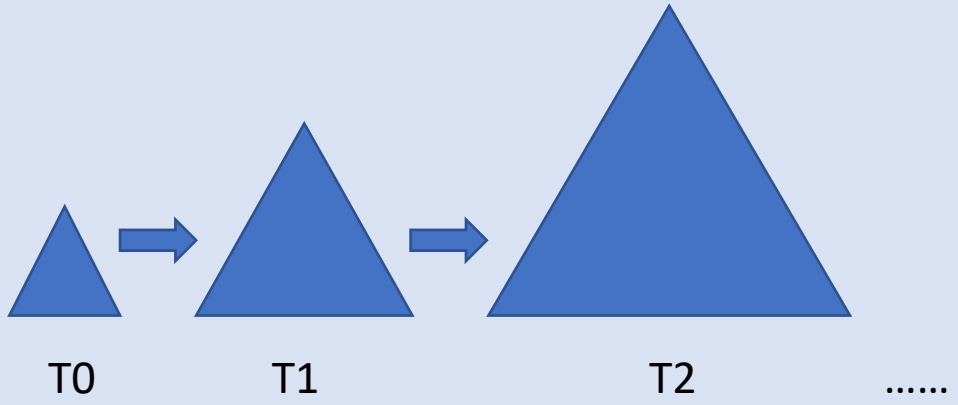
Основы построения файловых систем

Log-Structured Merge Tree

LSM-дерево – это иерархия B-деревьев.

- Поиск элемента делается по очереди в деревьях T_0, T_1, …,
- Вставки делаются только в дерево T_0,
- Дерево T_0 (возможно, несколько первых) располагается в RAM, гарантируя быструю вставку,
- При переполнении дерева T_i оно сливается с деревом T_{i+1}; полученное дерево объявляется новым T_{i+1},
- Удаление элемента реализуется как вставка элемента, помеченного флагом «удалённый»*. Фактическое удаление произойдёт при слиянии деревьев.



** Такой элемент называется “tombstone”.*

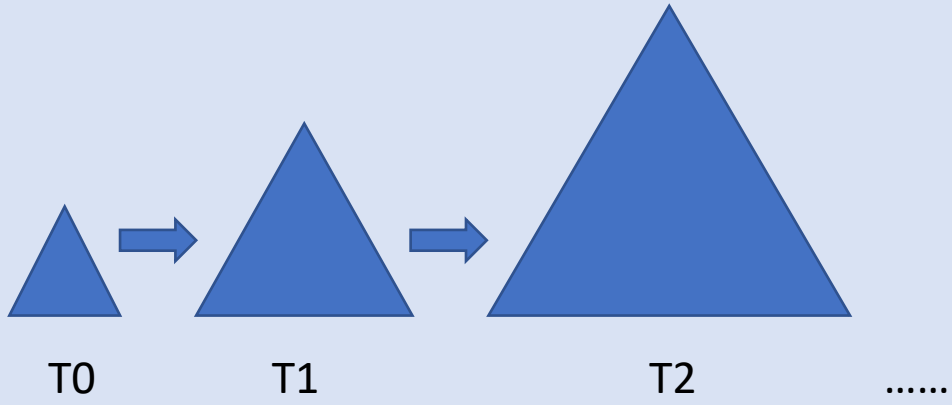
Log-Structured Merge Tree

При таком подходе вставки и удаления почти всегда получаются очень быстрыми, поскольку работают только с деревом в RAM.

Но теперь необходимо иногда выполнять слияние деревьев. Как мы обсудили, это можно сделать быстро и генерировать только эффективные последовательности чтений и записей.

Остаются вопросы:

1. Надёжность сохранения данных в T0.
2. Поиск необходимо выполнять не в одном дереве, а во многих.
3. Write amplification и непредсказуемые задержки: у LSM-дерева мало амортизированное время вставки в дерево, но вставки, которые приводят к слиянию деревьев, будут исполняться на несколько порядков дольше, чем типичные вставки, и создадут много IO.



Log-Structured Merge Tree

При таком подходе вставки и удаления почти всегда получаются очень быстрыми, поскольку работают только с деревом в RAM.

Но теперь необходимо иногда выполнять слияние деревьев. Как мы обсудили, это можно сделать быстро и генерировать только эффективные последовательности чтений и записей.

Остаются вопросы:

1. Надёжность сохранения данных в T0.
2. Поиск необходимо выполнять не в одном дереве, а во многих.
3. Write amplification и непредсказуемые задержки: у LSM-дерева мало амортизированное время вставки в дерево, но вставки, которые приводят к слиянию деревьев, будут исполняться на несколько порядков дольше, чем типичные вставки, и создадут много IO.

T0 хранится в RAM и при исчезновении питания теряется. Зачастую приложениям ОК потерять несколько последних секунд работы

Log-Structured Merge Tree

При таком подходе вставки и удаления почти всегда получаются очень быстрыми, поскольку работают только с деревом в RAM.

Но теперь необходимо иногда выполнять слияние деревьев. Как мы обсудили, это можно сделать быстро и генерировать только эффективные последовательности чтений и записей.

Остаются вопросы:

1. Надёжность сохранения данных в T0.
2. Поиск необходимо выполнять не в одном дереве, а во многих.
3. Write amplification и непредсказуемые задержки: у LSM-дерева мало амортизированное время вставки в дерево, но вставки, которые приводят к слиянию деревьев, будут исполняться на несколько порядков дольше, чем типичные вставки, и создадут много IO.

Bloom filters

Log-Structured Merge Tree

При таком подходе вставки и удаления почти всегда получаются очень быстрыми, поскольку работают только с деревом в RAM.

Но теперь необходимо иногда выполнять слияние деревьев. Как мы обсудили, это можно сделать быстро и генерировать только эффективные последовательности чтений и записей.

Остаются вопросы:

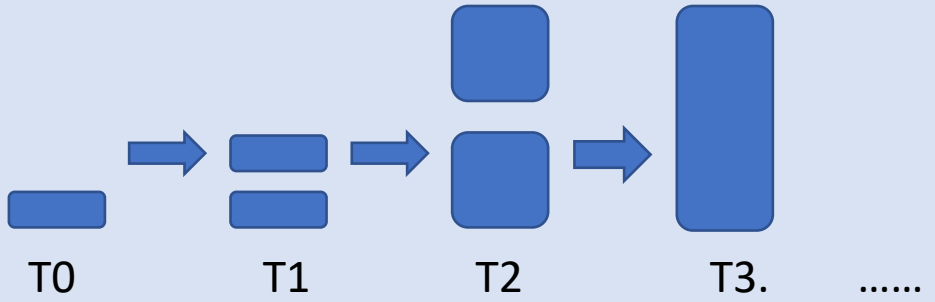
1. Надёжность сохранения данных в T0.
2. Поиск необходимо выполнять не в одном дереве, а во многих.
3. Write amplification и непредсказуемые задержки: у LSM-дерева мало амортизированное время вставки в дерево, но вставки, которые приводят к слиянию деревьев, будут исполняться на несколько порядков дольше, чем типичные вставки, и создадут много IO.

- Write stalls.
- Слияние деревьев в фоновом процессе.
- Компоненты не меняются, поэтому поиск может идти одновременно со слиянием и использовать старые версии деревьев.
- LSM partitioning.
- LSM tiering.

LSM tiering

Каждый этаж T_i дерева представляется как объединение нескольких деревьев $T_{i,j}$. Размер Ti,j поддерживается примерно равным размеру предыдущего этажа.

Слияние деревьев в i-м этаже только добавляет дерево в (i+1)-й этаж.



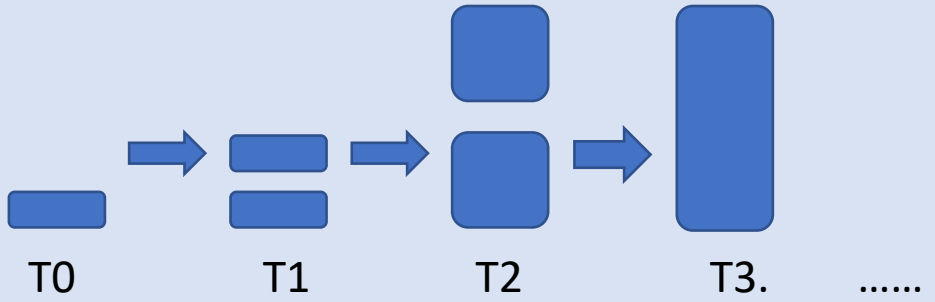
LSM tiering

Каждый этаж T_i дерева представляется как объединение нескольких деревьев $T_{i,j}$. Размер Ti,j поддерживается примерно равным размеру предыдущего этажа.

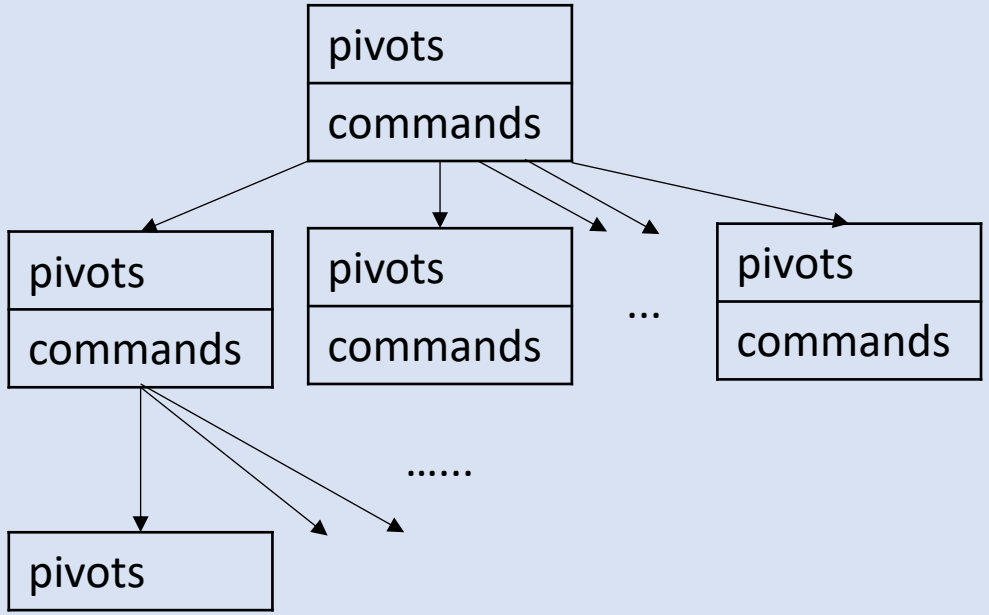
Слияние деревьев в i-м этаже только добавляет дерево в (i+1)-й этаж.

Такая конструкция выгодна тем, что уменьшает количество IO, требуемых для слияния, т.к. весь (i+1)-й этаж вычитывать не надо. LSM tiering применяется в системах, где важно поддерживать большое количество вставок.

Минусы состоят в дублировании ключей внутри этажа, т.е. менее эффективном использовании диска, и необходимости делать поиск по ключу в каждом дереве этажа по отдельности.



B^{ϵ}-деревья



B^{ϵ}-деревья объединяют B-дерево и журнал.

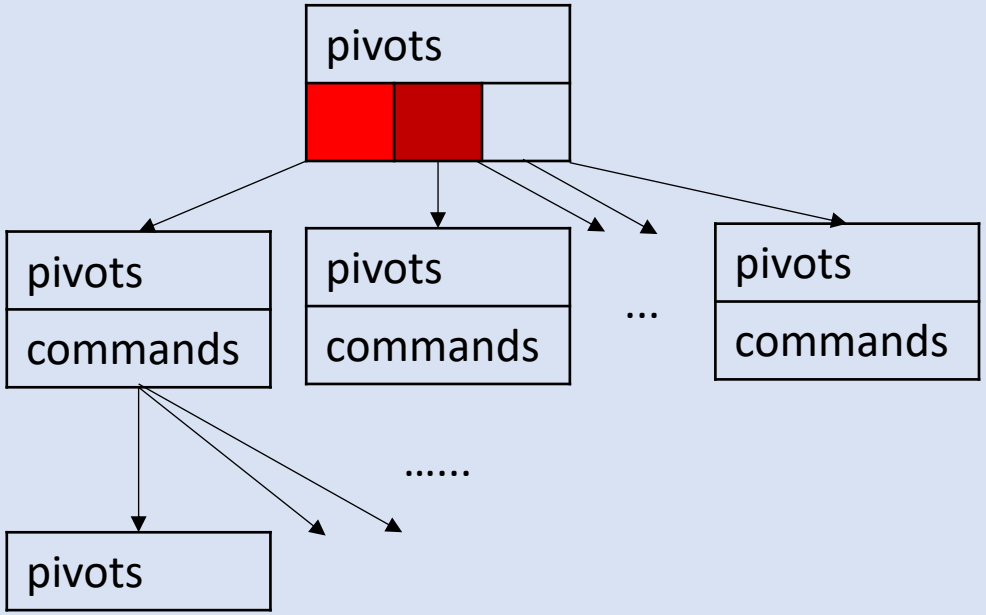
Как и в B-деревьях, узлы являются достаточно длинными блоками, но теперь они разделяются на две части:

- pivots (пары ключ-значение с указателями на пользовательские данные или на другие узлы дерева),
- commands (журнал вставок и удалений).

Узлы длиной B байт делятся в пропорции B^{ϵ} байт на pivots и $B - B^{\epsilon}$ байт на commands.

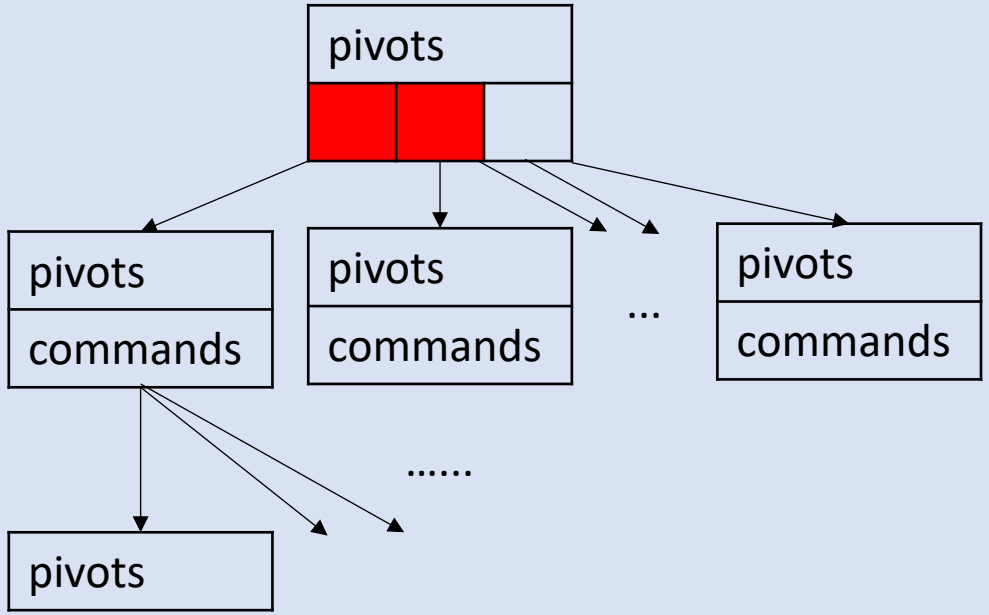
https://www.usenix.org/system/files/login/articles/login_oct15_05_bender.pdf

B^{ϵ}-деревья



- Вставки и удаления добавляются только в журнал корневого узла,
- При переполнении журнала в корне он выталкивается в журналы дочерних узлов, причём выталкиваются только модификации наиболее изменённых поддеревьев.
- В частности, число изменений, которые выталкиваются в поддерево, не меньше $(B - B^{\epsilon}) / B^{\epsilon} \approx B^{1-\epsilon}$. Таким образом, амортизированное время вставки составляет $O\left(\frac{\log_B N}{B^{\epsilon}}\right)$.

B^ϵ-деревья



У журнала есть ещё преимущества: записи в нём можно трактовать не как добавления элементов, а как команды, произвольно модифицирующие поддеревья.

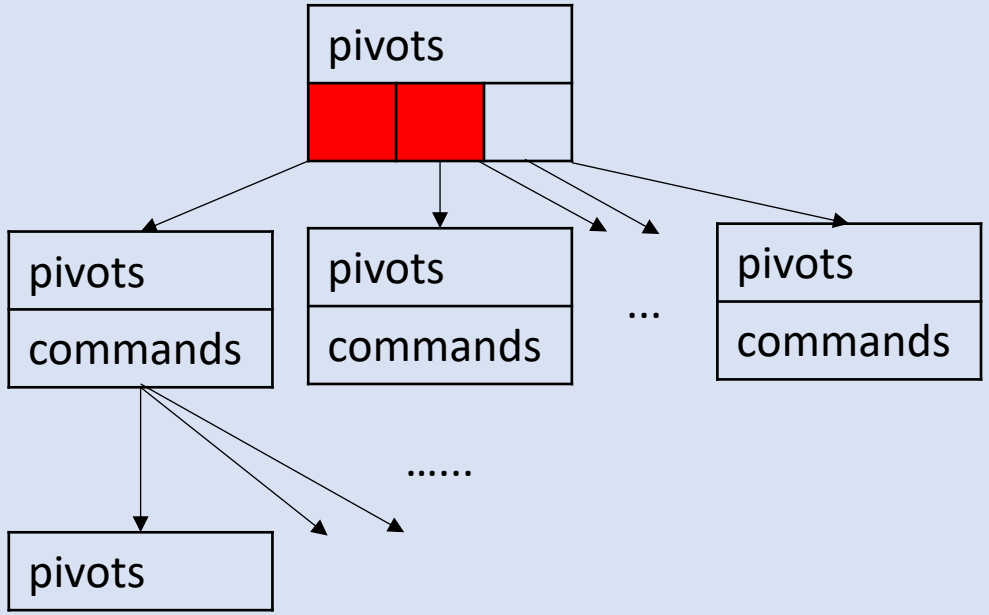
Примеры:

- update,
- upsert (UPdate or inSERT),
- range delete.

Как следствие, для B^ϵ-деревьев возможна эффективная реализация операции “range query” (получить все элементы множества, попадающие в указанный диапазон).

Какая сложность этой операции у B^ϵ-дерева и у LSM-дерева?

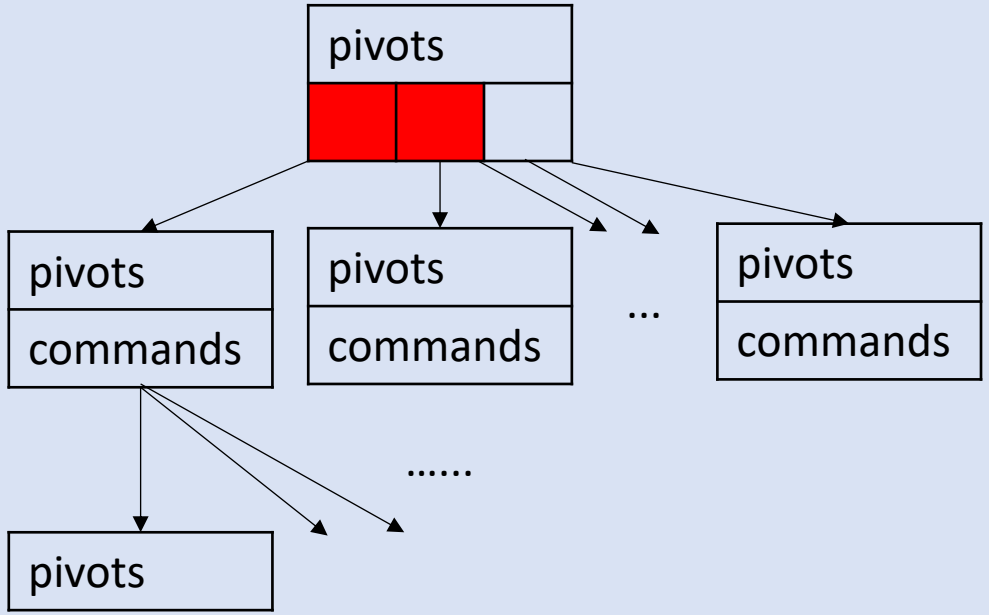
B^{ϵ}-деревья



Преимущества такой реализации:

- Нет необходимости поиска во многих деревьях или построения фильтров Блума,
- Журнал изменений расположен в корневом узле, в котором происходит бОльшая часть изменений, всегда в кеше,
- При расщеплении журнала изменений генерируется меньше IO, чем при слиянии компонент LSM-дерева,
- Размер узлов B^{ϵ}-деревьев можно делать много больше, чем у B-деревьев, что уменьшает их глубину.

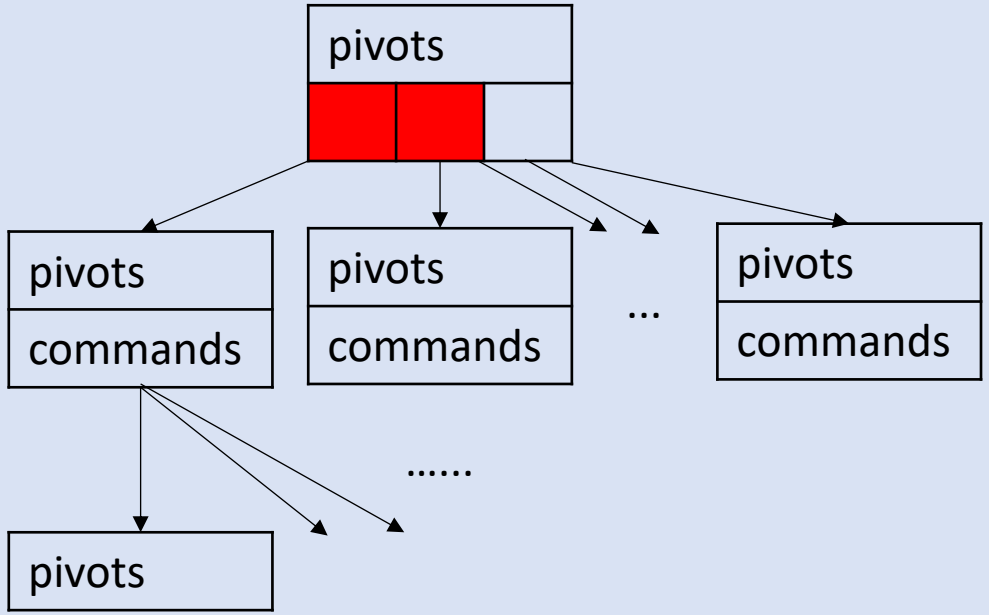
B^{ϵ}-деревья: журналирование и tree checkpointing



Файловые системы вроде ext4 и XFS ведут журнал физических изменений. Это много проще журнала логических изменений ФС, поскольку все операции в нём идемпотентны.

Что пишется в журнал B^{ϵ}-дерева: логические или физические изменения?

B^{ϵ}-деревья: журналирование и tree checkpointing

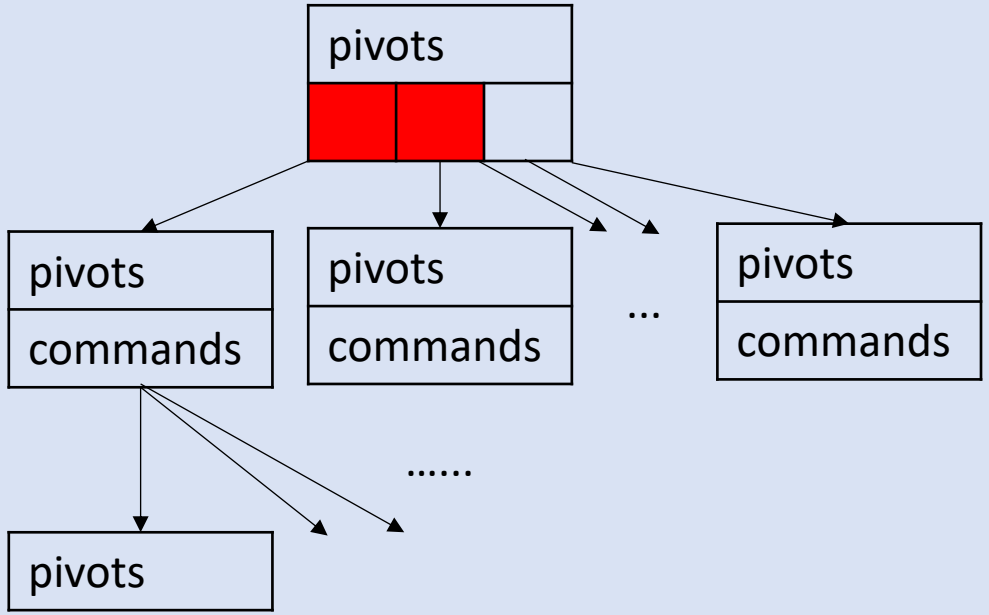


Файловые системы вроде ext4 и XFS ведут журнал физических изменений. Это много проще журнала логических изменений ФС, поскольку все операции в нём идемпотентны.

А в B^{ϵ}-дереве журналируются, наоборот, логические изменения.

В чём разница? Как делать crash recovery для B^{ϵ}-дерева и не получить проблем с неидемпотентностью?

B^{ϵ}-деревья: журналирование и tree checkpointing



Файловые системы вроде ext4 и XFS ведут журнал физических изменений. Это много проще журнала логических изменений ФС, поскольку все операции в нём идемпотентны.

А в B^{ϵ}-дереве журналируются, наоборот, логические изменения.

В чём разница? Как делать crash recovery для B^{ϵ}-дерева и не получить проблем с неидемпотентностью?

- Ссылка на корневой узел хранится в суперблоке дерева.
- Crash recovery не меняет узлы дерева, а только выписывает изменённые.
- Когда изменения полностью выписаны, обновляется указатель на корень в суперблоке.
- Если crash recovery выполняется второй раз, то эта она начинается со старого суперблока, т.е. с того же дерева, что и прерванная, поэтому никакая запись в журнале не применяется дважды.

Read, write, and space amplification

В рассмотренных сегодня конструкциях получалось улучшить один или два параметра за счёт остальных:

- B-дерево имеет лучшее время поиска, чем rb-дерево (если говорить о хранении на диске, а не в памяти), но занимает больше места на диске из-за не полностью заполненных узлов,
- LSM-дерево обеспечивает гораздо более быстрые вставки в типичном случае, чем B-дерево, но асимптотика поиска в нём хуже,
- Чтобы исправить асимптотику поиска в LSM-дереве, мы добавили фильтры Блума и потеряли немного места и ресурсов CPU,
- Сжатие данных в дереве может существенно уменьшать его размер на диске за счёт времени вставки и поиска,
- LSM tiering уменьшает количество IO, но приводит к большему использованию дискового пространства.

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

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

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

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

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,
- уменьшение tail latency в распределённых системах.

Power of two choices

Применения:

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

Проблема: удвоение числа запросов (не считая запросов на отмену).

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

Power of two choices

Применения:

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

Вопрос: пусть у нас есть N одинаковых серверов, которые 99% запросов обрабатывают < 10ms, а 1% запросов (случайных) обрабатывают 1s. Если для построения ответа пользователю требуется получить ответ от 10 серверов, то какая доля пользовательских запросов будет обработана за 10ms?

http://cseweb.ucsd.edu/~gmporter/classes/fa17/cse124/post/schedule/p74-dean.pdf

Power of two choices

Применения:

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

Идея: послать запрос одному серверу и, если время ответа превысило 90-й (95-й) персентиль, то перепослать запрос уже другому серверу.

Заодно это решает проблему удвоением числа запросов при наивном применении power of two choices.

http://cseweb.ucsd.edu/~gmporter/classes/fa17/cse124/post/schedule/p74-dean.pdf