Operating Systems

File Systems

Me

November 24, 2016

Файловая система

- Блочные устройства позволяют хранить большие объемы информации долгое время
  - чем больше информации тем больше хочется ее структурировать.
- Multics впервые (*) ввел в использование иерархическую файловую систему:
  - файлы, каталоги/директории/папки, жесткие и символьные ссылки;
  - так как Multics пофейлился, популярной иерархиеские ФС сделал скорее Unix.

Операции над файлами

- открытие файла - возвращает некоторый дескриптор, который используется для дальнейших операций с файлами
  - снаружи этот дескриптор обычно выглядит просто как целое число;
  - внутри ОС это обычно некоторая структура, которая кеширует информацию о файле (размер, права доступа, дата модификации и тд и тп);
- закрытие файла - освобождение всех ресурсов связанных с открытым файлом;
- чтение и запись по некоторому смещеную в файле;
- запись за пределы файла изменяет его размер.

Операции над каталогами

- создание и удаление файлов/жестких ссылок на файлы;
- создание и удаление символьных ссылок на файлы и каталоги;
- создание и удаление других каталогов;
- перечисление файлов/каталогов внутри каталогов.

Простейшая файловая система

- Основная структура данных - связный список:
  - мы будем связывать в список блоки фиксированного размера;
  - размер блока кратен размеру сектора диска.
- Файлы
  - файл идентифицируется первым блоком;
  - все блоки с содержимым файла просто связаны в список.
- Каталоги
  - каталог - файл, который хранит записи фиксированного формата
    - каждая запись хранит имя дочернего каталога/файла, тип и номер первого блока.
- Свободные блоки
  - все свободные блоки просто связаны в список.

"Эффективный" связный список

- Связный список не очень эффективная структура данных для индексации
    - но мы все же постараемся сделать с этим что-нибудь...
    - ну хоть что-нибудь...
- Что нужно учитывать?
    - общение с диском осуществляется секторами (минимум 512 байт);
    - общение с диском медленное - чем меньше обращений тем лучше.

"Эффективный" связный список

- Заведем таблицу:
  - каждая запись в таблице соответствует блоку ФС (i-ая запись, соответствует i-ому блоку);
  - каждая запись - это просто число, а именно номер следующего блока или маркер конца списка.
- Почему не хранить ссылку на следующий блок прямо в блоке?
  - таблица гораздо компактнее - за одно чтение мы получаем сразу несколько ссылок;
  - если повезет, то это будут нужные нам ссылки;
  - если повезет, то мы можем прочитать всю таблицу целиком и хранить ее в памяти.

File Allocation Table (FAT)

- ФС FAT12/16/32 - пожалуй одна из самых популярных ФС в мире:
  - активно используется во всяких устройствах (MP3 плееры, фотоаппараты, USB флешки и т.д. и т.п.);
  - мало функциональная;
  - не очень надежная;
  - зато очень простая.
- FAT использует связные списки и похожую таблицу блоков
  - эта таблица и называется File Allocation Table (FAT).

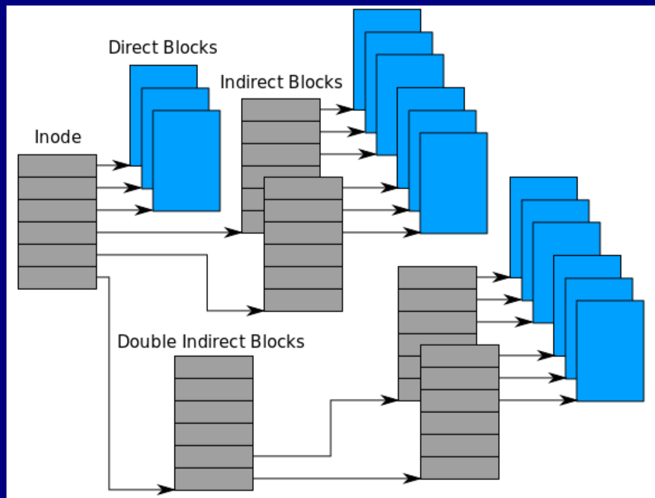
Суперблок

- ФС начинается с суперблока
  - суперблок - это структура, которая хранит общую информацию о ФС
    - магическое число/строка - чтобы убедиться, что на диске хранится именно наша ФС;
    - размер блока ФС;
    - версия ФС;
    - ссылки на какие-то общие структуры ФС, например, список свободных блоков.
  - Суперблок хранится в каком-то известном месте диска
    - иногда на диске хранится несколько копий супеблока, чтобы если один из них испортится, то можно будет использовать другой;
    - потеря суперблока, это практически потеря ФС.

Индексные узлы (Inode-ы)

- Inode - важная структура в классических Unix-овых (и не только) ФС
  - Inode представляет сущность внутри ФС (файл/каталог):
    - хранит права/привелегии доступа, даты модификации и прочее;
    - размер файла/каталога и *расположение его блоков на диске*;
    - примерное представление об содержимом Inode вам может дать *man 2 stat*;
  - каждый Inode имеет уникальный идентификатор, по которому его можно найти на диске
    - зная идентификатор вы можете легко найти Inode на диске;
    - в классических ФС, при форматировании выделяется таблица Inode-ов;
    - идентификатор Inode - это просто его позиция в таблице.

Описание расположения блоков



Fast File System

- Fast File System - первая из классических Unix-овых ФС, в которой были учтены особенности диска
  - не то, чтобы до этого никто не заботился о производительности;
  - но ребята из Berkely, которые создали FFS довольно буедительно показали, что диск в ФС того времени использовался неэффективно.
- В Fast File System минимальный размер блока - 4Kb
  - т. е. за одно обращение к диску ФС пишет/читаете сразу 8 секторов;
  - это привело к заметному увеличению скорости работы ФС, что ясно показало, что другими ФС диск использовался не эффективно.

Цилиндровые группы

- Цилиндровая группа - группа блоков, которые расположены на диске рядом
  - цилиндровые группы - еще одна оптимизация введенная в FFS;
  - каждая цилиндровая группа содержит свой набор Inode-ов, битовую карту свободных/занятых блоков внутри группы и прочее.
- Цилиндровые группы используются для более эффективной аллокации ресурсов на диске:
  - достаточно большие куски файла пытаются положить в одну цилиндровую группу;
  - Inode-ы файлов в одном каталоге, стараются также положить в одну цилиндровую группу;
  - при этом стараемся не заполнять цилинддровые группы под завязку.

Классические Unix-овые ФС

- FFS - это предшественник классических Unix-овых ФС
  - сейчас никто не использует FFS (скорее всего), но многие пользуются ее наследниками: ext3 и ext4 (ext2, если кто-то ей пользуется).
- Многие детали отличаются, но многие идеи остаются:
  - например, каталог это не просто набор записей, а HTree/B+-tree или какая-то подобная индексная структура данных;
  - журналирование для обеспечения консистентности вместо soft updates;
  - а цилиндровые группы остались, хотя возможно называются по-другому.

Индексные структуры данных

- Для быстрого поиска файла/каталога внутри каталога или смещения на диске по смещению внутри файла можно использовать подходящую структуру данных:
  - чтобы быстро находить и открывать файлы по имени;
  - чтобы читать/писать файлы не по-порядку.
- Имя подхоядщий словарь мы вообще можем хранить все дерево каталогов в одном словаре:
  - отображаем номер Inode родительского каталога и имя на номер Inode файла/каталога внутри каталога;
  - с таким словарем нам даже не нужно заранее выделять место под таблицу Inode-ов - мы можем просто хранить их в таком словаре.

Деревья поиска

- Бинарные деревья поиска - классический вариант индексной структуры данных в памяти:
  - Red-Black деревья;
  - AVL деревья;
  - Splay деревья;
  - декартовы деревья.
- Бинарные деревья поиска не годятся в качестве индекса на диске
  - они неэффективны для блочного интерфейса (меньше 512 байт за раз читать/писать мы все равно не можем);
  - бинарные деревья поиска очень высокие (даже идеально сбалансированные).

Ветвистые деревья

- Если бинарные деревья поиска слишком высокие, то просто будем использовать N-арные, где N будет 10, 100, 1000...
  - если каждый узел может хранить больше 2 ссылок на дочерние узлы, то высота дерева может быть меньше;
  - за одно обращение к диску мы можем читать сразу N ссылок.
- B-деревья - идеально сбалансированные N-арные деревья поиска:
  - со временем B-деревья получили много вариаций: $B+$, B^*, B^ϵ;
  - вариации B-деревьев используются повсеместно для индексации (в ФС, базах данных, Key-Value Store-ах и прочих).

B+ деревья

- B+ дерево - идеально сбалансированное сильно ветвистое дерево поиска, которое хранит значения только в листьях:
  - растояние от корня до всех листьев одинаковое;
  - каждый узел кроме корня хранит не менее B ключей, где B некоторый фиксированный параметр дерева;
  - словарь хранит пары - (ключ, значение) - все внутренние узлы хранят только ключи, и только листья хранят полные пары.

Инварианты B+ деревьев

- Каждый узел кроме корня хранит от B до $2B$ ключей (ссылок на дочерние узлы):
  - соответсвенно высота дерева растет как log_B;
  - на практике верхнюю границу делают больше, чтобы обеспечить гистерезис.
- Корень, если он не является листом соедржит как минимум 2 ключа:
  - это условие определяет когда можно удалить текущий корень и уменьшить высоту дерева.

Балансировка B+ деревьев

- Бинарные деревья поиска, зачастую, используют повтороты:
  - B деревья не используют повортов - их способ гораздо "топорнее".
- Если в узле мало ключей:
  - можно забрать часть ключей у соседа, если у него есть лишние;
  - можно просто объединиться с соседом и продолжить балансировку рекурсивно от родителя;
  - если соседей нет - значит мы корень.
- Если в узле много ключей:
  - просто разбиваем узел на два и продолжаем балансировку рекурсивно от родителя.

Пример вставки в B+ дерево

2	4	6	8

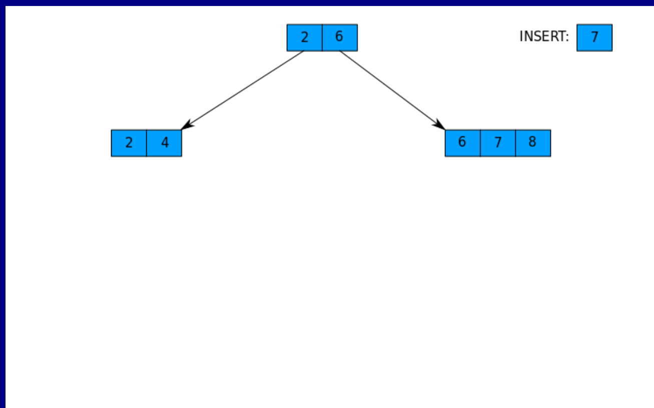
Пример вставки в B+ дерево

2	4	6	8

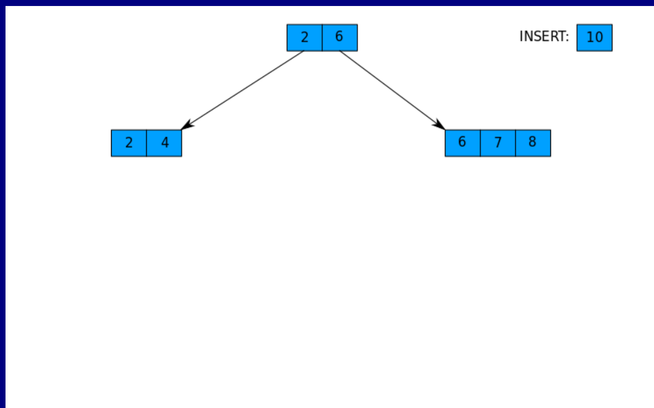
INSERT: 7

Пример вставки в B+ дерево

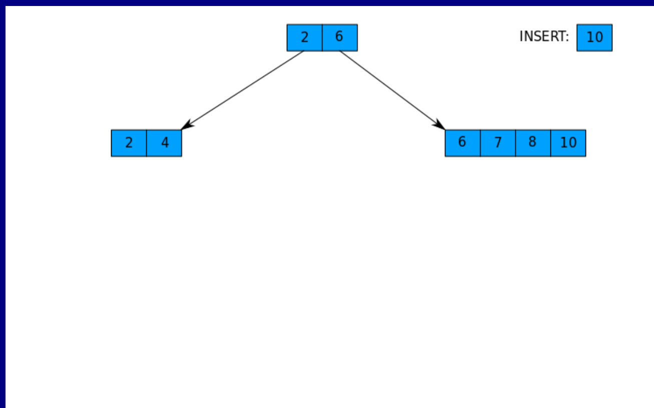
Пример вставки в B+ дерево



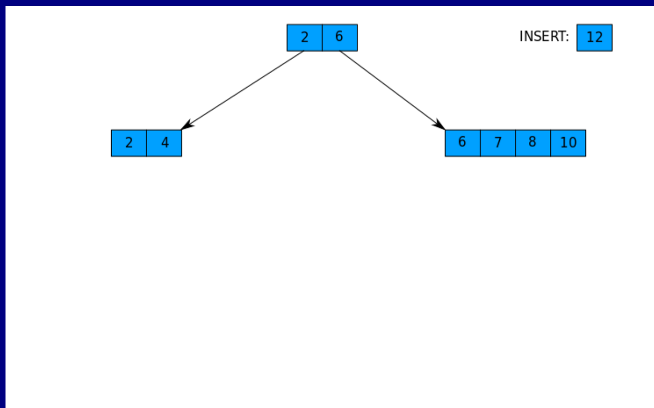
Пример вставки в B+ дерево



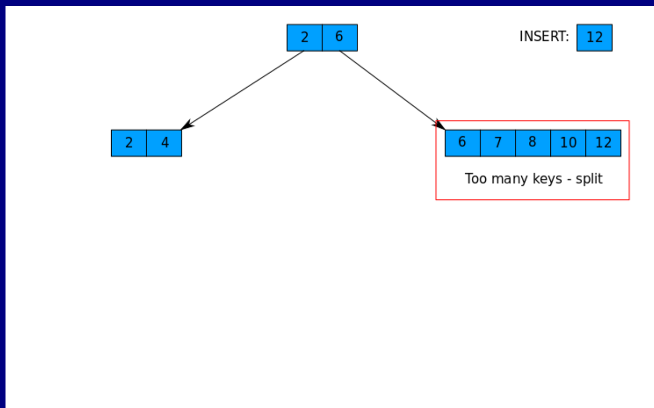
Пример вставки в B+ дерево



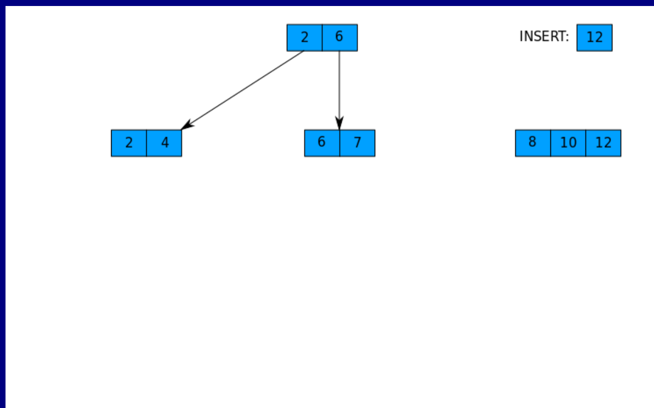
Пример вставки в B+ дерево



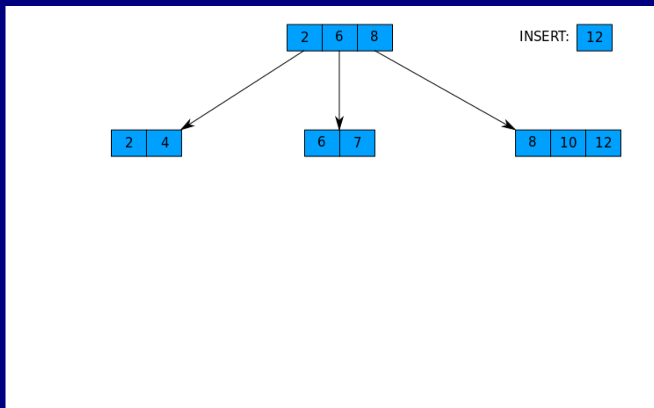
Пример вставки в B+ дерево



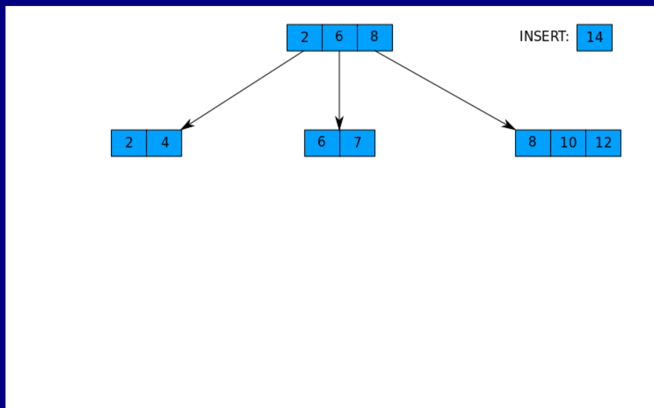
Пример вставки в B+ дерево



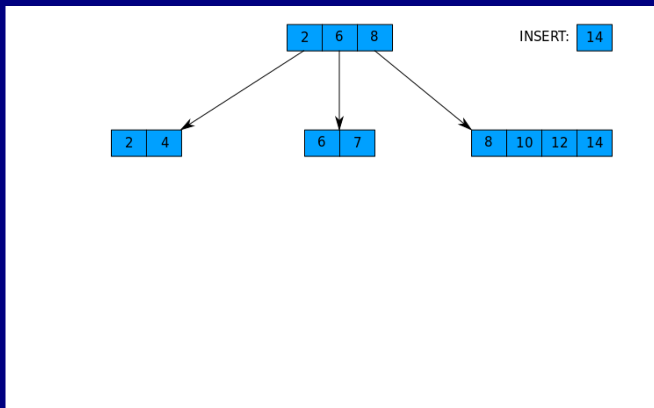
Пример вставки в B+ дерево



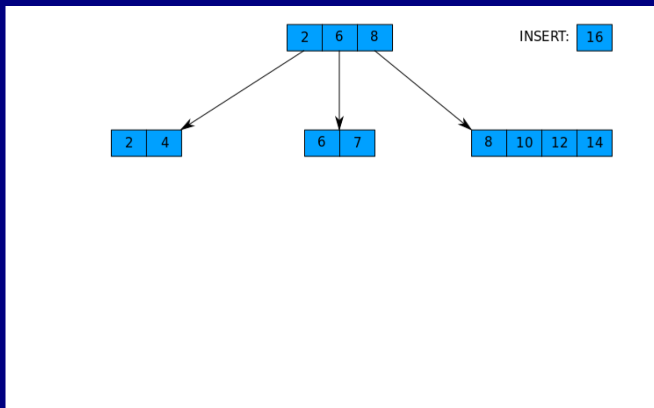
Пример вставки в B+ дерево



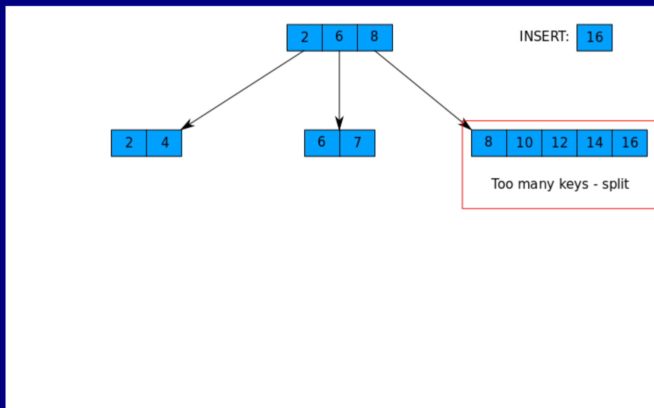
Пример вставки в B+ дерево



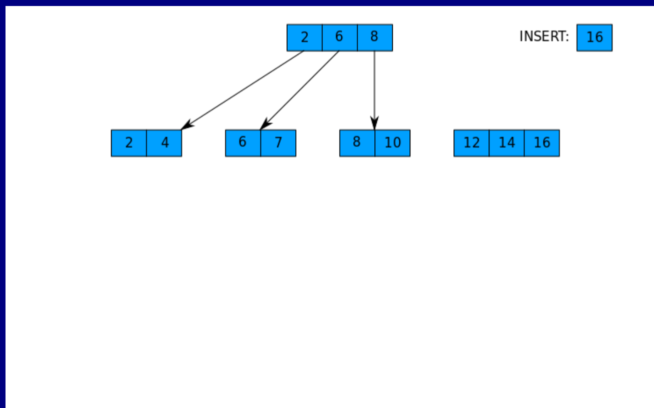
Пример вставки в B+ дерево



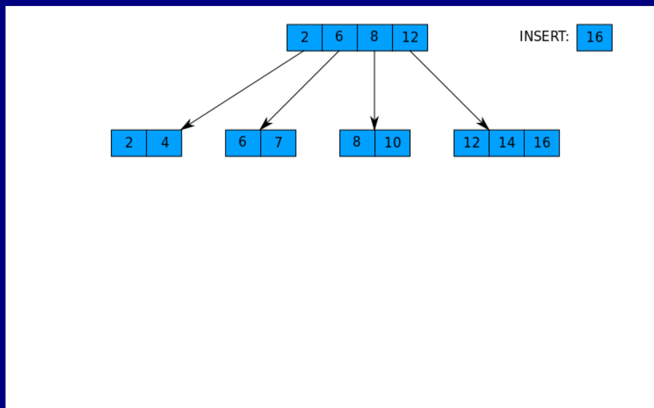
Пример вставки в B+ дерево



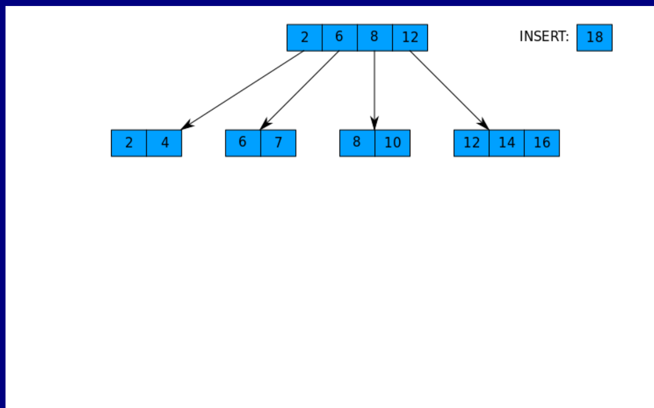
Пример вставки в B+ дерево



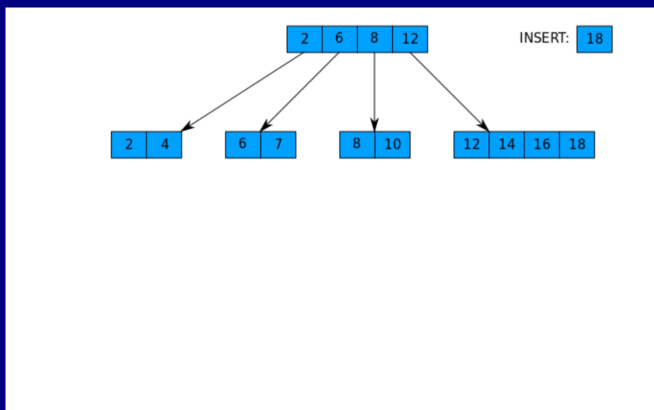
Пример вставки в B+ дерево



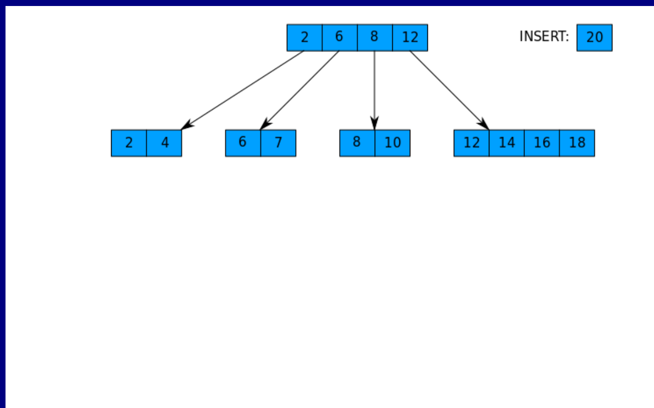
Пример вставки в B+ дерево



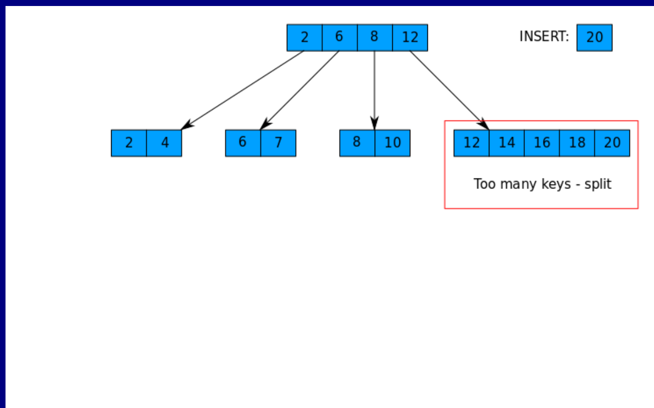
Пример вставки в B+ дерево



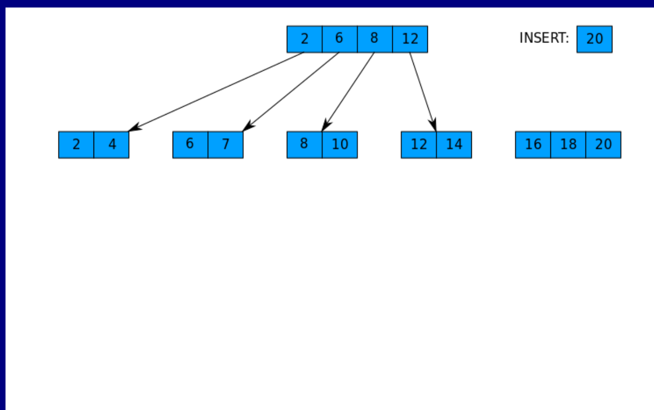
Пример вставки в B+ дерево



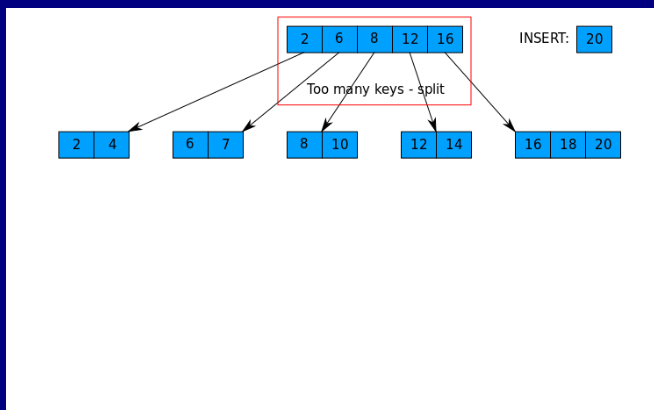
Пример вставки в B+ дерево



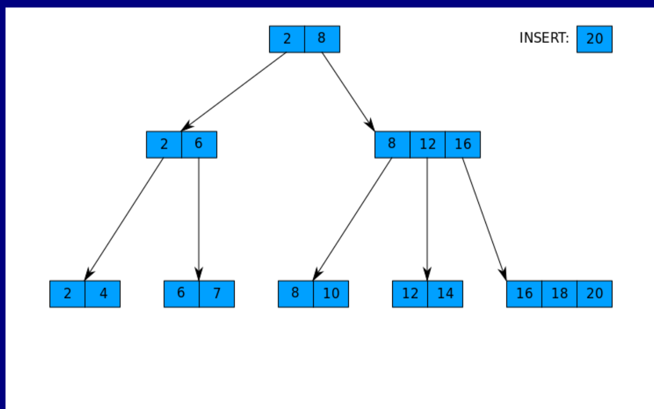
Пример вставки в B+ дерево



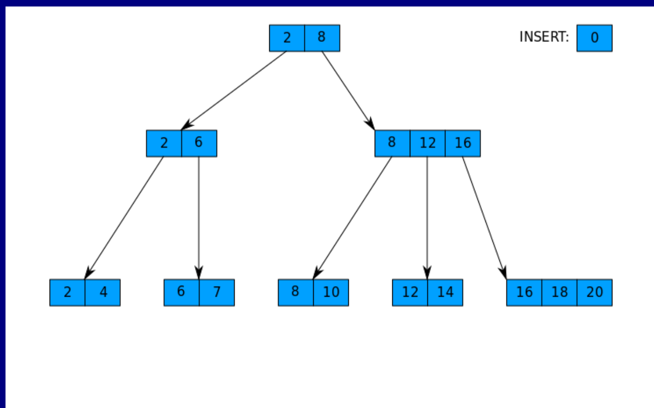
Пример вставки в B+ дерево



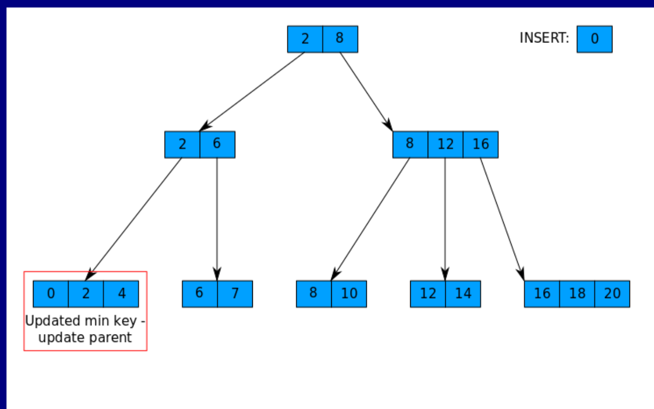
Пример вставки в B+ дерево



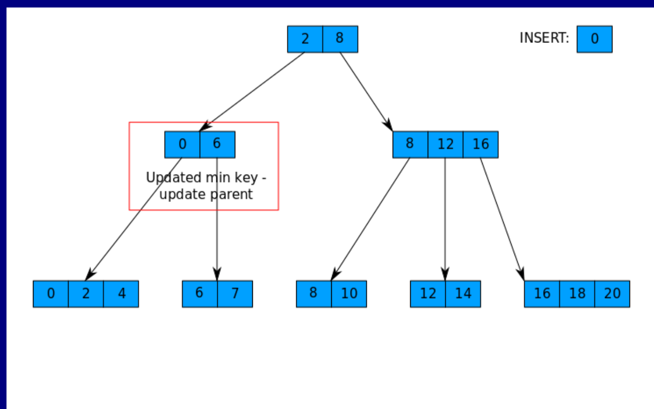
Пример вставки в B+ дерево



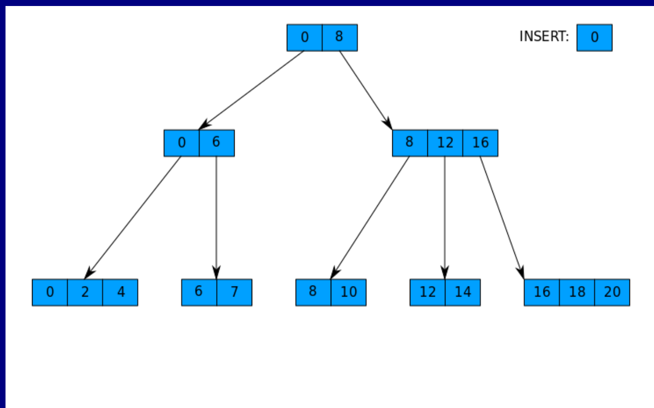
Пример вставки в B+ дерево



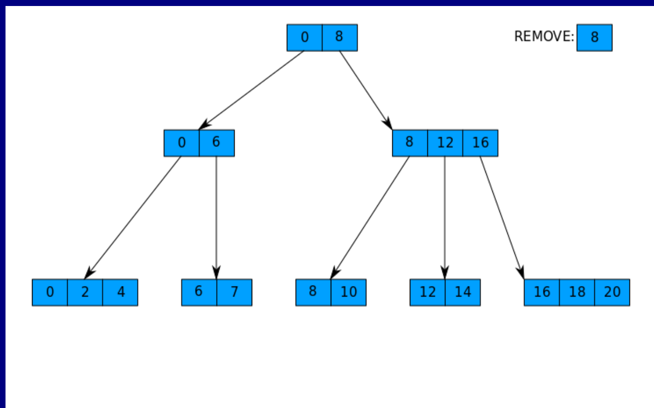
Пример вставки в B+ дерево



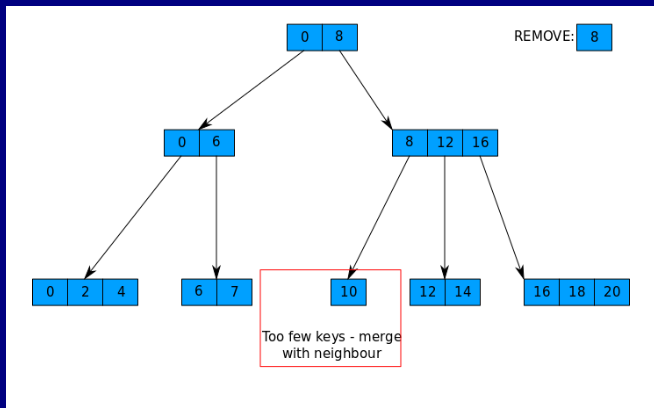
Пример вставки в B+ дерево



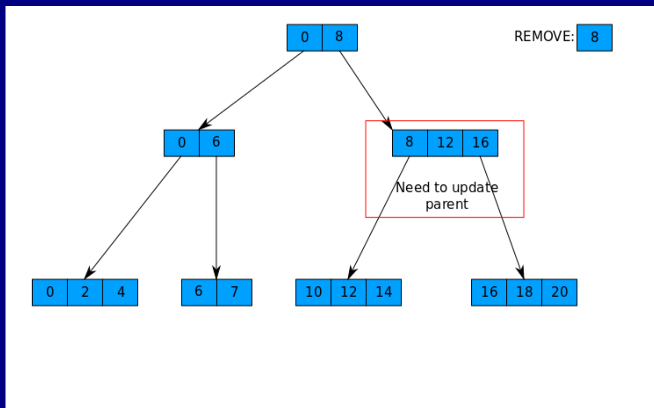
Пример удаления из B+ дерево



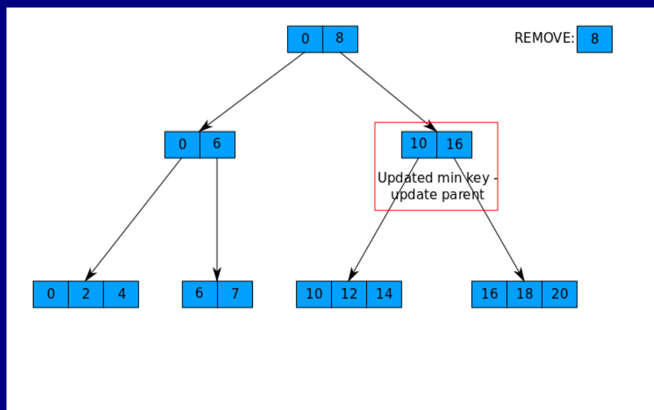
Пример удаления из B+ дерево



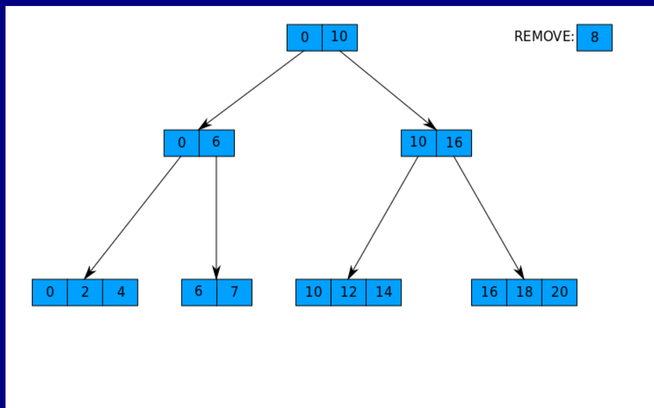
Пример удаления из B+ дерево



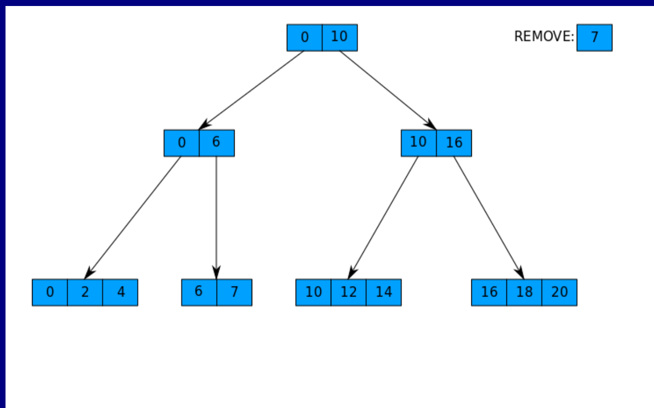
Пример удаления из B+ дерево



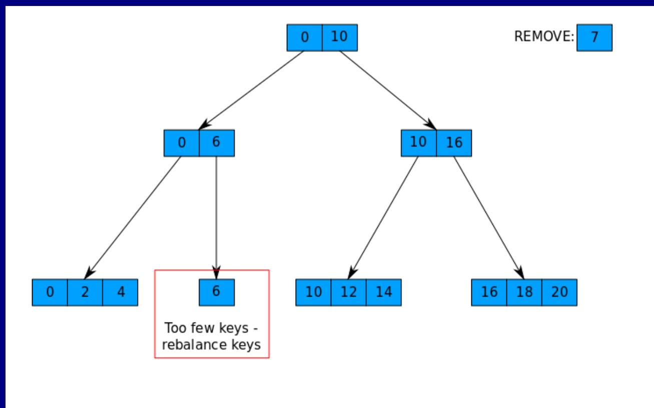
Пример удаления из B+ дерево



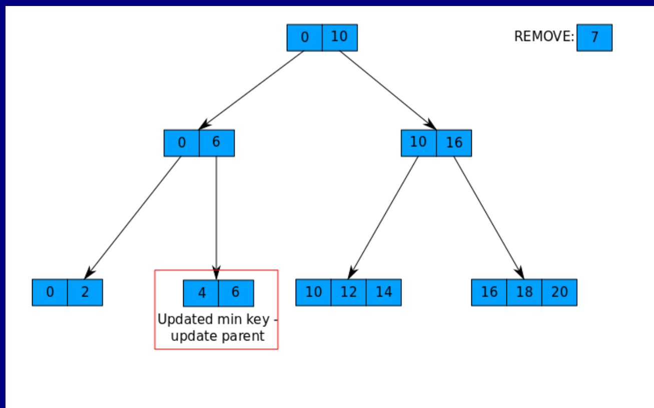
Пример удаления из B+ дерево



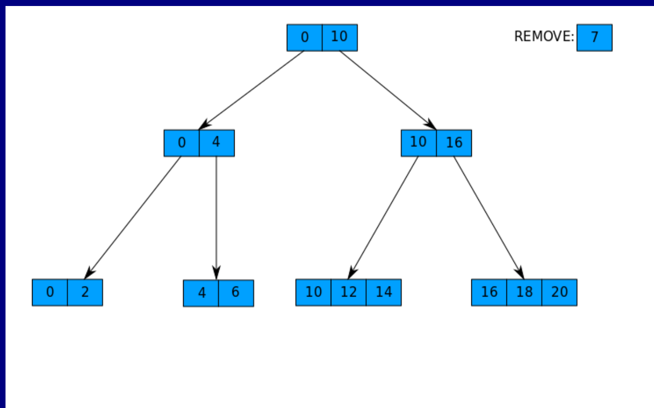
Пример удаления из B+ дерево



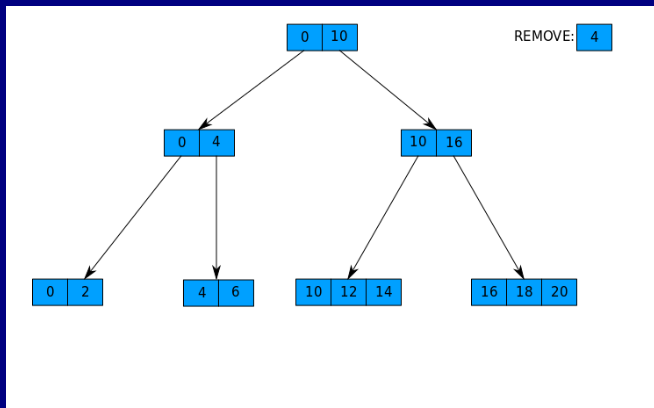
Пример удаления из B+ дерево



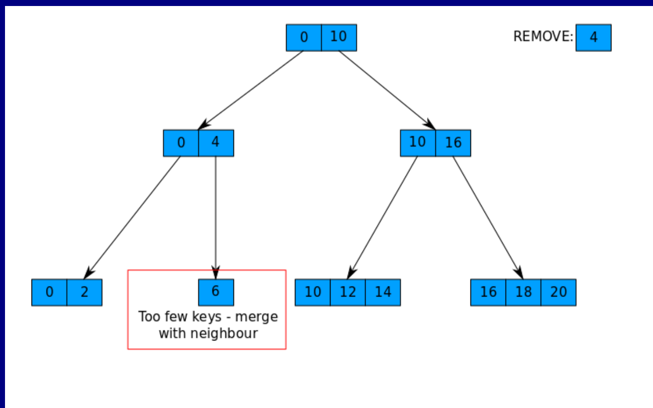
Пример удаления из B+ дерево



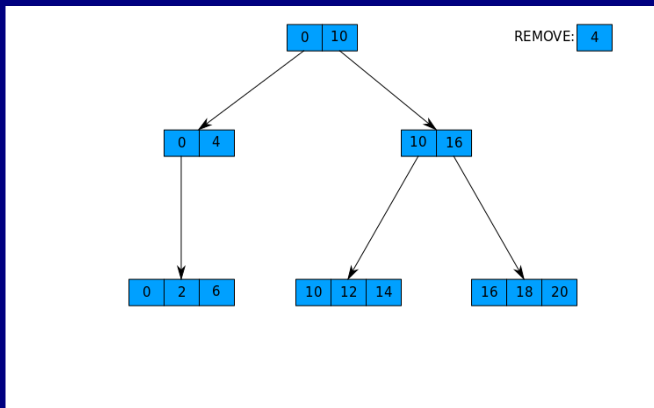
Пример удаления из B+ дерево



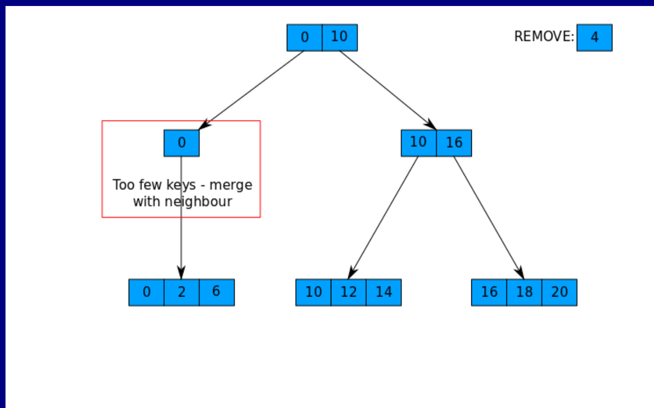
Пример удаления из B+ дерево



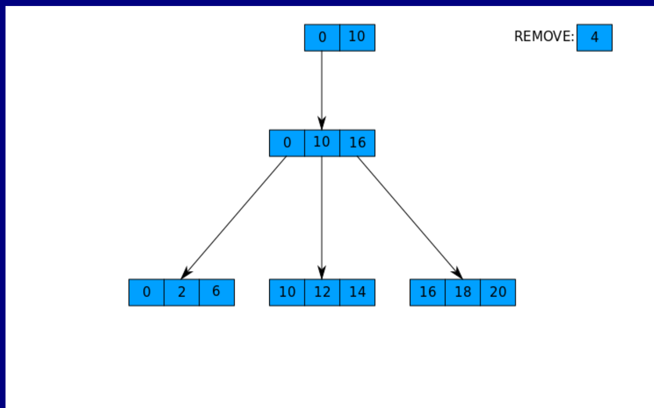
Пример удаления из B+ дерево



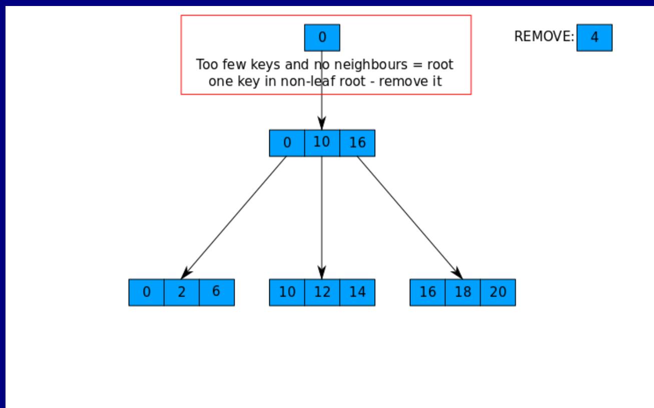
Пример удаления из B+ дерево



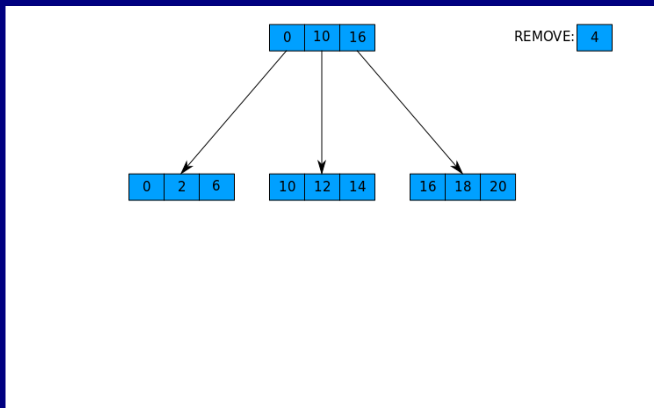
Пример удаления из B+ дерево



Пример удаления из B+ дерево



Пример удаления из B+ дерево



Параллельная работа с B+ деревьями

- Естественно ФС могут обрабатывать несколько запросов параллельно
  - деревья традиционно считаются очень недружественными структурами по отношению к параллельной обработке;
  - все потоки должны пройти через корень, для которого может потребоваться синхронизация - корень узкое место.
- Для B+ деревьев существуют неплохие схемы блокировок
  - классический вариант (B-link деревья) описан в статье Лемана и Яо;
  - Efficient locking for concurrent operations on B-trees.

Log-Structured Merge Tree (LSM)

- Недостатки B+ деревьев:
  - вставка и удаление в/из B+ дерева сложные;
  - параллельная работа с B+ деревьями сложная.
- Можно сделать проще:
  - пожертвовав алгоритмической сложностью поиска;
  - получив в замен "эффективную" параллельную работу.

Нам понадобятся

- Упорядоченный словарь в памяти с операцией вставки:
  - если ключ уже есть в словаре, то нужно заменить занчение;
  - подойдет любое дерево поиска, но плохо для параллельной обработки;
  - популярный вариант Skiplist - сравнительно легко параллелится.
- Обратите внимание:
  - нам не нужна операция удаления;
  - но будет полезной операция обмена двух словарей.

Нам понадобятся

- Упорядоченный словарь на диске:
  - мы должны уметь создавать словарь из *упорядоченного* набора пар (ключ, значение);
  - ветвистое дерево поиска (на подобие B+ дерева) очень легко построить из упорядоченного набора пар;
  - естественно, словарь должен поддерживать поиск;
  - а так же итерацию по ключам в отсортированном порядке.
- Обратите внимание:
  - нам не нужны операции вставки и удаления из словаря;
  - мы создаем словарь за раз, после чего он никогда не меняется.

Собираем все вместе

- Возьмем один словарь в памяти и несколько (скажем N) словарей на диске:
  - начальное состояние когда все эти словари пустые;
  - всех назовем и пронумеруем, словарь в памяти будет C_0, первый словарь на диске будет C_1, последний словарь на диске будет C_N.
- С каждый словарем кроме последнего свяжем ограничение на максимальный размер словаря:
  - обычно размеры словарей растут как члены геометрический прогрессии;
  - если размер словаря в памяти S и знаменатель прогрессии q, то размер C_1 ограничивается Sq, размер C_2 ограничивается Sq^2 и так далее;
  - будем называть эти ограничения S_i, где i - номер словаря.

Вставка в LSM

- Вставка всегда осуществляется в C_0 (т. е. словарь в памяти):
  - естественно C_0 при этом растет, и когда-нибудь станет больше S_0;
  - в этом случае нам нужно слить C_0 и C_1 в один словарь на диске;
  - полученный словарь заменит старый C_1, а C_0 нужно опустошить.
- Аналогичным образом мы поступаем когда C_i перерастает S_i
  - сливаем C_i и C_{i+1} в новую версию C_{i+1};
  - разница только в том, что теперь оба словаря изначально на диске.

Удаление из LSM

- У нас нет ни операции удаления из C_0 ни для всех остальных C_i
  - вместо этого мы можем вставить в C_0 ключ, который мы хотим удалить, но со специальным маркером;
  - каждый раз, когда мы видим этот маркер мы знаем, что ключ удален.
- Можно ли физически удалить ключ и освободить место?
  - когда мы сливаем C_i и C_{i+1} мы можем увидеть два одинаковых ключа;
  - в результирующий словарь нужно добавить только один, ключ из C_i;
  - если мы сливаем C_{N-1} и C_N, то ключи с маркером выписывать не нужно - таким образом мы физически удалим ключ из словаря.

Поиск в LSM

- Поиск в LSM делаем последовательно от C_0 до C_N:
  - до тех пор пока не найдем нужный ключ;
  - если ключ с маркером, то значит ключа нет и можно остановиться.
- Поиск в LSM алгоритмически хуже, чем в B+ дереве, что взамен?
  - простота - LSM деревья используют только простые примитивы (никакой перебалансировки и прочего);
  - все C_i для $i \in [1..N]$ не изменяются - не нужна синхронизация (главное, чтобы их не удалили пока, кто-то в них ищет);
  - C_0 не требует операции удаления - проще реализовать параллельную версию.

Финальные замечания про LSM

- Была описана только общая идея, детали могут отличаться:
  - можно использовать разные структуры данных для C_0 и C_i;
  - можно использовать несколько словарей в памяти (зачастую их два, это нужно чтобы не останавливать вставки/удаления пока происходит слияние C_0 и C_1);
  - можно задавать разные ограничения на размер словарей и сливать больше чем два словаря за раз.
- Используется довольно широко:
  - LevelDB/RocksDB, вероятно, еще в очень многих Key-Value/NoSQL/Any other buzzword;
  - Apache Cassandra;
  - Apache HBase/BigTable.

Надежность и консистентность

- Нам бы хотелось, чтобы ФС были надежными - не теряли пользовательские данные
  - мы не будем обсуждать вариант, когда вы уронили диск в костер или смыли;
  - т. е. мы не рассматриваем ситуацию, когда диск вышел из строя.
- Какие же проблемы у нас остаются?
  - неожиданное отключение питания, т. е. работа ФС была прервана неожиданно.

Какие могут возникнуть проблемы?

- Вспомните работу с B+ деревом:
  - мы можем прервать вставку/удаление в/из B+ дерева на середине;
  - т. е. когда мы добавили/удалили ключ в/из листа, но не восстановили инварианты дерева до конца;
  - в лучшем случае дерево не будет корректным B+ деревом, в худшем мы потеряем часть ключей и произойдет утечка места на диске.
- Удаление файла из каталога:
  - нужно удалить имя файла из списка имен файлов/каталогов в каталоге;
  - нужно освободить Inode, используемый для файла;
  - нужно освободить место занятое содержимым файла.

Доступные гарантии

- Мы можем считать, что запись одного сектора (512 байт) атомарна:
  - сектор либо полностью записался либо не записался вообще;
  - то что вы послали устройству команду на запись одного сектора, не значит, что запись реально произошла.
- Существует команда - барьер:
  - вы можете послать устройству специальную команду сброса кеша/буферов;
  - по завершении команды, гарантируется, что все записи отправленные до нее реально завершились.

Soft update и fsck

- Основная идея: выполняем все операции в таком порядке, чтобы:
  - если нас прервут, то ФС будет в рабочем (пусть и не очень консистентном состоянии);
  - все неконсистентности можно найти и поправить.
- Утилита fsck сканирует всю ФС, ищет и исправляет проблемы:
  - должна просканировать всю ФС, что может занять время;
  - может иногда помочь в случае поломки диска, о которых мы не говорим.

Пример: удание файла из каталога

- В первую очередь удаляем имя файла из списка файлов/каталогов в каталоге
  - если нас прервут сразу после этой операции, то у нас один из Inode-ов может остаться занятым, хотя ссылок на него не будет;
  - место занятое файлом не будет освобождено;
  - в остальном состояние ФС в рабочем состоянии;
  - fsck может найти Inode-ы и сектора на которые нет ссылок и освободить их.
- Если сначала освободить Inode или место занятое файлом?
  - тогда в каталоге останется ссылка на невалидный Inode и при чтении или, еще хуже, записи мы обратимся к чужим секторам на диске.

Финальные замечания про soft update

- Soft update и fsck сравнительно сложны в реализации:
  - требуется аккуратно продумать порядок обновления и строго его соблюдать - заставлять диск сбрасывать кеш.
- Soft update и fsck работают сравнительноо медленно:
  - постоянные сбросы кешей диска не способствуют скорости работы;
  - fsck должен сканировать всю ФС, чтобы понять на что есть ссылки, а на что нет.

Write Ahead Log (WAL, Journal)

- Перед тем как выполнить какую-то операцию мы можем записать в специальное место диска (журнал), что мы хотим сделать
  - обновляем структуры ФС только после того как запись была зафиксирована на диске;
  - записи должны быть структурированы так, чтобы мы могли отличать целые законченные записи от незаконченных (например, мы можем в конце записи добавить специальный сектор - признак завершения записи).
- Что делать если нас прервали?
  - мы просто повторяем операции записанные в журнале ("проигрываем" журнал);
  - после чего очищаем журнал.

Идемпотентность записей в журнале

- Что если нас прервут, пока мы проигрываем запись из журнала?
  - если записи в журнале идемпотентны, то ничего;
  - т. е. при следующем запуске мы заново "проимгрываем" журнал.
- Операция идемпотентна, если мы можем применить ее несколько раз без изменений результата
  - записать данную порцию данных в данный сектор диска - идемпотентная операция;
  - мы можем повторять запись сколько угодно раз, практически, без последствий.

Copy-On-Write

- Вместо того, чтобы обновлять данные на диске In-Place будем создавать копии в памяти, обновлять копии и выписывать копии в новое место на диске
    - после того как у нас готова исправленная версия данных нужно "просто" перенаправить ссылку со старой версии данных в новую;
    - перенаправить ссылку = записать, т. е. опять Copy-On-Write;
    - когда эта перезапись должна остановиться? Когда мы дойдем до, некоторого "корня" (например, корня дерева, или суперблока ФС), который можно записать атомарно.

Применение COW

- Copy-On-Write используется в ZFS и Btrfs (и не только):
  - Btrfs использует COW B+ деревья (Ohad Rodeh, BTRFS: The Linux B-tree Filesystem, там же ссылки на алгоритм COW B+ деревьев);
  - ZFS - другой, уже классческий, пример (Jeff Bonwick, тот же, что придумал SLAB);
  - обе ФС умеют делать snapshot-ы (версионирование) - приятный бонус использования COW.

Q&A