Purely practical data structures

Evgeny Lazin (@Lazin),
Fprog 2012-11

План

1. CIMDB, постановка задачи
2. Решение задачи, использующее могучую персистентную структуру данных
3. Еще больше интересных структур данных

CIMDB

- Main memory база данных, созданная в компании "Монитор Электрик" (Пятигорск)

CIMDB

1. Common Information Model (CIM).
2. Большой граф объектов, порядка 1KK элементов.
3. Используется для представления элементов энергосистемы.
4. Нужно поддерживать целостность данных.
5. Используется сложными расчетными задачами.

Немного истории

- Первая реализация - реляционная модель.
  - Каждая расчетная задача реализует чтение и кеширование объектов модели.
  - Множество дублирующих друг друга ad-hoc реализаций.
  - SQL JOIN
    - Дай мне все высоковольтные линии, связанные с подстанциями, принадлежащими определенной генерирующей компании.
  - Нет API

Немного истории

- Вторая реализация - client-side cache.
  - Решает проблемы:
    - изобретения велосипеда.
    - общий API.
    - могучие join-ы выполняются на клиенте, в памяти.
  - И создает:
    - Проблему целостности данных.
      - Инвалидация кэша
    - Shared mutable state.
      - Shared + mutable = PITA
    - Пессимистичные блокировки.
      - Блокировки реализованы на уровне БД
    - Использует очень много памяти.

Требования

- Распределенность.
- Транзакционность.
  - Нужно поддерживать ACID свойства.
  - Tradeoff - целостность достаточно поддерживать в рамках модели timeline consistency.
  - Оптимистичные блокировки.
- Performance.
  - Нужно очень быстро читать, на уровне предыдущей версии.
  - Писать можно не очень быстро.

Странное и необъяснимое

1. Инспектирование изменений и управление "миром".
   a. Нужно для технологических задач.
2. Очень быстрое создание снепшотов и веток изменений.
   a. Нужно для расчетных задач.

Возможные решения

Чтение данных через TCP/UDP/Pipes слишком медленно.

a. Время одного round trip-а - десятки/сотни микросекунд.
b. Требует кэширования на клиенте.
c. Кэш должен уметь prefetch.

Выход

Поместить клиент и сервер на одну машину, обмениваться данными через общую память.

Проблема синхронизации доступа к общей памяти.

Архитектура.

- Все объекты - неизменяемы. При изменении объекта, создается новая версия (MVCC).
- У объекта нет номера ревизии, версии или метки времени.
- Поиск объектов производится через индекс. Для каждой версии существует отдельный индекс.

Архитектура

Это позволяет естественным образом реализовать MVCC и избавиться от синхронизации при чтении.

Но теперь нам нужен очень эффективный индекс!

Архитектура

Персистентные структуры данных - естественный выбор в данной ситуации.

- Позволяют иметь одновременно несколько версий одной структуры данных, соответствующих разным моментам времени.
- Версии не являются полными копиями и разделяют часть данных.

Persistency

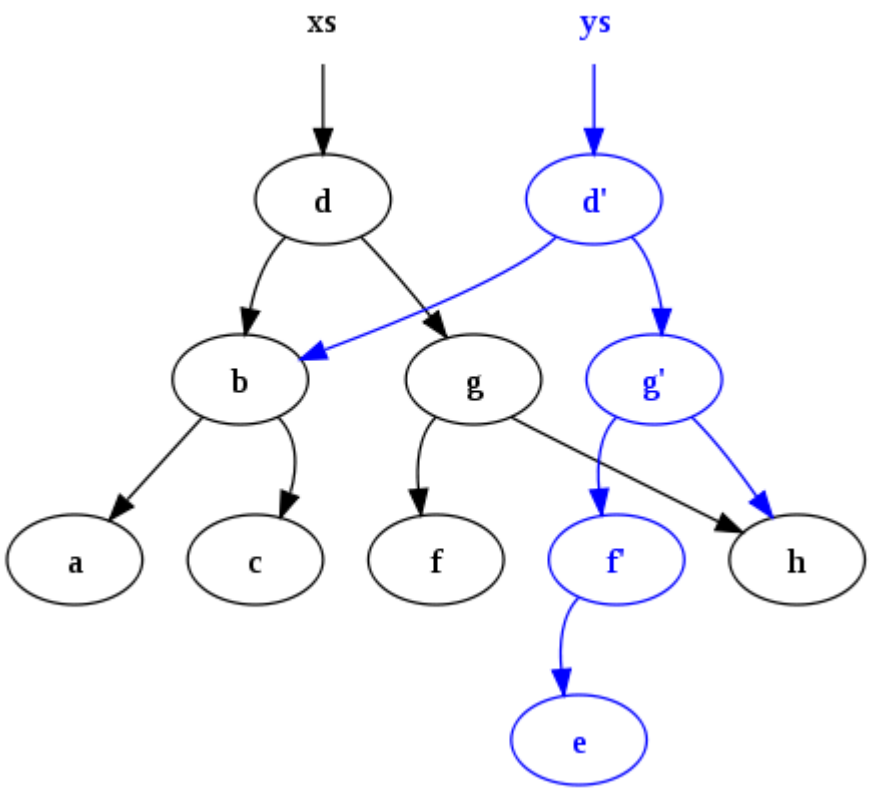
1. Partially persistent
   - Линейная история
   - Достаточно для реализации MVCC
2. Fully persistent
   - История изменений имеет форму дерева
   - Нужно для реализации веток изменений и изоляции транзакций

Хэш таблицы

- Очень быстрый поиск по ключу
- Коллизии
  - Коллизии возможны даже в том случае, если значения hash функции отличаются.
  - В нашем случае, у всех объектов есть уникальный hash.
- Невозможно сделать неизменяемой
  - Можно держать индекс в памяти каждого приложения, но это все усложняет.

Бинарные деревья + path copying

Функциональные версии различных сбалансированных деревьев.



Бинарные деревья

- Медленно
  - Скорость поиска по Id должна быть сравнима с hash таблицей.
- Глубина пропорциональна $\log_2 N$
- Для N = 1KK глубина дерева ~= 20

Префиксные деревья

- Глубина дерева ограничена длиной ключа
- Find, Insert, Delete - O(1)
- Компактное представление в памяти
- Можно сделать персистентным

Префиксные деревья

- Задача состоит в том, чтобы эффективно находить следующий узел.
- Можно использовать фрагмент ключа в качестве индекса массива.

Префиксные деревья

```
struct Node {
    Node[32] array;
}
```

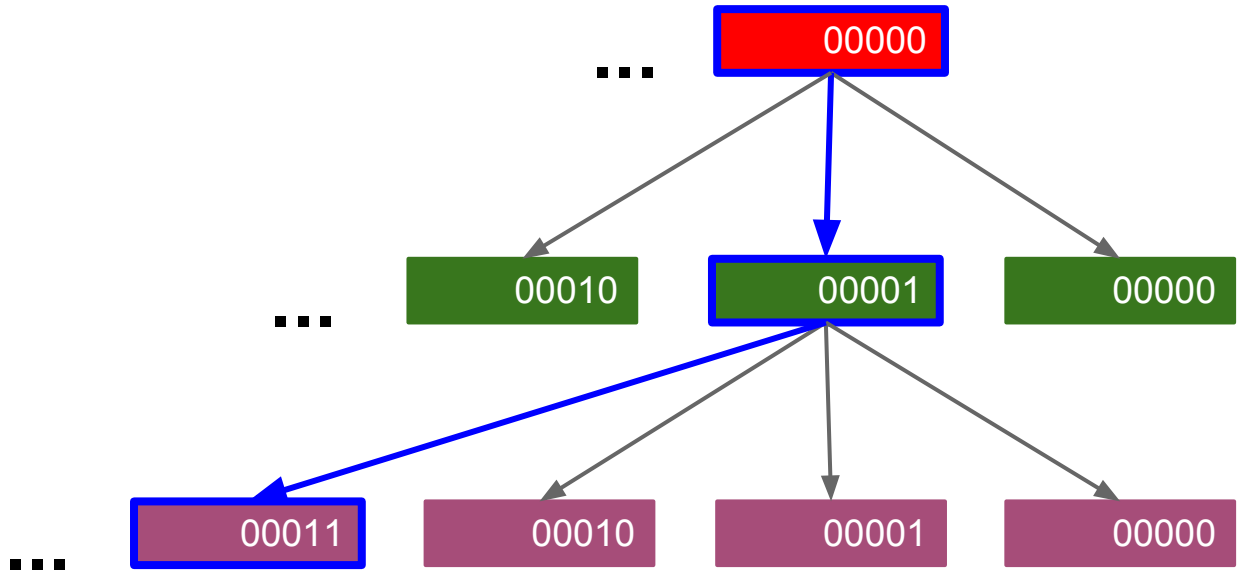
Префиксные деревья

Для поиска следующего элемента в дереве мы должны вычислить:

```
Find(int hash)
    index = hash & 0x1F
    next = this.array[index]
    if next != null:
        next.Find(hash >> 5)
    ...
```

Префиксные деревья

[00] [01001] [11001] [10001] [00011] [00001] [00000]



Префиксные деревья

- Существует множество реализаций префиксных деревьев, решающих проблему нулевых указателей
- Ideal Hash Trees. Phil Bagwell [2001]

Hash Array Mapped Trie

Реализации:

- Clojure's hash map
  - immutable
- Scala - Ctrie
  - mutable/concurrent
- Haskell - unordered-containers
- ...

HAMT

Каждый узел хранит:

- Количество дочерних элементов
- Массив дочерних элементов
  - Пара ключ - значение
  - Указатель на следующий узел
- Bitmap

HAMT

```
struct Node {
    int count;
    Node[count] array;
    int bitmap;
}
```

- Узел хранит только неравные null элементы
- А также битовую маску для вычисления индекса

HAMT

```
Find(int hash)
    int shift = hash & 0x1F
    int mask = (1 << shift) - 1
    int index = Popcnt(this.bitmap & mask)
    next = this.array[index]
    next.Find(hash >> 5)
```

Вычисляем значение ключа для поиска

HAMT

```
Find(int hash)
    int shift = hash & 0x1F
    int mask = (1 << shift) - 1
    int index = Popcnt(this.bitmap & mask)
    next = this.array[index]
    next.Find(hash >> 5)
```

(1 << shift) вернет слово, в котором будет установлен 1 бит, индекс которого равен индексу в несжатом массиве из 32х элементов

HAMT

```
Find(int hash)
    int shift = hash & 0x1F
    int mask = (1 << shift) - 1
    int index = Popcnt(this.bitmap & mask)
    next = this.array[index]
    next.Find(hash >> 5)
```

(1 << shift) - 1 - мы получаем маску, в которой установлены все биты, чей индекс меньше индекса искомого элемента

HAMT

```
Find(int hash)
    int shift = hash & 0x1F
    int mask = (1 << shift) - 1
    int index = Popcnt(this.bitmap & mask)
    next = this.array[index]
    next.Find(hash >> 5)
```

Popcnt - возвращает количество ненулевых бит в слове.

HAMT

```
Find(int hash = 5)
    int shift = 5 & 0x1F
    int mask = (1 << 5) - 1
    int index = Popcnt(100001b & 11111b)
    next = this.array[1]
    next.Find(hash >> 5)
```

```
bitmap =
00000000000000000000000000100001b

array = [a, b]
naive = [a, 0, 0, 0, 0, b, ...
```

HAMT

Можно не сжимать узлы, имеющие множество дочерних элементов.

Если предполагается наличие дубликатов, нужен еще один тип узла для разрешения конфликтов.

HAMT

Реализация на C# (прототип)

- В 2.5 - 3 раза медленнее чем Dictionary
- Расходует сопоставимое количество памяти
- Нет коллизий
- Нагружает GC

HAMT

Реализация на Си (production)

- В несколько раз быстрее чем Dictionary
- Использует подсчет ссылок и не нагружает GC
- Не использует атомарные инструкции (lock xchg и тп) - очень быстро работает в одном потоке

Pros

- Простота архитектуры
  - Клиент отправляет на сервер транзакции
  - В ответ получает указатель на root
  - Сообщает о переходе на новую версию
  - Очень похоже на VCS
- Сложные вещи реализуются очень просто
  - Очень дешево иметь множество слегка отличающихся версий одной структуры данных
  - Очень легко реализуется изоляция транзакций
- Нет блокировок
  - Только координация посредством сообщений

Cons

- Возможны утечки памяти
  - Но только если клиент неправильно работает
- Все еще низкая производительность в некоторых случаях :)

Ссылки

- Каждый объект может ссылаться на другие объекты.
- Физически, эти ссылки реализованы как массивы идентификаторов связанных объектов.
- Всегда есть обратные ссылки.

Ссылки

- Ходить по ссылкам через индекс - достаточно дорого.
- Можно ли хранить внутри объектов не только идентификаторы, но и указатели на другие объекты?

Ссылки

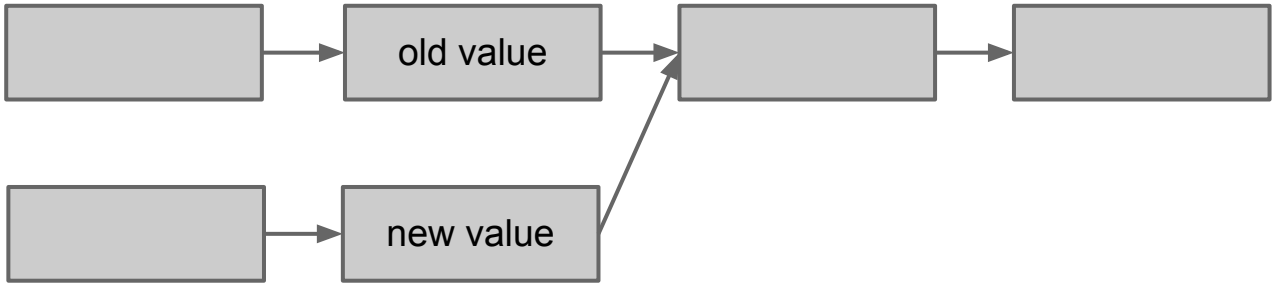
- Любую связную структуру данных можно сделать частично или полностью персистентной
- При определенных условиях - за O(1) времени и памяти
- "Making Data Structures Persistent" by James R. Driscoll, Neil Sarnak, Daniel D. Sleator, Robert E. Tarjan

Принцип работы

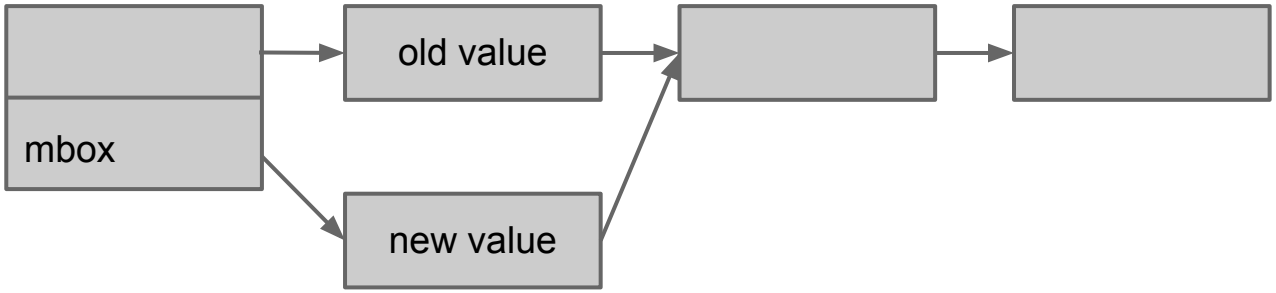
- Каждый объект может опционально хранить один или несколько указателей на связанные с ним объекты.
- Каждый объект получает дополнительное поле - modification box (m-box)
- m-box может хранить информацию об изменениях указателей
- С каждым изменением связан номер версии

Пример

immutable list update



mutable list update



Pros

- Быстрый переход по ссылкам
- Более быстрая проверка ссылочной целостности при commit-е

Cons

- Нужно больше памяти
- И процессорного времени при обновлениях
- Для чтения нужно знать номер ревизии
- Можно обеспечить только частичную персистентность, это означает что:
  - Никаких указателей в ветках
  - Никаких указателей в транзакциях

Вывод

Данный подход выгодно использовать только опционально, не для всех типов объектов и не для всех ссылок.

Вопросы?