Как написать хранилище БД и не сойти с ума

Журнал изменений в виде дерева с функциональным подходом

Проблемы

- B-tree (блочное дерево) и братья (B+-tree и так далее)
  - Великолепно работают добавления с последовательными ключами
  - Отвратительно работают добавления со случайными ключами (индексы, графы)
    - В SQL Server значения GUID создаются “псевдослучайно” (последовательно)
    - Фрагментация
      - нарушается последовательный порядок страниц
      - появляются “дырки” в карте страниц после установления баланса
    - B-tree можно считать статической структурой данных (как k-d-tree)
  - Изоляция сложна (ARIES - 69 страниц)
    - Журнал дорог! (на 1 байт изменений у BerkeleyDB в районе 10 байт журнала)
  - Даже простое хранение сложно
    - Страница 8К, наш ключ - 8К+1 байт. Что делать?
    - Страница 8К, в странице 512 ключей по 4 байта. Утилизация < 50%.
  - Считается. что 10+К строк кода - нормально для B+-дерева без журнала

Как решают

- Все применяют "логарифмический метод"
  - Берем статическую структуру в качестве базовой
  - Делаем иерархию уровней в виде структур всё большего размера
    - РазмерУровня(i) = c*РазмерУровня(i-1) (c, обычно, 2)
      - Большее c - быстрое чтение (меньше уровней)
      - Меньшее c - быстрая запись (реже слияние больших данных)
  - Мы можем <u>вообще избежать добавления</u> в существующую структуру
    - сортированный массив: вставка дорога, построение из сортированных данных дешево
    - при добавлении элемента мы:
      - создаём дополнительный "отсортированный массив" длиной 1,
      - или сливаем отсортированные массивы
        - есть массив длиной 1 - делаем массив длиной 2, есть с длиной 2...
  - Tocutek fractal indices!
    - (примерно - фрактальные индексы сложнее, но идея та же)

Как решают

- Дерево в виде журнала (Log-structured merge tree):
  - Логарифмический метод, примененный к B-tree
  - Данные накапливаются в памяти, потом отправляются на диск, со слиянием со старыми данными
    - вместо удаления элемента мы пишем отметку об удалении
      - вставка большую часть времени!
      - создавая/изменяя последний уровень, мы удаляем элементы при слиянии
    - при слиянии уровня из памяти и уровня на диске в оригинальной статье использовалось изменение B-tree по протоколу ARIES
      - у нас много последовательных данных (сотни килобайт) - фрагментация меньше
      - нагрузка на журнал меньше (сразу много изменений в странице)
  - Одиночное чтение - идем по уровням до самого недавнего изменения
  - последовательное чтение (курсор) - слияние всех данных на диске

Как решают

- Tocutek fractal indices:
  - буквально, это сортированные массивы, где уровень с меньшим размером ещё и служит индексом в уровень большим размером
    - Ключи могут быть любого размера, хранятся последовательно
  - Данные пересоздаются вместо изменений! (персистентная структура данных)
  - Последнее обеспечивает управление синхронизацией с помощью версий (MVCC, multi version concurrency control)
  - Большие массивы пересоздаются экспоненциально реже, чем малые.
- SQLite4, LevelDB, RocksDB - LSM
  - сортированные (memcmp, другое практически не используется) данные с индексом для быстрого доступа
  - SQLite4 за счёт этого объединила индексы и таблицы (нет rowid, если есть primary key)
  - разница в обслуживании части в ОЗУ (LevelDB -skiplist. почему???)

Моё решение

- Словарь терминов
  - **файл хранилища** - файл с данными хранилища
    - содержит заголовок (р-р страницы), информацию об уровнях и данные
  - **страница** - последовательность байтов длиной 2^L, расположена на кратной размеру позиции в файле
  - **блок** - последовательность страниц в файле, длина произвольна
  - **последовательность** - набор (не последовательных) блоков, содержащих данные
    - данные внутри последовательности не имеют границ, ключ может пересечь блок
    - количество блоков не более некой константы
  - **уровень** - 1..K последовательностей, где
    - последовательности i=1..K-1 содержат ключи и “указатели” на начало ключей в последовательности i+1 и
    - последовательность с номером K содержит ключи и данные
    - уровень содержит информацию о количестве ключей и размерах ключей/данных
  - **иерархия уровней** - 1..N записей об уровнях

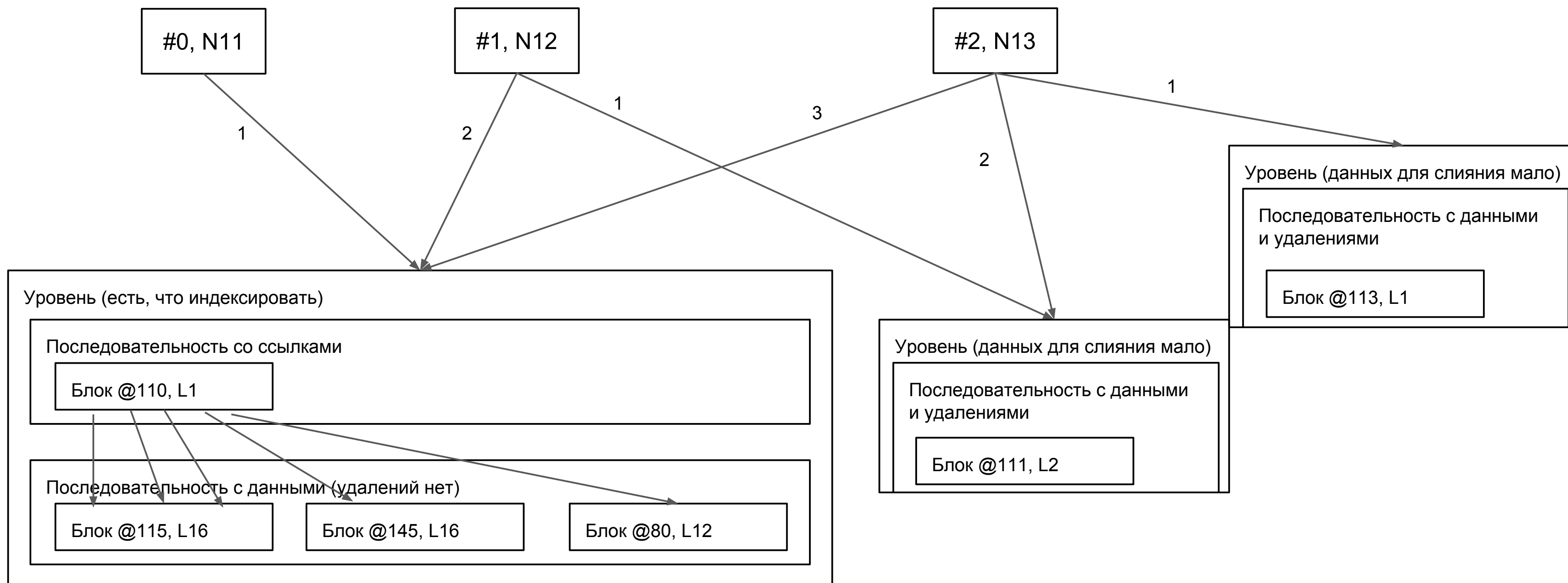
Моё решение

- ➔ Общая структура файла

- ➔ При фиксации (commit):
  - выбираем самую старую транзакцию (меньший последовательный номер)
  - переписываем по её физ. номеру новую информацию (и синхронизируем)
  - несколько копий - надежность (durability)
  - информация разделяется между транзакциями

Моё решение

➔ Как части связаны друг с другом



Моё решение

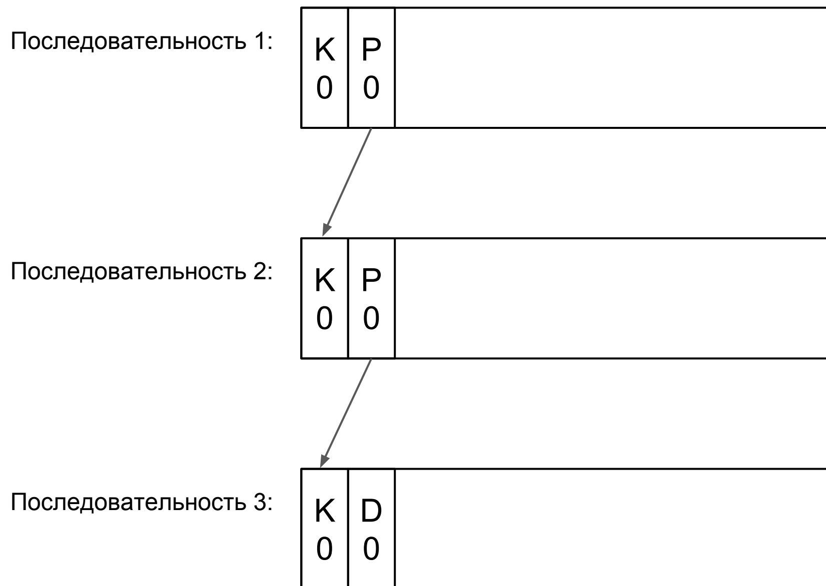
- Дерево в виде журнала (LSM), с отличиями:
  - Мы не меняем данные на месте, мы создаём новые
    - Отсюда MVCC
    - Сколько угодно читателей и писателей, без взаимной блокировки
      - синхронизация писателей дело уровня выше (сервера, кода пользователя)
  - Каждый уровень дерева представляет собой (структуру, похожую на) B-tree, но построеное слиянием, а не вставками
    - широкое дерево - быстрая навигация
  - Всё вместе позволяет лучше управлять памятью!
    - индексы и данные хранятся в O(1) (<= 8) последовательных блоков
      - $O(\log^2 N \log\log N)$ цена выделения памяти, и многих других операций
    - размер блока вычисляется заранее и может быть скорректирован позже
    - ключи и данные хранятся последовательно, без дыр на заполнение страницы

Моё решение

- Построение “B-дерево-подобной структуры”:
  - Если в сливаемой последовательности мы ожидаем N элементов, то легко вычислить количество уровней дерева: $L=\log_{ветвистость}(N)$. $\log_{1024}(8844221)=3$ (2,3, округляем вверх)
    - уровни дерева хранятся в **последовательностях**
  - Ожидая S байт, легко вычислить размер блока: S/размер страницы/8 (округляем вверх)
  - Создаём L писателей, индекс ключа по модулю ветвистости K_i ставим 0.
  - Далее для всех сливаемых ключей:
    - При записи ключа и данных писателем i проверяем, равен ли K_i 0?
      - равен - пишем текущий ключ и позицию писателя i (как данные) в писатель i+1, с аналогичной проверкой
    - Пишем ключ и данные писателем i
    - Увеличиваем Ki: $K_i=K_i+1$ mod ветвистость

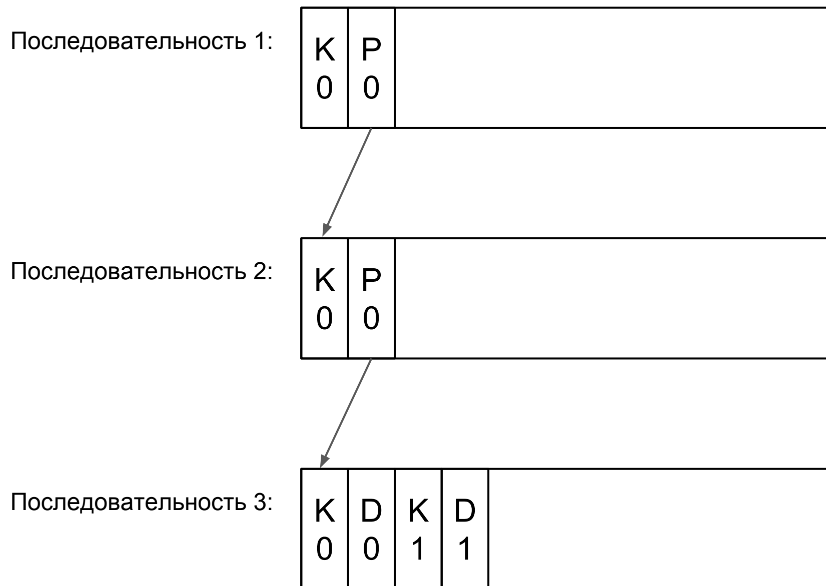
Пример записи данных

➔ Ожидается три последовательности, 8844221 элементов
  - записали ключ и данные с индексом 0 (самые первые)



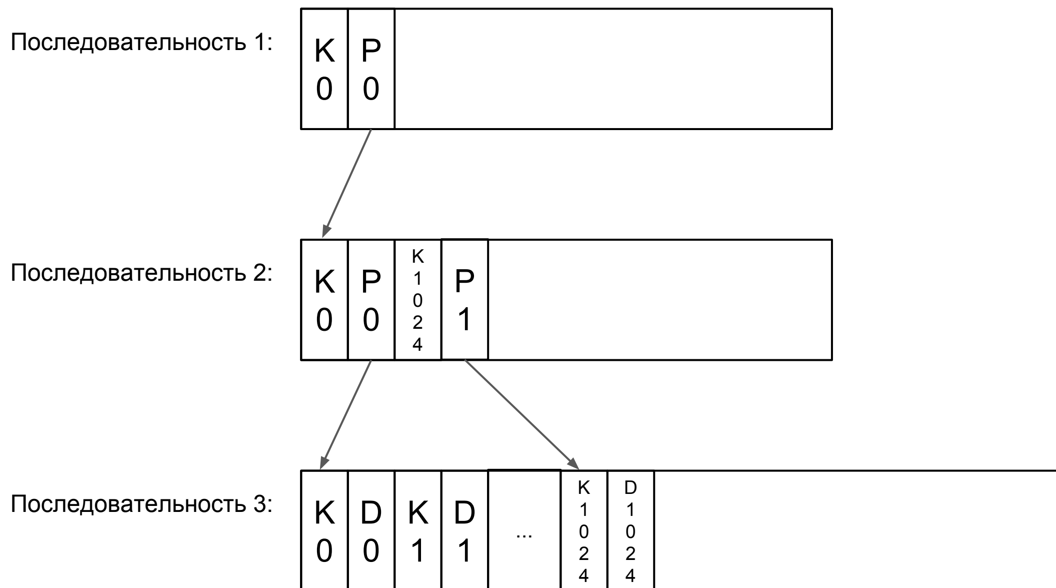
Продолжение примера (данные)

- Ожидается три последовательности, 8844221 элементов
  - записали ключ и данные с индексом 1



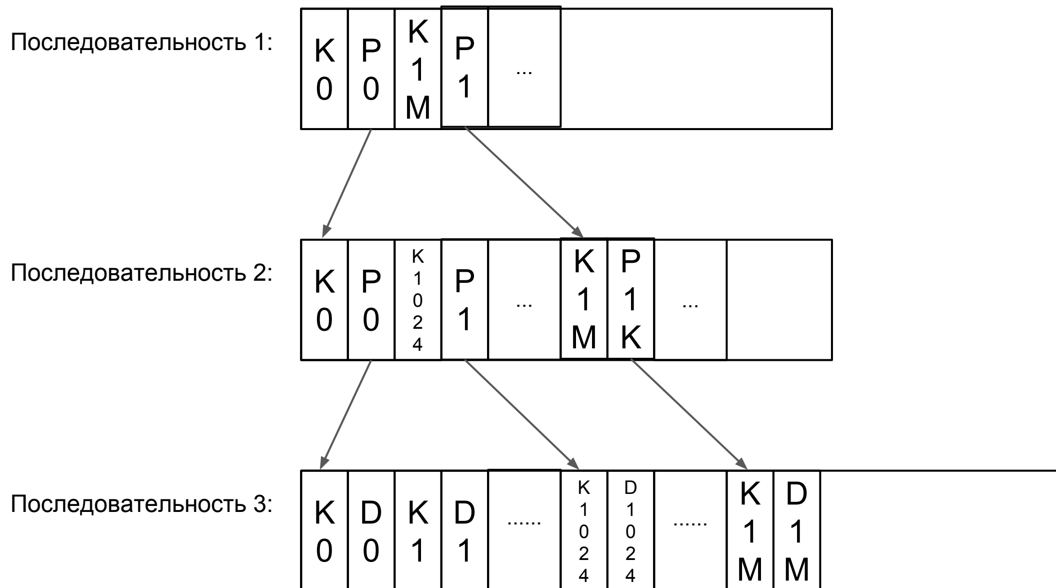
Продолжение примера (средний индекс)

- Ожидается три последовательности, 8844221 элементов
  - записали ключ и данные 1024 (mod ветвистость = 0)



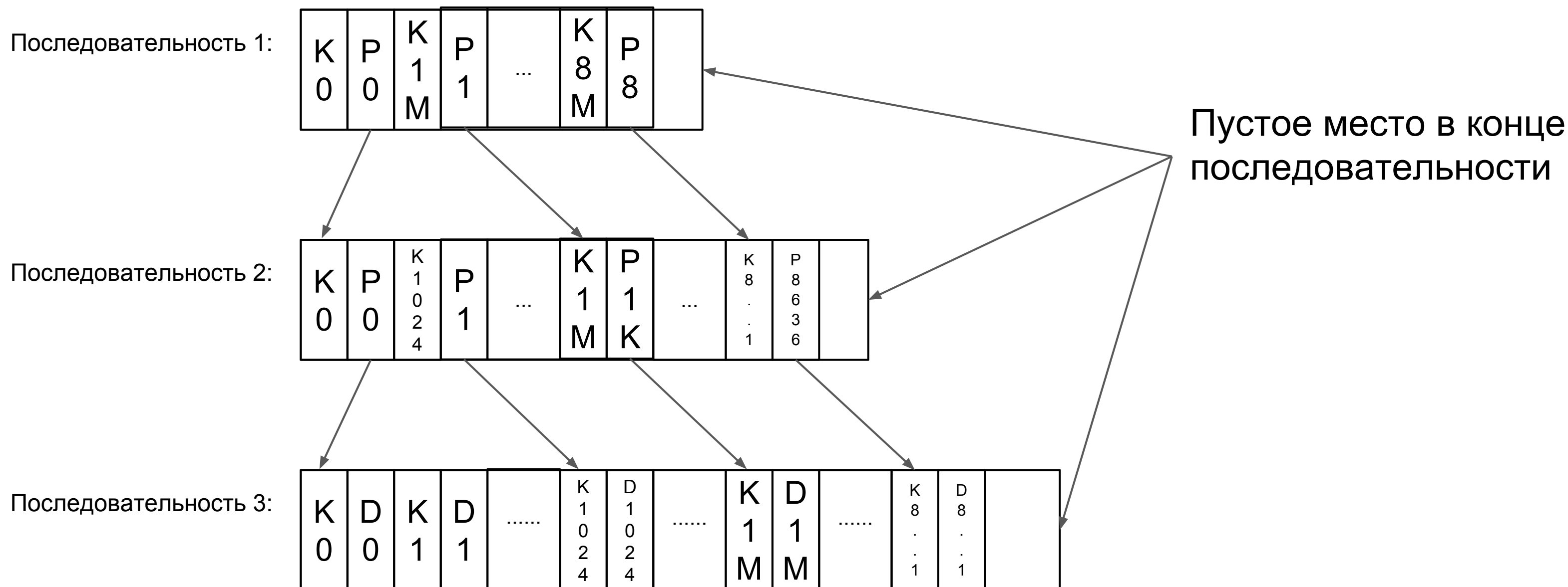
Продолжение примера (верхний индекс)

➔ Ожидается три последовательности, 8844221 элементов
  - записали ключ и данные 1048576 (ветвистость^2)



Завершение примера (всё записали)

- ➔ Ожидается три последовательности, 8844221 элементов
  - записали ключ и данные 8843211 (кое-что выпало из-за удаления)



Архитектура

- Архитектурно:
  - центральный процесс, ведущий информацию о БД и блокирующий операции:
    - распределение памяти
      - читатель требует отсутствия изменений в читаемых данных
      - писатель требует отсутствия изменения в записанных данных
      - счётчик ссылок!
      - $O(\log^2 N \log\log N)$ сложность, по моим оценкам.
    - фиксации (commit)
      - фиксация может потребовать слияния, которое потребует выделения памяти
    - Висит на канале (Control.Concurrent.Chan), выполняет команды
      - ответ через MVar в данных команды
      - Chan быстр -6000 тактов ЦП (сравнимо с запуском ветки OpenMP)
  - Процессы пользователя, которые оперируют транзакциями через вышеупомянутый канал

Уровни изоляции

- Легко поддерживаются два режима изоляции:
  - Read Committed (чтение данных, зафиксированных в параллельных транзакциях)
    - Перед чтением в транзакции просим у хранилища текущую иерархию уровней
    - считаем наши данные более свежими, чем данные из хранилища
    - читаем со слиянием наших и “старых” данных
    - фиксация на диск (commit) происходит с текущей иерархией уровней
  - Snapshot Isolation (транзакция не влияет на другие и другие не влияют на неё)
    - в начале транзакции мы забираем всю иерархию уровней и работаем только с ней
    - в момент фиксации на диск мы заменяем иерархию уровней в хранилище на вычисленную нами
- Остальные уровни
  - Read Uncommitted - требует общей работы всех транзакций с данными в памяти
  - Serializable - требует синхронизации выше уровня хранилища.

Текущее состояние

- Близится к состоянию “бета”
  - github
- Выполнил похожее до этого на C#
  - Великолепная скорость вставок
    - последовательные данные - сравнимо с BerkeleyDB (BDB отстаёт в два-три раза)
    - R-MAT граф - чем больше размер, тем сильнее отставание BDB
      - свыше двухсот пятидесяти (250+) раз на графах с сотней тысяч узлов!
      - на миллионах было бы ещё больше (скучно)
  - Приемлемая скорость чтения
    - O(logN) отставание от BDB
    - однако, если структура сложная, то у LSM локальность выше (фрагментация!)
      - чтение может даже обгонять BDB

...кто говорит „плагиат“, я говорю „традиция“.

- ➔ Документация SQLite и SQLite4 - клад просто!
- ➔ Длины ключей и данных хранятся в кодировке переменной длины
  - ◆ Кодировка переменной длины SQLite4 хорошо кодирует небольшие значения (0-240) и отлично кодирует большие (сравнимо с кодировкой “7бит+флаг” и UTF-8).
- ➔ Ключ может иметь признак “специальные данные” - нулевой длины или удаление
  - ◆ В обеих случаях данные отсутствуют
- ➔ Реализация LSM в SQLite4 имеет специальный ключ “удалить с таким префиксом”: но:
  - ◆ удаляет значения с префиксом, кроме первого и последнего и
  - ◆ плохо влияет на статистику - одно значение может удалять мегабайты данных
    - ● непонятно, когда сливать данные

Завершение

- Сейчас кода ~600 строк
  - недоделано, увы
  - нет последовательного чтения
  - нет специализированной структуры для байтовых массивов (сейчас Data.Map.Map)
    - должна серьёзно ускорить работу слияния и последовательного чтения
- Ожидается <1500 строк
  - Никак не 10+ тысяч
- Планы
  - тесты а-ля SQLite (проверка корректной работы в случае ошибок разного плана)
  - добавить к этому таблицы для лучшей типизации
  - транзакции с оптимизациями
  - в планах повторить Calvin сперва на одной машине, потом на нескольких
    - PAXOS implementation in 21-st century - я ржал