) она составит

$$\frac{\log \log N}{\log d} + O(1)$$

Использование двух хеш-функций вместо одной улучшает асимптотику длины наиболее занятого списка. Использование большего числа только уменьшает константу в O().

Power of two choices

Применения:

- хеш-таблицы,
- фильтры Блума,
- ?

Power of two choices

Применения:

- хеш-таблицы,
- фильтры Блума,
- балансировка нагрузки в распределённых системах.

Power of two choices

Применения:

- хеш-таблицы,
- фильтры Блума,
- балансировка нагрузки в распределённых системах.

Пример: если есть несколько HTTP-серверов, с которых можно скачать файл, то можно очень просто распределить нагрузку между ними:

1. выбрать два случайных сервера,
2. отправить обоим один и тот же запрос,
3. с того, кто первым начнёт слать ответ, скачать файл,
4. у более медленного отменить запрос.

https://people.cs.umass.edu/~ramesh/Site/PUBLICATIONS_files/MRS01.pdf

Способы проверки целостности данных

У нас упоминались два инструмента для проверки целостности данных:

- Cyclic redundancy checks,
- Криптографические хеш-суммы.

Обсудим их детальнее.

Cyclic Redundancy Check

Рассмотрим сообщение как последовательность битов (элеметов GF(2)) и сопоставим ему многочлен из $GF(2)[X]$:

$$a_{n-1}a_{n-2}\dots a_0 \leftrightsquigarrow M(X) = X^n + a_{n-1}X^{n-1} + a_{n-2}X^{n-2} + \dots + a_0$$

Возьмём многочлен $C \in GF(2)[X]$ степени m, посчитаем r(X) – остаток от деления $M(X) * X^m$ на $C(X)$.
Теперь построим многочлен

$$M(X) * X^m + r(X)$$

Ему соответствует сообщение $a_{n-1}a_{n-2}\dots a_0$, к которому дописали биты, равные коэффициентам r (внимание: r может быть степени меньше m, тогда считаем коэффициенты пристарших степенях нулями).

CRC очень хорошо приспособлены для аппаратной реализации: из арифметических операций нужен только XOR.

Многочлен C(X) подбирается так, чтобы обеспечить обнаружение определённых типов ошибок.

Ошибки, которые находит CRC (упражнения)

- Если C(X) имеет два и более ненулевых коэффициентов, то он определяет любую ошибку, изменяющую только один бит.
- Если в разложении C(X) на неприводимые множители есть многочлен степени m, то C(X) определяет любую ошибку, изменяющую только два бита, расположенных на расстоянии, меньшем m.
- Если C(X) делится на X+1, то он определяет любую ошибку, меняющую нечётное число бит.

Примеры CRC

Название	Где применяется	Порождающий многочлен*
CRC-16-CCITT	Bluetooth	0x1021
CRC-16-IBM	USB	0x8005
CRC-32	Ethernet, SATA, MPEG-2, gzip, bzip2, PNG	0x04C11DB7
CRC-32C (Castagnoli)	iSCSI, SCTP, SSE4.2, btrfs, ext4, Ceph	0x1EDC6F41

** Отдельные биты числа рассматриваются как коэффициенты порождающего многочлена.*

Криптографические хеши

CRC очень просты в вычислении и обнаруживают простые ошибки. Но их легко обмануть намеренными ошибками.

Для надёжной проверки того, что блок данных не был повреждён или изменён, применяются криптографические хеши:

- MD4, MD5
- SHA1, SHA-256, SHA-384, SHA-512.

На них полагаются, поскольку сейчас не известно алгоритмов поиска коллизий этих хешей, кроме перебора:

- Для MD4, MD5 и SHA1 – большого числа вариантов,
- Для SHA-256 и старше – полного перебора.

Примерная скорость вычисления хешей и CRC*

Алгоритм	MB/sec	Cycles/byte
CRC32	253	6.9
Adler32**	920	1.9
MD5	255	6.8
SHA-1	153	11.4
SHA-512	99	17.7

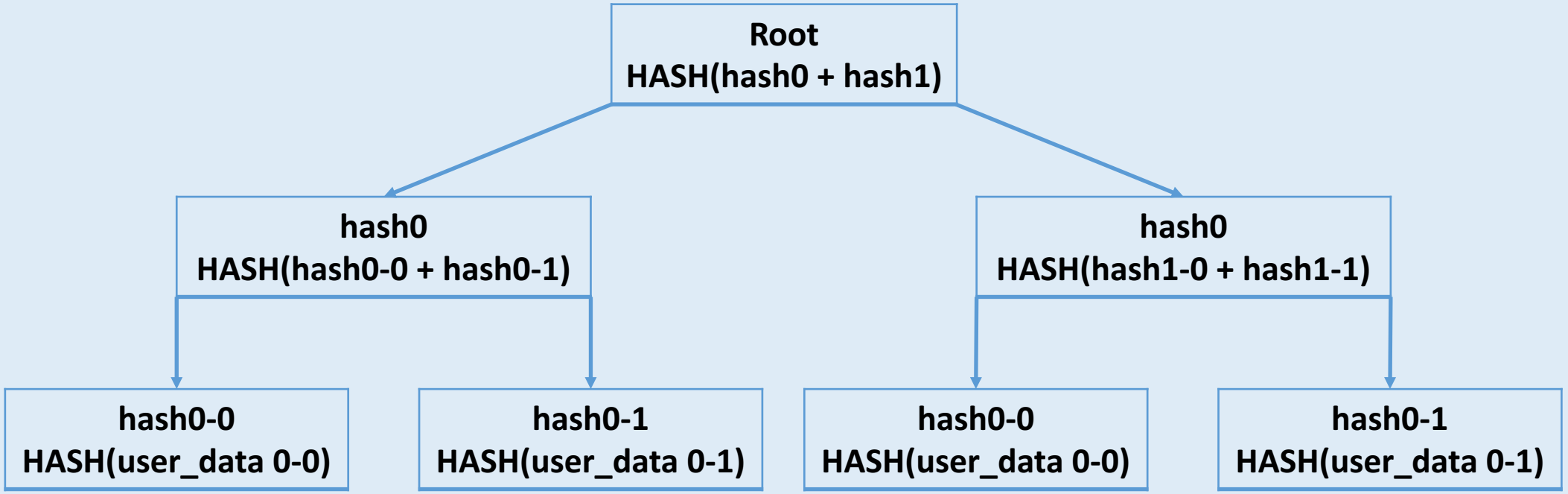
** Данные для Core 2 1.83GHz*

*** Не является CRC; используется в zlib*

Примерная скорость вычисления хешей и CRC*

Алгоритм	MB/sec	Cycles/byte
CRC32	~~253~~ 2500 (Haswell @ 2.2Ghz)	6.9
Adler32	920	1.9
MD5	255	6.8
SHA-1	153	11.4
SHA-512	~~99~~ 300	17.7

Пример проверки целостности дерева: Merkle trees



Применения:

- проверка целостности структуры дерева каталогов и дерева екстентов (ZFS, btrfs),
- проверка подлинности данных в p2p-сетях.

Пример неправильного использования криптографических хешей

Для криптографические хеши трудно найти коллизии (preimage attacks). Но их можно создать ненамеренно.

REST-запросы зачастую используют параметры URL: /path?key1=value1&key2=value2&…

Amazon в первых версиях S3 и SimpleDB использовала следующее правило подписи REST-запросов:

1. Отсортировать пары key=value по значению ключа,
2. Написать отсортированные пары key=value подряд: key1 + value1 + key2 + value2 + …,
3. Подписать полученную строчку с помощью HMAC-SHA-1.

Пример неправильного использования криптографических хешей

Для криптографические хеши трудно найти коллизии (preimage attacks). Но их можно создать ненамеренно.

REST-запросы зачастую используют параметры URL: /path?key1=value1&key2=value2&…

Amazon в первых версиях S3 и SimpleDB использовала следующее правило подписи REST-запросов:

1. Отсортировать пары key=value по значению ключа,
2. Написать отсортированные пары key=value подряд: key1 + value1 + key2 + value2 + …,
3. Подписать полученную строчку с помощью HMAC-SHA-1.

Тут легко создать коллизию у подписи: достаточно передать параметр вида key=“value” + “another_key” + “another_value”.

Сценарий атаки: если есть доступ до сервиса, который позволяет менять часть параметров, передавая запросы вида

/path?Attribute.0.Name=name&Attribute.0.Value=new_value

то:

- Попросить установить параметр benign_parm в значение “benign_value” + “Attribute.1.Name” + “owner” + “Attribute .1.Replace” + “true” + “Attribute.1.Value” + “Dr.Evil”
- Подсмотреть подпись, созданную для этого запроса и использовать её же для запроса с аргументами /path?Attribute.0.Name=benign_parm&Attribute.0.Value=benign_value&Attribute.1.Name=owner&Attribute.1.Replace=true &Attribute.1.Value=Dr.Evil