Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет имени первого Президента России Б. Н. Ельцина»

УТВЕРЖДАЮ

Директор по образовательной деятельности

_____ С.Т. Князев

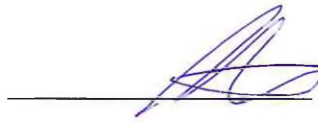
«7» сентября 2023 г.

Подсистемы хранения и извлечения данных

Учебно-методические материалы по направлению подготовки
09.03.03 Прикладная информатика
Образовательная программа «Прикладной искусственный интеллект»

Екатеринбург

РАЗРАБОТЧИКИ УЧЕБНО-МЕТОДИЧЕСКИХ МАТЕРИАЛОВ

Доцент кафедры информационных технологий и систем управления		М.В. Ронкин
Ассистент кафедры информационных технологий и систем управления		А.С. Аксенов

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ В АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. ИСПОЛЬЗОВАНИЕ МАССИВОВ И СВЯЗАННЫХ СПИСКОВ

Что такое структуры данных?

По сути, это способы хранить и организовывать данные, чтобы эффективней решать различные задачи. Данные можно представить по-разному. В зависимости от того, что это за данные и что вы собираетесь с ними делать, одно представление подойдёт лучше других.

Чтобы понять, почему так происходит, сперва поговорим об алгоритмах.

Алгоритмы

Алгоритм — такое хитроумное название для последовательности совершаемых действий.

Структуры данных реализованы с помощью алгоритмов, алгоритмы — с помощью структур данных. Всё состоит из структур данных и алгоритмов, вплоть до уровня, на котором бегают микроскопические человечки с перфокартами и заставляют компьютер работать. (Ну да, у Интела в услужении микроскопические люди. Поднимайся, народ!)

Любая данная задача реализуется бесконечным количеством способов. Как следствие, для решения распространённых задач изобрели множество различных алгоритмов.

Например, для сортировки неупорядоченного множества элементов существует до смешного большое количество алгоритмов:

- Сортировка вставками,
- Сортировка выбором,
- Сортировка слиянием,
- Сортировка пузырьком,
- Сортировка кучи,
- Быстрая сортировка,
- Сортировка Шелла,
- Сортировка Тима,

- Блочная сортировка,
- Поразрядная сортировка...

Некоторые из них значительно быстрее остальных. Другие занимают меньше памяти. Третьи легко реализовать. Четвёртые построены на допущениях относительно наборов данных.

Каждая из сортировок подходит лучше других для определённой задачи. Поэтому вам надо будет сперва решить, какие у вас потребности и критерии, чтобы понять, как сравнивать алгоритмы между собой.

Для сравнения производительности алгоритмов используется грубое измерение средней производительности и производительности в худшем случае, для обозначения которых используется термин «О» большое.

О большое

«О» большое — обозначение способа приблизительной оценки производительности алгоритмов для относительного сравнения.

О большое — заимствованное информатикой математические обозначение, определяющее, как алгоритмы соотносятся с передаваемым им некоторым количеством N данных.

О большое характеризует две основные величины:

Оценка времени выполнения — общее количество операций, которое алгоритм проведёт на данном множестве данных.

Оценка объёма — общее количество памяти, требующееся алгоритму для обработки данного множества данных.

Оценки делаются независимо друг от друга: одни алгоритмы могут производить меньше операций, чем другие, занимая при этом больше памяти. Определив свои требования, вы сможете выбрать соответствующий алгоритм.

Структуры данных позволяют производить 4 основных типа действий: доступ, поиск, вставку и удаление.

Идеальной структуры данных не существует. Вы выбираете самую подходящую, основываясь на данных и на том, как они будут обрабатываться.

Чтобы сделать правильный выбор, важно знать различные распространённые структуры данных.

Память

Компьютерная память — довольно скучная штука. Это группа упорядоченных слотов, в которых хранится информация. Чтобы получить к ней доступ, вы должны знать её адрес в памяти.

Структура данных

Структура данных — это контейнер, который хранит данные в определенном макете. Этот «макет» позволяет структуре данных быть эффективной в некоторых операциях и неэффективной в других.

Линейные, элементы образуют последовательность или линейный список, обход узлов линеен. Примеры: Массивы. Связанный список, стеки и очереди.

Нелинейные, если обход узлов нелинейный, а данные не последовательны. Пример: граф и деревья.

Основные структуры данных.

- Массивы
- Стеки
- Очереди
- Связанные списки
- Графы
- Деревья
- Префиксные деревья
- Хэш таблицы

Массивы

Массив — это самая простая и широко используемая структура данных. Другие структуры данных, такие как стеки и очереди, являются производными от массивов.

Каждому элементу данных присваивается положительное числовое значение (индекс), который соответствует позиции элемента в массиве. Большинство языков определяют начальный индекс массива как 0.

Бывают

- Одномерные, как показано выше.
- Многомерные, массивы внутри массивов.

Основные операции

- Insert-вставляет элемент по заданному индексу
- Get-возвращает элемент по заданному индексу
- Delete-удаление элемента по заданному индексу
- Size-получить общее количество элементов в массиве

Стеки

Стек — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Это не массивы. Это очередь. Придумал Алан Тюринг.

Примером стека может быть куча книг, расположенных в вертикальном порядке. Для того, чтобы получить книгу, которая где-то посередине, вам нужно будет удалить все книги, размещенные на ней. Так работает метод LIFO (Last In First Out). Функция «Отменить» в приложениях работает по LIFO.

Основные операции

- Push-вставляет элемент сверху
- Pop-возвращает верхний элемент после удаления из стека
- isEmpty-возвращает true, если стек пуст
- Top-возвращает верхний элемент без удаления из стека

Очереди

Подобно стекам, очередь — хранит элемент последовательным образом. Существенное отличие от стека – использование FIFO (First in First Out) вместо LIFO.

Пример очереди – очередь людей. Последний занял последним и будешь, а первый первым ее и покинет.

Основные операции

- Enqueue () — вставляет элемент в конец очереди
- Dequeue () — удаляет элемент из начала очереди
- isEmpty () — возвращает значение true, если очередь пуста
- Top () — возвращает первый элемент очереди

Связанный список

Связанный список – массив где каждый элемент является отдельным объектом и состоит из двух элементов – данных и ссылки на следующий узел.

Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Бывают

Однонаправленный, каждый узел хранит адрес или ссылку на следующий узел в списке и последний узел имеет следующий адрес или ссылку как NULL.

Двунаправленный, две ссылки, связанные с каждым узлом, одним из опорных пунктов на следующий узел и один к предыдущему узлу.

Круговой, все узлы соединяются, образуя круг. В конце нет NULL. Циклический связанный список может быть одно-или двукратным циклическим связанным списком.

Самое частое, линейный однонаправленный список. Пример – файловая система.

Основные операции

- InsertAtEnd — Вставка заданного элемента в конец списка
- InsertAtHead — Вставка элемента в начало списка
- Delete — удаляет заданный элемент из списка
- DeleteAtHead — удаляет первый элемент списка

- Search — возвращает заданный элемент из списка
- isEmpty — возвращает True, если связанный список пуст

Графы

Бывают:

- Ориентированный, ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами.
- Неориентированные, к каждому из ребер можно осуществлять переход в обоих направлениях.
- Смешанные

Встречаются в таких формах как

- Матрица смежности
- Список смежности

Общие алгоритмы обхода графа

- Поиск в ширину – обход по уровням
- Поиск в глубину – обход по вершинам

Деревья

Дерево - это иерархическая структура данных, состоящая из узлов (вершин) и ребер (дуг). Деревья, по сути связанные графы без циклов.

Древовидные структуры везде и всюду. Дерево скилов в играх знают все.

Типы деревьев

- N дерево
- Сбалансированное дерево
- Бинарное дерево
- Дерево Бинарного Поиска
- AVL дерево
- 2-3-4 деревья

Бинарное дерево самое распространенное.

«Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка»

Три способа обхода дерева

- В прямом порядке (сверху вниз) — префиксная форма.
- В симметричном порядке (слева направо) — инфиксная форма.
- В обратном порядке (снизу вверх) — постфиксная форма.

Trie (префиксное деревое)

Разновидность дерева для строк, быстрый поиск. Словари. Т9.

Хэш таблицы

Хэширование — это процесс, используемый для уникальной идентификации объектов и хранения каждого объекта в заранее рассчитанном уникальном индексе (ключе).

Объект хранится в виде пары «ключ-значение», а коллекция таких элементов называется «словарем». Каждый объект можно найти с помощью этого ключа.

По сути, это массив, в котором ключ представлен в виде хеш-функции.

Эффективность хеширования зависит от

- Функции хеширования
- Размера хэш-таблицы
- Метода борьбы с коллизиями

2. ИСПОЛЬЗОВАНИЕ ХЭШ-ТАБЛИЦ. АЛГОРИТМ И ПРИМЕРЫ

Хэш-функции — это функции, получающие на входе данные, обычно строку, и возвращающие число. При многократном вызове хэш-функции с одинаковыми входными данными она всегда будет возвращать одно и то же число, и возвращаемое число всегда будет находиться в гарантированном интервале. Этот интервал зависит от хэш-функции: в некоторых используются

32-битные целочисленные значения (то есть от 0 до 4 миллиардов), в других интервалы гораздо больше.

Мотивация использовать хеш-таблицы

Для наглядности рассмотрим стандартные контейнеры и асимптотику их наиболее часто используемых методов.

Контейнер \ операция	insert	remove	find
Array	O(N)	O(N)	O(N)
List	O(1)	O(1)	O(N)
Sorted array	O(N)	O(N)	O(logN)
Бинарное дерево поиска	O(logN)	O(logN)	O(logN)
Хеш-таблица	O(1)	O(1)	O(1)

Все данные при условии хорошо выполненных контейнерах, хорошо подобранных хеш-функциях

Из этой таблицы очень хорошо понятно, почему же стоит использовать хеш-таблицы. Но тогда возникает противоположный вопрос: почему же тогда ими не пользуются постоянно?

Ответ очень прост: как и всегда, невозможно получить все сразу, а именно: и скорость, и память. Хеш-таблицы тяжеловесные, и, хоть они и быстро отвечают на вопросы основных операций, пользоваться ими все время очень затратно.

Понятие хеш-таблицы

Хеш-таблица — это контейнер, который используют, если хотят быстро выполнять операции вставки/удаления/нахождения. В языке C++ хеш-таблицы скрываются под флагом unoredered_set и unordered_map. В Python вы можете использовать стандартную коллекцию set — это тоже хеш-таблица.

Реализация у нее, возможно, и не очевидная, но довольно простая, а главное — как же круто использовать хеш-таблицы, а для этого лучше научиться, как они устроены.

Для начала объяснение в нескольких словах. Мы определяем функцию хеширования, которая по каждому входящему элементу будет определять

натуральное число. А уже дальше по этому натуральному числу мы будем класть элемент в (допустим) массив. Тогда имея такую функцию мы можем за O(1) обработать элемент.

Теперь стало понятно, почему же это именно хеш-таблица.

Проблема коллизии

Естественно, возникает вопрос, почему невозможно такое, что мы попадем дважды в одну ячейку массива, ведь представить функцию, которая ставит в сравнение каждому элементу совершенно различные натуральные числа просто невозможно. Именно так возникает проблема коллизии, или проблемы, когда хеш-функция выдает одинаковое натуральное число для разных элементов.

Существует несколько решений данной проблемы: метод цепочек и метод двойного хеширования. В данной статье я постараюсь рассказать о втором методе, как о более красивом и, возможно, более сложном.

Решения проблемы коллизии методом двойного хеширования

Мы будем (как несложно догадаться из названия) использовать две хеш-функции, возвращающие взаимопростые натуральные числа.

Одна хеш-функция (при входе g) будет возвращать натуральное число s, которое будет для нас начальным. То есть первое, что мы сделаем, попробуем поставить элемент g на позицию s в нашем массиве. Но что, если это место уже занято? Именно здесь нам пригодится вторая хеш-функция, которая будет возвращать t — шаг, с которым мы будем в дальнейшем искать место, куда бы поставить элемент g.

Мы будем рассматривать сначала элемент s, потом s + t, затем s + 2*t и т.д. Естественно, чтобы не выйти за границы массива, мы обязаны смотреть на номер элемента по модулю (остатку от деления на размер массива).

Наконец мы объяснили все самые важные моменты, можно перейти к непосредственному написанию кода, где уже можно будет рассмотреть все оставшиеся нюансы.

Какая хэш-функция может считаться хорошей?

Так как input может быть любой строкой, а возвращаемое число находится в каком-то гарантированном интервале, может случиться так, что при двух разных входных строках будет возвращено одно и то же число. Это называется «коллизией»; хорошие хэш-функции стремятся к минимизации создаваемых ими коллизий.

Однако полностью избавиться от коллизий невозможно. Если бы мы написали хэш-функцию, возвращающую число в интервале от 0 до 7, и передали бы ей 9 уникальных входных значений, то гарантированно получили бы как минимум 1 коллизию.

Выходные значения хорошо известной хэш-функции modulo 8 (деление на 8 с остатком). Какие бы 9 значений мы ни передали, есть всего 8 уникальных чисел, поэтому коллизии неизбежны. Цель заключается в том, чтобы их было как можно меньше.

Подробнее, что под капотом

В программировании достаточно распространена задача хранения данных в ассоциативных контейнерах. В этом случае мы ставим в соответствие некоторым ключам некоторые значения. При этом мы бы хотели иметь возможность более-менее быстро запрашивать значение по ключу.

Простой массив из пар ключей и значений

Или можно завести два отдельных массива - один для всех ключей и один для всех значений. И договориться, что если некий ключ лежит по индексу i в первом массиве, то по тому же индексу во втором массиве лежит связанное с ним значение.

Добавление элемента в такой контейнер устроено очень легко - просто дописываем элемент в конец массива. Алгоритмическая сложность O(1). Если, конечно, у нас не кончится память, выделенная под массив и не придётся запрашивать у ОС новый кусок, но этот выходит за рамки данной статьи.

Зато с получением элемента по ключу всё очень плохо - в худшем случае (искомый элемент находится в конце массива) нужно обойти все элементы и

применить к каждому операцию сравнения. Алгоритмическая сложность O(N).

В общем, такая структура подходит только для ситуаций, когда мы добавляем (именно добавляем, с удалением всё тоже не очень хорошо - из-за необходимости сдвигать все элементы, которые были за удаляемым, получаем тоже сложность O(N)) новые элементы гораздо чаще, чем их ищем. Зато максимально простая и в осмыслении, и в реализации.

Технически можно перейти от массива к связанному списку, тогда вставка останется O(1), но ещё и удаление станет O(1), однако поиск всё равно останется O(N). К тому же связанный список плохо ложится на кеш процессора (ведь каждый элемент имеет случайный адрес в памяти, а в кеш попадают только соседние адреса), так что реальная скорость поиска будет даже ниже, чем у массива.

Отсортированный массив из пар ключей и значений

Можно при вставке нового элемента в массив добавлять его не в конец, а искать ему такое место, чтобы ключи в массиве после вставки сохранили неубывающий или невозрастающий порядок.

В таком случае мы сможем использовать бинарный поиск и для собственно поиска нужного ключа, и для его вставки. В итоге поиск будет иметь алгоритмическую сложность O(logN), что значительно лучше предыдущего варианта. Вставка будет иметь сложность в худшем случае O(N), потому что хоть место вставки мы и можем найти на O(logN), но в зависимости от того куда нам нужно вставить элемент, нам потребуется двигать все элементы, которые расположены после него. Удаление тоже будет иметь сложность O(N) по той же причине сдвига элементов.

Можно отойти от массива в сторону бинарного дерева, тогда вставка и удаление станут также как и поиск O(logN).

Ещё мы получаем побочный эффект, что мы в любой момент можем обойти все элементы контейнера в порядке возрастания/убывания ключей. Это может быть полезно (например, для вывода на экран в отсортированном виде).

Из минусов, собственно, что нам нужно иметь эту самую функцию сравнения "больше"/"меньше" для нашего типа ключей. И если для чисел и даже строк вопросов обычно не возникает, то при использовании более сложных ключей интуитивной функции сравнения может и не быть (ну, например, как сравнить две трёхмерные координаты?). Конечно, всегда можно придумать фиктивную функцию сравнения, но это уже не так удобно.

Ну и ещё алгоритмы работы с бинарными деревьями часто весьма нетривиальны для понимания и написания, если по какой-то причине мы не можем воспользоваться готовой библиотекой.

Хеш-таблица

Как было бы удобно, если бы все ключи имели однозначное соответствие числам, причём небольшим числам (относительно объёма памяти компьютера, которую мы морально готовы отдать под хранение наших данных). Тогда можно было бы просто использовать их как индексы в массиве.

Например, в задаче подсчёта количества повторений символов в строке мы можем просто завести массив из 256 элементов, зная, что код символа в ASCII не может превышать 255. И когда нам нужно узнать или увеличить счётчик повтора символов, просто использовать числовой код символа как индекс в этом массиве. Алгоритмическая сложность доступа к одному счётчику - O(1).

Однако, часто тип ключа имеет слишком много возможных значений. Например, даже обычный числовой ключ int имеет больше 4 миллиардов возможных значений. Строки вообще условно бесконечное количество значений (на практике ограничено размером ОЗУ компьютера). Нет никакой возможности заводить массивы такой размерности.

Однако, существует решение этой проблемы и этим решением является применение хеш-функции. Хеш-функция - это одностороннее преобразование с потерей информации, позволяющее свести большую область значений к меньшей. Простейшей хеш-функцией будет, например, взятие остатка от деления. Например, беря остаток от деления на 1000 мы сводим множество из

4 миллиардов значений int к 2000 значений (из-за отрицательных чисел, для unsigned int множество стало бы ещё меньше - всего 1000 значений).

Ну вот мы и придумали хеш-таблицы. Чтобы построить хеш-таблицу нужно совершить два действия: вычислить хеш от ключа, а затем найти его остаток от деления на длину массива, в котором мы решили хранить наши данные (исходя из доступных нам ресурсов и ожидаемого объёма данных). Этот результат и будет индексом, по которому нужно сохранять наш элемент или наоборот извлекать. Неиспользуемые ячейки массива можно пометить каким-нибудь образом (сохранить туда "невозможное" значение ключа, либо завести дополнительную карту занятости ячеек).

Однако, такое простое решение не могло бы не иметь подвоха. Поскольку мы сводим большее множество значений к меньшему, неизбежны коллизии. Например, для хеш-функции остатка от деления на 1000, числа 123, 1123, 2123, 1000123 дадут один и тот же результат. В итоге все эти значения должны будут быть сохранены в одной и той же ячейке нашего массива, но мы можем хранить только одно значение. Что же делать?

Существует два основных подхода к решению этой проблемы.

Во-первых, кто сказал, что одна ячейка массива не может хранить множество значений? Пусть у нас будет не массив значений, а массив связанных списков наших значений (или массив динамических массивов наших значений). Если случается коллизия, то просто добавляем новый элемент в список. При поиске ключа если в ячейке несколько значений, то ищем нужное обычным перебором (поэтому хеш-таблице важно, чтобы для ключей была определена не только хеш-функция, но и функция сравнения, но в отличии от подхода с отсортированным массивом, функции сравнения нужно выдавать только "равно"/"неравно" и её легко интуитивно определить для любого типа ключа). Таким образом теоретически имеем алгоритмическую сложность O(N), но на практике принято говорить о "средней" сложности O(1). Так как для худшего случая нам должно очень неповезти и для абсолютно всех элементов нашего контейнера должно

совпасть значение хеш-функции. На практике хеш-функции выбираются таким образом, чтобы коллизии были весьма редки (особенно на конкретном типе наборов данных, с которыми будет работать программа).

У этого подхода есть минус - как уже было сказано выше, связанные списки плохо дружат с кешами современных процессоров, потому что данные оказываются размазаны по всему адресному пространству вместо того, чтобы быть расположенными рядом. К тому же, каждое добавление элемента будет вызывать обращение к менеджеру памяти за новым кусочком памяти, что тоже весьма небыстрое дело. Конечно, можно использовать вместо связанного списка динамический массив, но это не сильно спасает ситуацию. Если под саму хеш-таблицу мы можем заранее выделить память по принципу "мы ожидаем получить N элементов, значит создадим массив на (N + некая констата) элементов", то поступить так с внутренними массивами элементов мы не можем, иначе наша программа будет потреблять N^2 памяти, что редко бывает допустимо. Угадать какой именно ячейке хеш-таблицы "не повезёт" заранее и выделить много памяти только ей, мы тоже едва ли можем.

К счастью, есть ещё одно решение. Допустим, мы вычислили хеш и обнаружили, что в соответствующей ему ячейке хеш-таблицы уже лежит значение. Не беда! Просто посмотрим в следующую за ней ячейку (значение хеша + 1). Если и там занято, смотрим следующую за ней и т. д. При достижении последнего индекса массива, переходим к нулевому. Если хеш-таблица имеет размер N, то по такому алгоритму мы гарантированно сможем положить в неё не меньше N элементов (для N + 1 элемента мы не найдём свободного места перебрав все элементы и единственным выходом будет увеличить её размер). Алгоритм поиска аналогичен - вычисляем хеш элемента, а затем сравниваем ключ с тем, что реально лежит по этому индексу. Если там лежит что-то не то, то смотрим ключ следующего элемента. Повторяем, пока не найдём нужный элемент, пока не сделаем полный круг, либо пока не встретим пустое место. По нашему алгоритму вставки, мы не могли

пропустить пустое место, если бы пытались вставить нужный нам ключ, так что обнаружение пустого места гарантирует, что этого элемента в таблице нет.

Несложно догадаться, что при полной заполненности хеш-таблицы и при поиске отсутствующего элемента, мы гарантированно обойдем все элементы массива, получая сложность O(N). Звучит не круто, поэтому держать такие хеш-таблицы полными не принято, а принято говорить о "коэффициенте заполнения" при превышении которого хеш-таблица увеличивается в размере несмотря на то, что технически в неё ещё можно было вставить элемент. Считается хорошим значением коэффициента заполнения число близкое к 75%. Это значит, что хеш-таблица на 100 элементов, в которую реально вставлено 75 элементов, при вставе 76-го обязана вырасти в размере несмотря на наличие свободных мест. Такой подход создаёт некоторый оверхед по памяти, зато даёт значительный выигрыш в скорости. К тому же чем больше свободных элементов, тем ниже вероятность коллизии хеш-функции в целом.

Итак, мы имеем худшую сложность вставки O(N), среднюю O(1). При этом наша структура данных имеет отличную локальность (все элементы хранятся рядом), что хорошо для работы процессора с кешем. Если известно максимальное число элементов, можно заранее создать хеш-таблицу нужного размера и гарантированно избежать перевыделения памяти. Идеально же?

Конечно, не бывает ничего идеального. Можно легко догадаться, что каждая коллизия в такой таблице повышает вероятность следующей коллизии - ведь элемент, у которого случилась коллизия, займёт место, куда должны был встать другой элемент. И этот элемент тоже будет вынужден занять не своё место и т. д. Для многих наборов данных и хеш-функций это не является большой проблемой, однако существует модификация алгоритма выбора места в среднем улучшающая ситуацию - квадратичный поиск (quadratic probing). Идея в том, что мы после первого обнаруженного занятого места, смотрим в следующее. Если и оно занято, то смотрим не в следующее, а через одно. Если и там занято, то через два и т. д. То есть каждая неудачная попытка занять место увеличивает на единицу шаг, с которым происходит выбор

следующего места. На практике согласно исследованиям, это часто даёт лучшие результаты.

Удаление элементов

Можно заметить подвох. При удалении элемента из хеш-таблицы нам придётся пометить его место как свободное. Однако, обнаружение свободного места в том числе является условием выхода из цикла поиска элемента по ключу "элемент с таким ключом не найден". Таким образом удаление элемента разорвёт цепочку перехода к следующей ячейке, которая, возможно, привела бы нас к искомому элементу.

Если нам не нужно удалять элементы из хеш-таблицы, то можно ничего не делать, однако и для реализации хеш-таблицы общего назначения придумано решение.

Мы можем вместо двух состояний ячейки "свободна" и "занята" ввести три состояния - "свободна", "занята", "удалена". Последнее состояние будет обрабатываться по-разному в алгоритме поиска и алгоритме вставки - алгоритм поиска будет пропускать эту ячейку как занятую не тем значением (хранящийся ключ лучше не проверять в общем случае, вдруг он, например, ссылается на динамически выделенную память, которую, конечно же освободил пользователь при удалении ключа), а вот алгоритм вставки будет считать эту ячейку свободной и вставлять в неё новое значение (при этом, разумеется, ячейка будет менять статус на "занята").

Ещё одна оптимизация

Если принять, что размер хеш-таблицы обязан быть степенью 2, то мы сможем очень эффективно вычислять положение элемента по его хешу с помощью битовых операций, которые работают гораздо быстрее вычисления остатка от деления.

3. ВВЕДЕНИЕ В СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ. ЧТО ТАКОЕ B-ДЕРЕВЬЯ И КАК ОНИ РАБОТАЮТ

В деревьях поиска, таких как двоичное дерево поиска, AVL дерево, красно-чёрное дерево и т.п. каждый узел содержит только одно значение (ключ) и максимум двое потомков. Однако есть особый тип дерева поиска, который называется B-дерево (произносится как Би-дерево). В нем узел содержит более одного значения (ключа) и более двух потомков. B-дерево было разработано в 1972 году Байером и МакКрейтом и называлось *Сбалансированное по высоте дерево поиска порядка m (Height Balanced m-way Search Tree)*. Свое современное название B-дерево получило позже.

B-дерево можно определить следующим образом:

B-дерево – это сбалансированное дерево поиска, в котором каждый узел содержит множество ключей и имеет более двух потомков.

Здесь количество ключей в узле и количество его потомков зависит от порядка B-дерева. Каждое B-дерево имеет порядок.

B-дерево порядка **m** обладает следующими свойствами:

- Свойство 1: Глубина всех листьев одинакова.
- Свойство 2: Все узлы, кроме корня должны иметь как минимум (m/2) – 1 ключей и максимум m-1 ключей.
- Свойство 3: Все узлы без листьев, кроме корня (т.е. все внутренние узлы), должны иметь минимум m/2 потомков.
- Свойство 4: Если корень – это узел, не содержащий листьев, он должен иметь минимум 2 потомка.
- Свойство 5: Узел без листьев с n-1 ключами должен иметь n потомков.
- Свойство 6: Все ключи в узле должны располагаться в порядке возрастания их значений.

Например, B-дерево 4 порядка содержит максимум 3 значения ключа и максимум 4 потомка для каждого узла.

Операции над B-деревом

Над B-деревом можно проводить следующие операции:

- Поиск
- Вставка
- Удаление

Поиск по B-дереву

Поиск по B-дереву аналогичен поиску по двоичному дереву поиска. В двоичном дереве поиска поиск начинается с корня и каждый раз принимается двустороннее решение (пойти по левому поддереву или по правому). В B-дереве поиск также начинается с корневого узла, но на каждом шаге принимается n-стороннее решение, где n – это общее количество потомков рассматриваемого узла. В B-дереве сложность поиска составляет O(log n). Поиск происходит следующим образом:

Шаг 1: Считать элемент для поиска.

Шаг 2: Сравнить искомый элемент с первым значением ключа в корневом узле дерева.

Шаг 3: Если они совпадают, вывести: «Искомый узел найден!» и завершить поиск.

Шаг 4: Если они не совпадают, проверить больше или меньше значение элемента, чем текущее значение ключа.

Шаг 5: Если искомый элемент меньше, продолжить поиск по левому поддереву.

Шаг 6: Если искомый элемент больше, сравнить элемент со следующим значением ключа в узле и повторять Шаги 3, 4, 5 и 6 пока не будет найдено совпадение или пока искомый элемент не будет сравнен с последним значением ключа в узле-листе.

Шаг 7: Если последнее значение ключа в узле-листе не совпало с искомым, вывести «Элемент не найден!» и завершить поиск.

Операция вставки в B-дерево

В B-дереве новый элемент может быть добавлен только в узел-лист. Это значит, что новая пара ключ-значение всегда добавляется только к узлу-листу. Вставка происходит следующим образом:

Шаг 1: Проверить пустое ли дерево.

Шаг 2: Если дерево пустое, создать новый узел с новым значением ключа и его принять за корневой узел.

Шаг 3: Если дерево не пустое, найти подходящий узел-лист, к которому будет добавлено новое значение, используя логику дерева двоичного поиска.

Шаг 4: Если в текущем узле-листе есть незанятая ячейка, добавить новый ключ-значение к текущему узлу-листу, следуя возрастающему порядку значений ключей внутри узла.

Шаг 5: Если текущий узел полон и не имеет свободных ячеек, разделите узел-лист, отправив среднее значение родительскому узлу. Повторяйте шаг, пока отправляемое значение не будет зафиксировано в узле.

Шаг 6: Если разделение происходит с корнем дерева, тогда среднее значение становится новым корнем дерева и высота дерева увеличивается на единицу.

Пример:

Давайте создадим B-дерево порядка 3, добавляя в него числа от 1 до 10.

Insert(1):

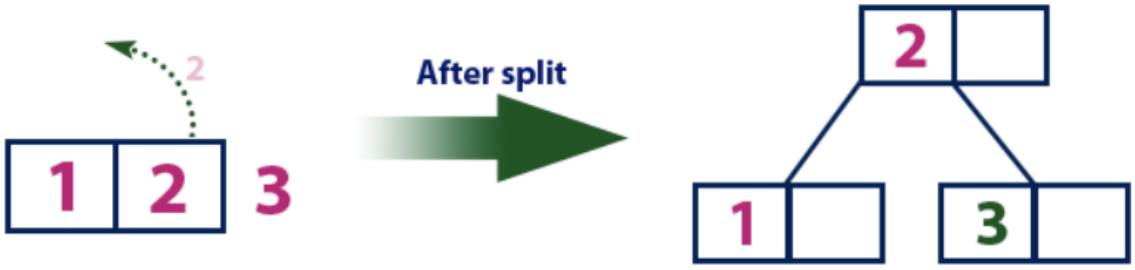
Поскольку «1» — это первый элемент дерева – он вставляется в новый узел и этот узел становится корнем дерева.

Insert(2):

Элемент «2» добавляется к существующему узлу-листу. Сейчас у нас всего один узел, следовательно, он является и корнем, и листом одновременно. В этом листе имеется пустая ячейка. Тогда «2» встает в эту пустую ячейку.

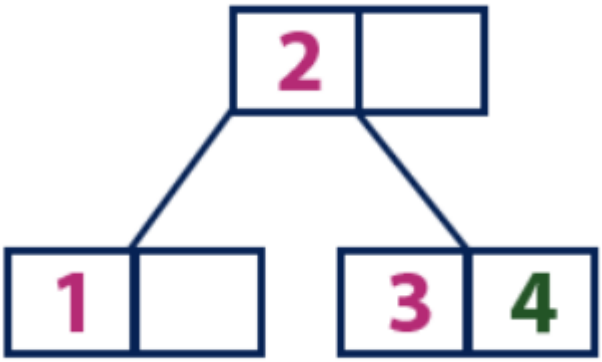
Insert(3):

Элемент «3» добавляется к существующему узлу-листу. Сейчас у нас только один узел, который одновременно является и корнем, и листом. У этого листа нет пустой ячейки. Поэтому мы разделяем этот узел, отправляя среднее значение (2) в родительский узел. Однако у текущего узла родительского узла нет. Поэтому среднее значение становится корневым узлом дерева.



Insert(4):

Элемент «4» больше корневого узла со значением «2», при этом корневой узел не является листом. Поэтому мы двигаемся по правому поддереву от «2». Мы приходим к узлу-листу со значением «3», у которого имеется пустая ячейка. Таким образом, мы можем вставить элемент «4» в эту пустую ячейку.



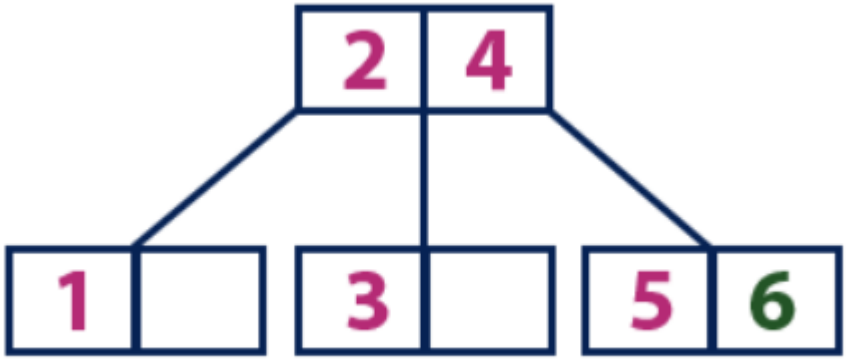
Insert(5):

Элемент «5» больше корневого узла со значением «2», при этом корневой узел не является листом. Поэтому мы двигаемся по правому поддереву от «2». Мы приходим к узлу-листу и обнаруживаем, что он уже полон и не имеет пустых ячеек. Тогда мы делим этот узел, отправляя среднее

значение (4) в родительский узел (2). В родительском узле есть для него пустая ячейка, поэтому значение «4» добавляется к узлу, в котором уже есть значение «2», а новый элемент «5» добавляется в качестве нового листа.

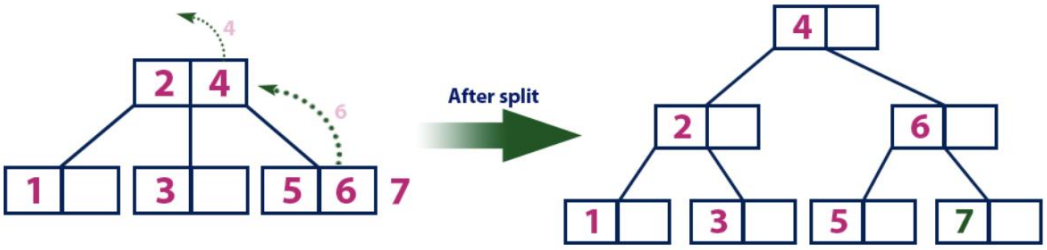
Insert(6):

Элемент «6» больше, чем элементы корня «2» и «4», который не является листом. Мы двигаемся по правому поддереву от элемента «4». Мы достигаем листа со значением «5», у которого есть пустая ячейка, поэтому элемент «6» помещаем как раз в нее.



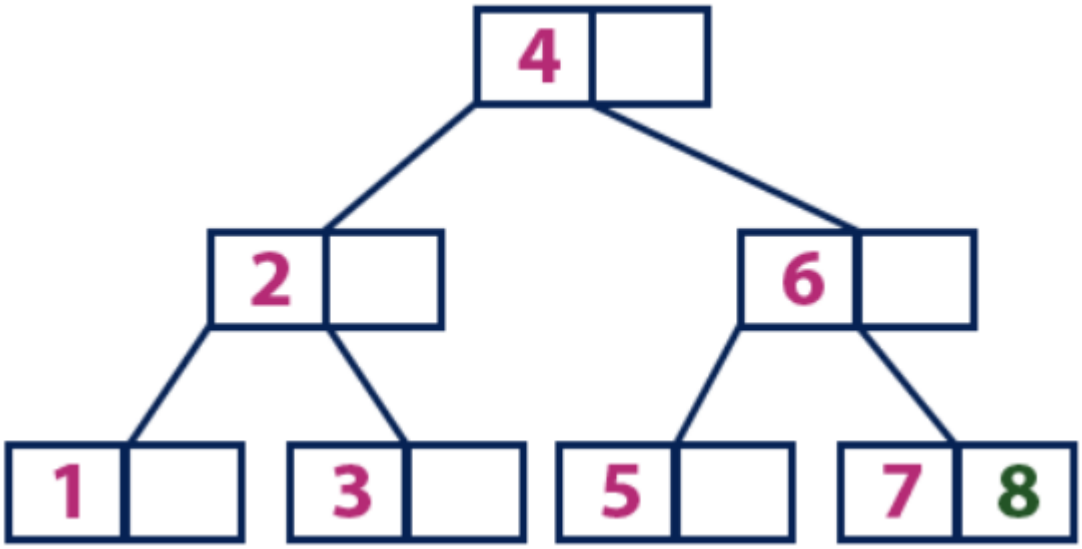
Insert(7):

Элемент «7» больше, чем элементы корня «2» и «4», который не является листом. Мы двигаемся по правому поддереву от элемента «4». Мы достигаем узла-листа и видим, что он полон. Мы делим этот узел, отправляя среднее значение «6» вверх к родительскому узлу с элементами «2» и «4». Однако родительский узел тоже полон, поэтому мы делим узел с элементами «2» и «4», отправляя значение «4» родительскому узлу. Только вот этого узла еще нет. В таком случае узел с элементом «4» становится новым корнем дерева.



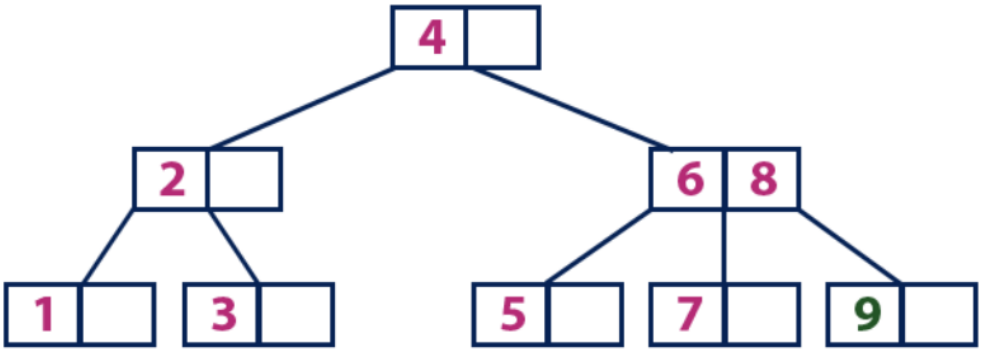
Insert(8):

Элемент «8» больше корневого узла со значением «4», при этом корневой узел не является листом. Мы двигаемся по правому поддереву от элемента «4» и приходим к узлу со значением «6». «8» больше «6» и узел с элементом «6» не является листом, поэтому двигаемся по правому поддереву от «6». Мы достигаем узла-листа с «7», у которого есть пустая ячейка, поэтому в нее мы помещаем «8».



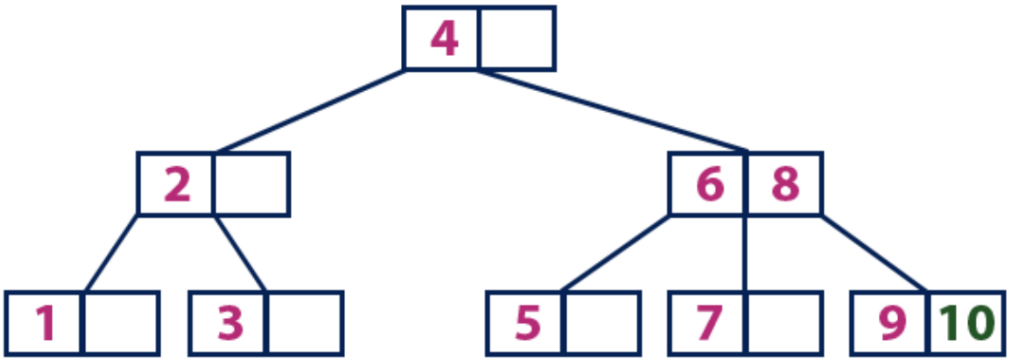
Insert(9):

Элемент «9» больше корневого узла со значением «4», при этом корневой узел не является листом. Мы двигаемся по правому поддереву от элемента «4» и приходим к узлу со значением «6». «9» больше «6» и узел с элементом «6» не является листом, поэтому двигаемся по правому поддереву от «6». Мы достигаем узла-листа со значениями «7» и «8». Он полон. Мы делим этот узел, отправляя среднее значение (8) родительскому узлу. Родительский узел «6» имеет пустую ячейку, поэтому мы помещаем «8» в нее. При этом новый элемент «9» добавляется в узел-лист.



Insert(10):

Элемент «10» больше корневого узла со значением «4», при этом корневой узел не является листом. Мы двигаемся по правому поддереву от элемента «4» и приходим к узлу со значениями «6» и «8». «10» больше «6» и «8» и узел с этими элементами не является листом, поэтому двигаемся по правому поддереву от «8». Мы достигаем узла-листа со значением «9». У него есть пустая ячейка, поэтому туда мы помещаем «10».



4. ВВЕДЕНИЕ В LSM ДЕРЕВЬЯ. ЧТО ТАКОЕ LSM-ДЕРЕВЬЯ И КАК ОНИ РАБОТАЮТ

С увеличением спроса на операции, которые требуют больших объемов записи, традиционные базы данных, использующие **B-дерево**, становятся узким местом, поскольку обновление записей в b-дереве приводит к многочисленным беспорядочным операциям ввода-вывода (IO) и обновлению нескольких страниц на диске. B-дерево очень хорошо подходит для "тяжелых" операций чтения. Для операций с большими объемами записи у нас есть LSM-дерево.

LSM-деревья (Log-Structured Merge) — это тип структуры данных, используемый для эффективного хранения и извлечения больших объемов данных в динамичной и изменяющейся среде.

Необходимость в такой структуре данных возникла из-за ограничений традиционных систем хранения данных, таких как B-деревья, при обработке рабочих нагрузок с высокой интенсивностью записи.

Основная идея LSM-деревьев заключается в объединении преимуществ файловых систем с лог-структурой и B-деревьев.

LSM-деревья работают путем организации данных в серию более мелких, легко управляемых файлов, называемых SSTables (Sorted String Tables).

Что такое SSTables?

SSTables (Sorted String Tables) — строительные блоки LSM-деревьев (Log-Structured Merge). Это дисковые структуры данных, которые хранят информацию в отсортированном формате, что позволяет эффективно искать и извлекать ее оттуда.

SSTables — это результат процесса уплотнения в LSM-деревьях, когда мелкие, несортированные файлы данных объединяются вместе, образуя более крупные, лучше организованные SSTables.

Данные в SST-таблице хранятся в отсортированном виде, что позволяет быстро искать конкретную информацию с помощью алгоритмов двоичного поиска.

Пояснение на примере

SSTable состоит из отсортированной, иммутабельной последовательности пар ключ-значение, где каждый ключ уникален и соответствует одному значению.

Предположим, у нас есть база данных записей о сотрудниках, где каждая запись имеет уникальный идентификатор сотрудника и содержит такую информацию, как имя, возраст, зарплата и отдел. Мы можем представить эти данные в виде SSTable со следующей структурой

Employee ID	Name	Age	Salary	Department
1001	John Doe	30	$50,000	HR
1002	Jane Doe	25	$40,000	IT
1003	Bob Smith	35	$60,000	Sales

В этом примере SSTable содержит три пары ключ-значение, по одной для каждой записи о сотруднике.

Ключи — это идентификаторы сотрудников (ID), а значения — остальные поля записи. Ключи отсортированы в порядке возрастания, чтобы обеспечить эффективный двоичный поиск и запросы по диапазону.

Когда выполняется операция чтения, к SSTable обращаются для получения значения, связанного с данным ключом (ID сотрудника). Например, если мы хотим получить запись для идентификатора сотрудника 1002, SSTable будет просмотрена с использованием индекса, чтобы найти соответствующий блок. Затем будет возвращено значение, связанное с ключом 1002.

SSTable обычно содержит следующую информацию:

1. Пары ключ-значение: Основные данные, хранящиеся в SSTable. Каждая пара ключ-значение представляет собой одну запись в базе данных.
2. Фильтр Блума: Структура данных, используемая для быстрого определения наличия или отсутствия ключа в SSTable.
3. Индекс: Структура данных, используемая для сопоставления ключей с соответствующим блоком в SSTable. Это позволяет эффективно выполнять поиск по ключу.
4. Сжатие: SSTables обычно подвергаются сжатию с помощью таких алгоритмов, как Snappy или LZ4, чтобы уменьшить использование дискового пространства.

Фильтр Блума

Фильтр Блума — это структура данных, которая используется для определения того, является ли элемент членом множества или нет. В контексте SSTable фильтр Блума используется для определения того, присутствует ли данный ключ в SSTable или нет.

Фильтр Блума обеспечивает быстрый и эффективный способ проверки наличия ключа без необходимости выполнять полное сканирование диска.

Пояснение на примере

Предположим, у нас есть SSTable с большим количеством ключей, и мы хотим проверить, присутствует ли ключ k в SSTable. Вместо того чтобы сканировать всю SSTable, можно воспользоваться фильтром Блума, чтобы быстро определить, присутствует ключ или нет.

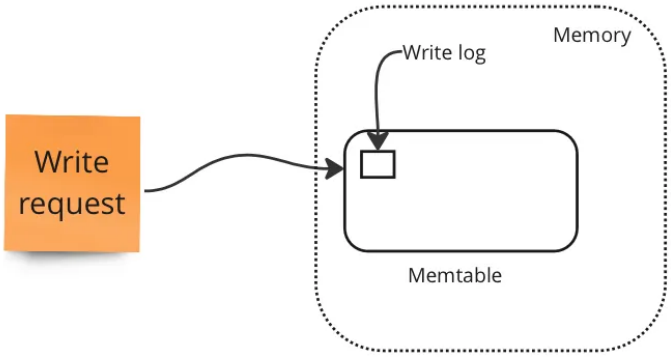
Если вероятность присутствия ключа невелика (т.е. фильтр Блума возвращает false), то поиск можно прекратить, сэкономив время и дисковый ввод-вывод. Если ключ, скорее всего, присутствует (т.е. фильтр Блума возвращает true), то можно выполнить поиск в SSTable для получения значения, связанного с ключом.

LSM-деревья считаются динамическими системами хранения данных, поскольку они могут обрабатывать большие объемы информации, которые постоянно изменяются с течением времени.

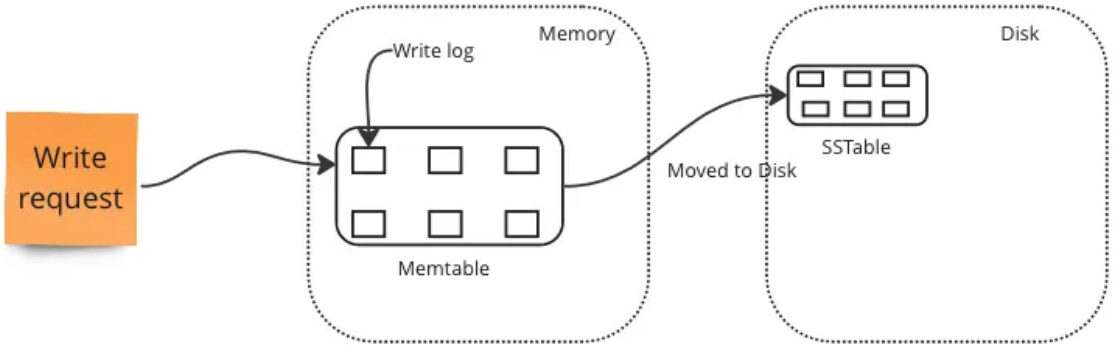
Как работает LSM-дерево?

LSM-деревья работают путем организации данных в серию небольших, более удобных для управления файлов, называемых SSTables (Sorted String Tables), которые могут быть легко объединены вместе для формирования более крупного индекса. Такой подход обеспечивает быстрое и эффективное хранение данных, а также быстрый доступ к ним.

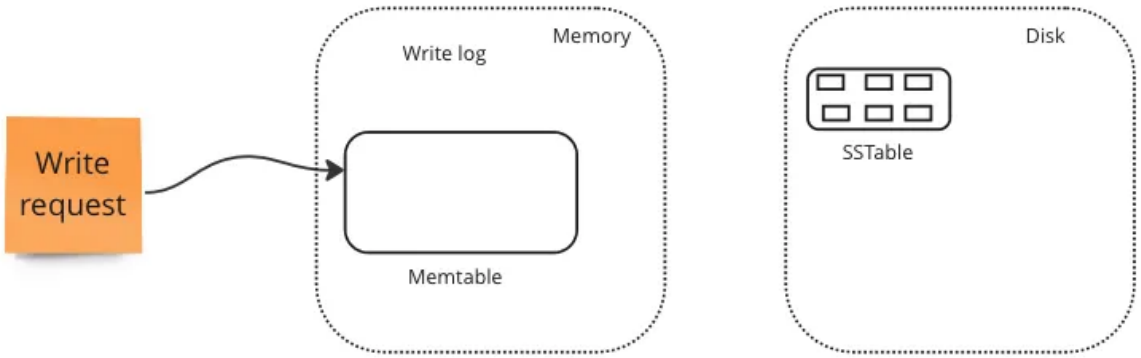
1. Сначала данные добавляются в оперативную память (in-memory) Memtable.



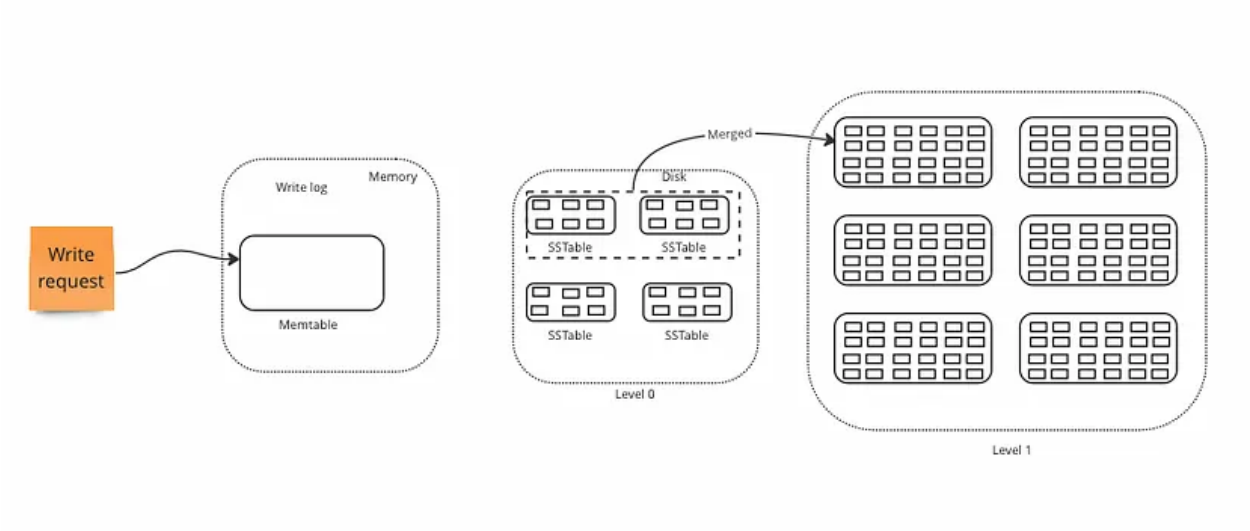
2. Когда memtable достигает определенного размера, она сбрасывается на диск и становится новой SSTable.



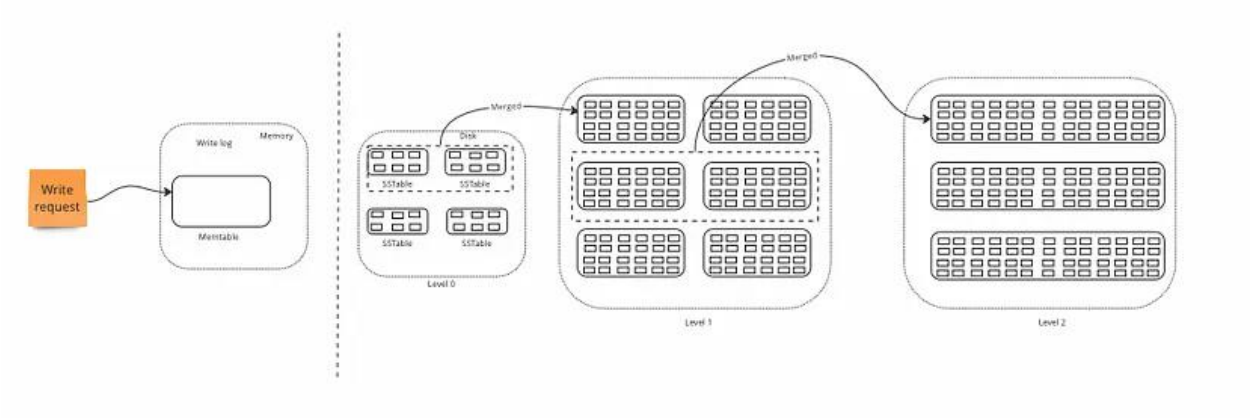
3. Затем SSTable добавляется на самый нижний уровень LSM-дерева.



4. По мере записи информации в базу данных создается множество SSTables, которые добавляются на самый нижний уровень LSM-дерева.



5. Когда самый нижний уровень LSM-дерева становится слишком большим, SSTables внутри него объединяются вместе, образуя более крупные SSTables, и перемещаются на следующий уровень дерева.



Объединение (мерджинг) SSTables называется **уплотнением**.

Уплотнение

Уплотнение является важным аспектом LSM-деревьев и играет решающую роль в производительности и эффективности системы хранения данных.

Уплотнение — это процесс объединения нескольких небольших SST-таблиц в более крупные, что уменьшает количество SST-таблиц и объем дискового пространства, используемого данными.

Это, в свою очередь, уменьшает амплификацию чтения, то есть количество операций ввода-вывода на диск, необходимых для получения одной записи из SSTables.

Уплотнение помогает сохранить управляемость всех SSTables, что особенно важно для нагрузок, связанных с записью, где количество таблиц SSTables может быстро расти.

Уплотнение также помогает поддерживать эффективность и производительность LSM-дерева с течением времени. По мере записи новых данных в систему количество SSTables может увеличиваться, что может замедлить операции чтения и привести к повышению дискового ввода-вывода.

Потеря данных при выключении питания?

Если вы заметили, 1st data [данные, которые компании получают от клиентов напрямую через регистрации, пиксели и cookies на сайте, CRM и другие сервисы] записываются в memTable, которая находится в памяти, а как мы знаем, память волатильна, и данные в ней теряются, если отключается питание или что-то идет не так. Значит, в этом случае наши данные будут потеряны?

Ответ — нет!

В LSM-дереве memtable используется как временная структура хранения для новой информации, которая записывается в базу данных. При отключении питания все данные, которые еще не были выгружены на диск и скомпилированы в SSTable, будут потеряны.

Чтобы предотвратить потерю данных в случае отключения питания, в LSM-деревьях часто реализуется механизм логирования с опережением записи, при котором изменения данных сначала записываются в журнал на диске, а затем фиксируются в memtable.

Таким образом, в случае отключения питания информация из лога может быть использована для восстановления любых данных, которые были утеряны из memtable.

Преимущества LSM-деревьев

Существует несколько ключевых преимуществ использования LSM-деревьев, в том числе:

1. Высокая пропускная способность записи: LSM-деревья разработаны для обработки рабочих нагрузок с высокой интенсивностью записи, что делает их хорошим выбором для сред с большим объемом входящего трафика записи.

2. Быстрый доступ к данным: LSM-деревья оптимизированы для быстрого доступа к данным, что позволяет оперативно извлекать информацию из больших массивов.

3. Эффективное хранение: Разбивая данные на более мелкие SST-таблицы и объединяя их по мере увеличения размера, LSM-деревья способны эффективно хранить большие объемы информации на диске.

4. Динамичная и изменяющаяся среда: LSM-деревья разработаны для работы с динамичными и изменяющимися данными, что делает их хорошим выбором для сред, где информация постоянно добавляется, обновляется и удаляется.

Недостатки LSM-деревьев

Хотя LSM-деревья имеют много преимуществ, есть и некоторые недостатки, которые следует учитывать, в том числе:

Более интенсивное использование диска: Процесс уплотнения, используемый в LSM-деревьях, может привести к повышенному использованию диска, поскольку на нем может храниться несколько копий одних и тех же данных.

1. Сложность: LSM-деревья могут быть сложными в реализации и обслуживании, и для их правильной настройки и управления может потребоваться высокий уровень квалификации.

2. Повышенное использование памяти: LSM-деревья могут быть требовательны к памяти. Для хранения данных в памяти может потребоваться ее значительный объем.

5. ВВЕДЕНИЕ В ХРАНЕНИЕ И ИЗВЛЕЧЕНИЕ ДАННЫХ

Что такое база данных и зачем она?

Компании часто собирают информацию о своих клиентах, сотрудниках, операциях, финансах и т. д. Потом эту информацию можно выгодно использовать. Например, можно ее проанализировать и понять, какими способами можно увеличить прибыль. Можно на ее основе построить хитрые ML модели, которые помогут улучшить продукт. Или, в конце концов, эту информацию можно просто перепродать другим компаниям.

Чтоб собирать и анализировать информацию, надо уметь ее сохранять. Конечно, можно сохранять информацию в печатном виде в обычных папках или в Excel-файлах. И многие компании до сих пор так сохраняют информацию. Однако, такое подойдет только для маленьких компаний с небольшим количеством данных. Когда компания вырастает, то и данных становится много, такие варианты сохранения информации становятся непригодны. Тогда на помощь приходят базы данных.

Базы данных помогают справиться с большим количеством проблем, решить которые папкам и Excel-файлам не под силу:

- В базе данных можно хранить очень огромное количество данных – миллиарды и триллионы записей;
- Базы помогают защищать данные - они позволяют давать доступ к данным только определенному кругу лиц. При этом можно ставить ограничения, кому к каким данным можно давать доступ и какого типа доступ, только чтение или редактирование тоже;
- Базы данных могут помогать следить за правильностью данных с помощью различного вида проверок;
- Также, базы данных могут позволять большому количеству людей одновременно взаимодействовать с данными.

Так что же такое база данных? Если говорить коротко, то это определенная структура, в которой хранится информация. Я понимаю, что из

этого определения пока мало что понятно. Однако, более конкретное определение дать сложно, потому что существует много типов баз данных, и все они совершенно разные.

Я думаю, это определение станет понятнее, когда я далее опишу наиболее популярные типы баз данных на конкретных примерах.

Типы баз данных

Существует много разных типов баз данных. Наиболее популярные типы:

- Реляционные базы данных
- Key-value базы данных
- Документно-ориентированные базы данных
- Графовые базы данных
- Колоночные базы данных

Реляционные базы данных (MySQL, PostgreSQL, Oracle DB)

Реляционная база данных – это база данных, которая состоит из таблиц. У реляционной базы данных 2 очень важные характеристики:

- Данные распределены по смыслу по таблицам
- Между таблицами есть отношения

Рассмотрим пример реляционной базы. Допустим, у нас есть сервис доставки еды. Тогда, если мы построим реляционную базу данных для этого сервиса, то она, скорее всего, будет содержать следующие таблицы:

- Таблица с заказами
- Таблица с клиентами
- Таблица с курьерами
- Таблица с ресторанами

Как я отметила выше, второй важной характеристикой реляционных баз данных является то, что между таблицами существуют отношения. Отношения между таблицами определяются с помощью primary key и foreign key.

Primary key – это столбец (или группа столбцов) таблицы, который содержит уникальные значения для каждой строки. На примере выше primary key каждой таблицы я выделила зеленым цветом. То есть, например, в таблице с заказами каждая строка будет описывать отдельный заказ. Не будет 2 строк, которые описывают один и тот же заказ, потому ID заказа будет разный для каждой строки.

Foreign key – это столбец в таблице, который содержит primary key другой таблицы. На рисунке foreign key отмечены желтым. То есть, таблица с заказами содержит ID клиента, который является primary key в таблице с клиентами, но в таблице с заказами он будет foreign key.

Primary key и foreign key помогают не только связывать между собой таблицы реляционной базы данных отношениями. Они еще помогают следить за целостностью и правильностью данных в базе. Например, если мы ошибемся в ID клиента, добавляя новый заказ в таблицу с заказами, то база выдаст ошибку, так как не найдет соответствующий ID клиента в таблице с клиентами.

Для взаимодействия с реляционными базами данных чаще всего используется SQL (Structured QueryLanguage). Это специальный язык программирования, на котором пишутся запросы к реляционной базе. SQL-запросами можно создавать и удалять таблицы в реляционной базе, изменять данные в существующих таблицах и доставать из таблиц необходимую информацию.

Как я уже говорила выше, реляционные базы данных удобно использовать в аналитике, так как информация в них структурирована и распределена по смыслу, что, конечно, мечта любого аналитика. Однако, аналитики часто пишут сложные и не очень эффективные SQL-запросы, потому важно придумывать способы ускорения обработки запросов к реляционной базе.

Одним из наиболее популярных методов ускорения работы запросов к реляционным базам данных является индексирование таблиц. Индекс – это определенный столбец в таблице, по которому осуществляется поиск.

Приведу пример работы индекса. Например, мы хотим найти все заказы клиента 007 из ресторана 1. Тогда, если у нас в таблице с заказами нет индекса, то мы будем перебирать все заказы пока не найдем нужные. Если же у нас есть индекс в таблице с заказами, то ситуация будет иной. Допустим, что индексом является столбец ID ресторана. Тогда наши данные в таблице с заказами будут сгруппированы по ID ресторана. И тогда при поиске заказов клиента 007 из ресторана 1, мы не будем перебирать всю таблицу с заказами, а найдем группу заказов из ресторана 1 и будем искать необходимые данные внутри этой группы.

Из примера выше с индексом выше понятно, что индексом удобно выбирать такой столбец, в разрезе которого часто ищутся данные.

Также, одним из важных свойств реляционных баз данных является соответствие требованиям ACID. Я не буду углубляться в детали этих требований, только отмечу, что эти требования гарантируют целостность и корректность данных, несмотря на ошибки, системные сбои, перебои в питании, изменение данных несколькими пользователями одновременно и прочие необычные ситуации.

Выглядит так, что реляционная база данных идеальная база, и непонятно, почему бы постоянно ее не использовать. Однако, у реляционной базы данных есть и недостатки, и потому данный тип не всегда подходит для нужд бизнеса. Например, реляционная база данных не подходит для данных без четкой структуры, потому что мы не сможем разложить эти данные в отдельные таблицы по смыслу. А данных без четкой структуры гораздо больше, чем данных с четкой структурой.

Какие еще есть типы баз данных?

Прочие типы баз данных, которые не реляционные, часто называются noSQL базы данных. Обсудим наиболее популярные типы нереляционных баз данных.

Key-value базы данных (пример - Redis)

Название говорит о том, какие данные удобно хранить в Key-value базе – в такой базе хранят данные, которые удобно представить в виде пары ключ-значение. Основное преимущество таких баз – это очень быстрый поиск значения по ключу. При этом значение может содержать какие угодно типы данных.

Такие базы данных удобно применять в проектах, где необходимо выдавать быстрый результат по ключу, например, для онлайн торгов или сделок.

Документно-ориентированные (пример - Mongo DB)

В документно-ориентированной базе данных единицей хранения является документ (который может быть в формате json, или xml, или в каком-нибудь еще формате). Удобство таких баз в том, что в них быстро и легко записывать любые типы данных, при этом эти данные не обязаны обладать четкой структурой. Минус таких баз в том, что данные в них неудобно анализировать.

В моей предыдущей компании такой тип баз данных служил базой для реляционных баз. То есть сначала все данные попадали и сохранялись в документно-ориентированной базе. Потом команда дата инженеров обрабатывала эти огромные полотна информации, структурировала и складывала в реляционную базу данных, которую уже могла использовать команда аналитиков и Data Science.

Графовые базы данных (пример - Orient DB)

Как следует из названия, в графовой базе данных данные хранятся в виде графов. Данный тип баз удобен, когда надо находить информацию не только о каком-то объекте, но и доставать информации о связях этого объекта с другими.

Например, в моей текущей компании данный тип баз используется для нахождения куки конкретного юзера и всех взаимосвязанных с этой кукой идентификаторов. Также, такой тип данных часто используется социальными сетями для сохранения информации не только о пользователях, но и о связях каждого пользователя с другими.

Колоночные (столбцовые) базы данных (примеры -Cassandra, Clickhouse)

В реляционных базах данных данные записаны в виде строк. Что же касается колоночных баз данных, то тут данные записываются в виде столбцов. Потому поиск данных в колоночной базе данных осуществляется не перебором всех строк, как это происходит в реляционной базе данных, а поиском необходимого значения в тех столбцах таблицы, которые нас интересуют.

Преимущество колоночных баз данных в том, что они могут быстро находить определенные значения в столбцах, которые нас интересуют.

Переход на in-memory

Современные тенденции финансового рынка предъявляют гораздо более строгие требования по времени отклика и работе средств автоматизации процессов в целом. К тому же, практически все крупнейшие финансовые институты стремятся сегодня к построению собственных экосистем.

В связи с этим мы видим для себя два основных применения in-memory решений. Во-первых, это кэширование интеграционных данных. По классическому сценарию в крупных компаниях существует нескольких автоматизированных систем, которые обеспечивают предоставление данных по запросу пользователя. Либо внешней системы — но в этом случае инициатором в большинстве случаев является пользователь. Традиционно эти системы хранили структурированные определенным образом данные в БД, осуществляя доступ к ней по запросу.

Сегодня такие системы уже не удовлетворяют требованиям в части нагрузки. Здесь не стоит забывать и про удаленные вызовы указанных систем

системами-потребителями. Отсюда вытекает необходимость пересмотра подходов к хранению и представлению данных — пользователям, автоматизированным системам или отдельным сервисам. Логичный выход — хранение актуальных данных, используемых сервисами, на уровне слоя in-memory; на рынке есть немало подобных успешных кейсов.

Это был первый кейс. Второй — это эффективное, с технической точки зрения, управление бизнес-процессами. Традиционные BPM-системы автоматизируют выполнение тех или иных операций в соответствии с заранее определенным алгоритмом. И во множестве случаев возникают вопросы: почему же эти системы работают недостаточно эффективно и недостаточно быстро?

Как правило, подобные системы пишут каждый свой шаг (или небольшой набор шагов, оформленный в виде бизнес-транзакции) в базу данных. Так что они завязаны на время отклика и взаимодействия с данными системами. Сейчас количество экземпляров бизнес-процессов, выполняющихся одновременно в режиме реального времени, на порядки больше чем 10 лет назад. Так что современные системы управления бизнес-процессами должны иметь существенно более высокую производительность и обеспечивать исполнение децентрализованных приложений. Тем более что сегодня все компании движутся к формированию большого микросервисного окружения. Задача в том, чтобы различные экземпляры бизнес-процессов могли разделять и эффективно использовать оперативные данные. В рамках оркестровки имеет смысл хранить их в in-memory решении.

Проблема согласования

Предположим, что у нас огромное количество узлов и сервисов, что выполняется ряд бизнес-процессов, действия которых реализованы в виде микросервисов. Чтобы повысить производительность, каждый из них начинает писать свое состояние в локальный экземпляр памяти. Мы получаем большое количество локальных экземпляров. Как обеспечить актуальность и согласованность для всех?

Мы используем зонирование in-memory области. Например, в зависимости от бизнес-домена. Когда мы нарезаем бизнес-домен, то определяем, чтобы те или иные микросервисы/бизнес-процессы работали только в рамках той зоны, которая отвечает за соответствующий домен. Так мы можем ускорить обновление кэша и всю работу in-memory решения.

При этом кэш, отвечающий за домен, работает в режиме полной репликации – ограниченное ввиду распределения по доменам количество узлов обеспечивает скорость и и корректность работы решения в данном режиме. Зонирование и максимальное дробление помогает решить проблемы синхронизации, работы кластера и т.д. на большом общем количестве узлов.

Часто возникают естественные вопросы о надежности in-memory решений. Да, туда можно класть не все. У нас с целью обеспечения надежности рядом с in-memory всегда остаются базы данных. Например, для важных вопросов с отчетностью, которую нужно сводить вместе, что бывает сложно на большом количестве узлов. Так что наше видение на сегодняшний день: синергия двух подходов.

Также стоит отметить, что эти два подхода тоже не совсем правильно только противопоставлять. И в то же время замыкаться на них. Производители и контрибьюторы современных систем виртуализации в режиме контейнеров, такие как Kubernetes, уже обеспечивают нам варианты для долговременного надежного хранения. Уже появились хорошие промышленные кейсы по внедрению решений, в которых хранение осуществляется в подобном виртуализированном формате.

Одна из крупнейших газет США предоставляет возможность своим читателям в режиме онлайн получить любой номер, который выходил с момента старта выпуска этой газеты в 19 веке. Можем представить уровень нагрузки. Хранение реализовано ими посредством платформы Apache Kafka, развернутой в Kubernetes. Вот еще один вариант хранения информации и предоставления к ней доступа под большой нагрузкой большому количеству

клиентов. При проектировании новых решений на этот вариант также стоит обратить внимание.

6. БАЗОВЫЕ СУБД ТИПА КЛЮЧ-ЗНАЧЕНИЕ. КАК С НИМИ РАБОТАТЬ, ПРИМЕРЫ ПРИМЕНЕНИЯ

Базовые структуры данных БД

Наверное, самой простой БД в мире, реализованная в виде двух функций, является командная оболочка Bash. Обе функции реализуют хранилище типа «ключ — значение». Можно вызвать команду db_set key value для сохранения key и value в базе данных. Затем можно вызвать команду db_get key для поиска последнего относящегося к искомому ключу значения и его возврата.

Лежащий в их основе формат хранения очень прост: он представляет собой текстовый файл, в котором каждая строка содержит пару «ключ — значение», разделенную запятой. Каждый вызов функции db_set приводит к добавлению данных в конец файла, так что при обновлении ключа несколько раз старые версии значений не будут затерты. Такой механизм называется журналом, представляющий собой файл, предназначенный только для добавления данных в его конец.

Производительность функции db_get ужасна в случае большого количества записей в БД. Каждый раз, когда нужно найти ключ, db_get приходится просматривать всю базу от начала до конца, выискивая вхождения ключа. Говоря в алгоритмических терминах, сложность поиска — порядка O(n): при удвоении количества записей n в БД поиск занимает вдвое больше времени.

Для эффективного поиска значения конкретного ключа в БД необходима другая структура данных: индекс. Общая их идея заключается в дополнительном хранении определенных метаданных, служащих своеобразным дорожным указателем, помогающим найти нужные данные.

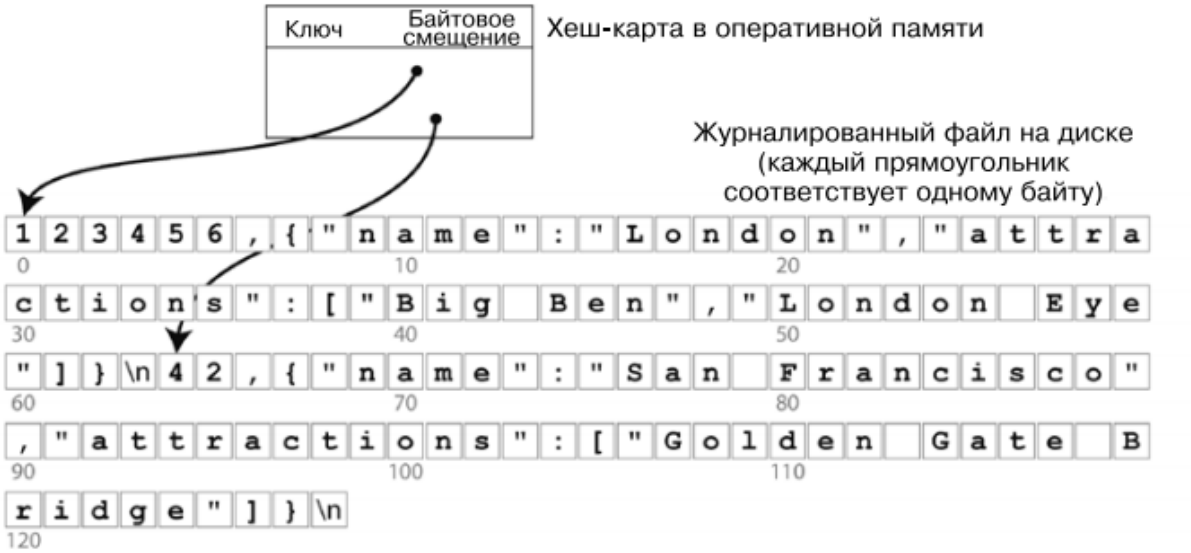
Индекс — дополнительная структура, производная от основных данных. Многие БД предоставляют возможность добавлять и удалять индексы без какого-либо воздействия на содержимое базы, это влияет только на производительность запросов. Любые индексы обычно замедляют запись, так как индекс тоже приходится обновлять всякий раз при записи данных на диск.

Это важный компромисс в системах хранения данных: хорошо подобранные индексы ускоряют запросы на чтение, но замедляют запись. Поэтому БД обычно не индексируют по умолчанию все, что можно, а заставляют разработчика приложения или администратора БД выбирать индексы вручную, на основе знания типичных для приложения паттернов запросов.

Хеш-индексы

Хранилища данных типа «ключ — значение» очень схожи с типом «словарь», реализуемым обычно в виде хеш-карты (hash map)/хеш-таблицы (hash table).

Предположим, что наше хранилище работает только путем добавления в конец файла. Тогда простейшая стратегия индексации такова: хранить в оперативной памяти хеш-карту, в которой каждому ключу поставлено в соответствие смещение (относительный адрес) в файле данных — место, где находится значение, как показано на рисунке. При добавлении в файл новой пары «ключ — значение» происходит также обновление хеш-карты для отражения в ней относительного адреса только что записанных данных. Если нужно найти значение, то можно воспользоваться хеш-картой для поиска относительного адреса в файле данных, перейти в это место и прочитать значение.



Представленный подход может показаться некоторым упрощением, но вполне жизнеспособен. Фактически именно так работает Bitcask (подсистема хранения в распределенной NoSQL СУБД Riak). Более подробно можно прочесть в книге.

Пока что рассматривали только запись в конец файла. Как же избежать ситуации исчерпания места на диске? Хорошим решением будет разбить журнал на сегменты определенного размера, закрывая файл сегмента при достижении им определенного размера и записывая последующие данные уже в новый файл. Затем можно выполнить уплотнение (compaction) этих сегментов, как показано на рисунке. Уплотнение означает отбрасывание дублирующихся ключей из журнала и сохранение только последней версии данных для каждого ключа. Более того, поскольку уплотнение часто приводит к значительному уменьшению размера сегментов, можно также слить несколько сегментов в один во время уплотнения.

Теперь у каждого сегмента имеется своя хеш-таблица в оперативной памяти, ставящая ключам смещение в файле. Чтобы найти значение для ключа, необходимо сначала заглянуть в хеш-карту последнего сегмента: если ключа там нет, то проверяется следующий по времени сегмент и т. д. Благодаря процессу слияния количество сегментов остается небольшим, поэтому при поиске не придется проверять слишком много хеш-карт.

Чтобы воплотить эту простую идею в жизнь, понадобится учесть немало нюансов. Более подробно можно прочесть в книге. Если кратко, то нужно учесть – формат файлов, удаление записи, восстановление после сбоев, недописанные записи, управление конкурентным доступом.

На первый взгляд, журналы, допускающие только добавление в конец файла, кажутся довольно неэкономными: почему бы не обновлять файл на месте, заменяя старое значение новым? Есть несколько причин:

- Добавление в конец файла и слияние сегментов — последовательные операции записи, в большинстве случаев выполняющиеся намного быстрее случайной записи.
- Конкурентный доступ и восстановление после сбоев сильно упрощаются в случае допускающих только добавление или вообще неизменяемых файлов данных.

Однако у индексов хеш-таблиц тоже есть ограничения.

- Хеш-таблица должна помещаться в оперативной памяти, так что если очень много ключей, то данный тип индекса скорее всего не подойдет. В

принципе, можно поддерживать хеш-карту на диске, но, к сожалению, добиться хорошей ее производительности непросто.

- Запросы по диапазону неэффективны. Например, невозможно с легкостью просмотреть все записи между kitty00000 и kitty99999 — необходимо искать каждый ключ отдельно в хеш-картах.

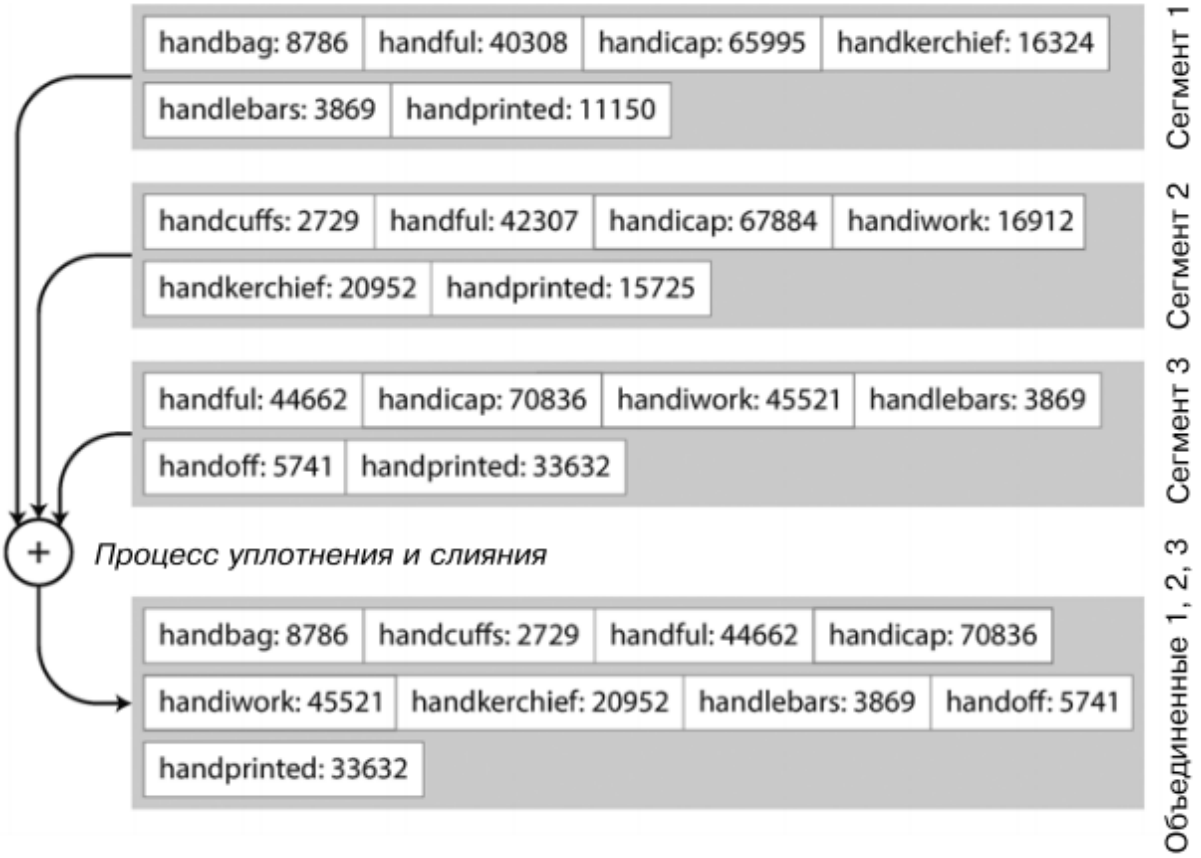
Далее рассмотрим индексную структуру, у которой нет этих ограничений.

SS-таблицы

Давайте теперь поменяем формат файлов сегментов: потребуем, чтобы последовательность пар «ключ — значение» была отсортирована по ключу.

Назовем новый формат отсортированной строковой таблицей (sorted string table, SSTable) или SS-таблицей. Потребуем также, чтобы каждый ключ встречался лишь один раз в каждом объединенном файле сегмента (процесс уплотнения сразу гарантирует это). У SS-таблиц есть несколько больших преимуществ перед журнальными сегментами с хеш-индексами.

Первое преимущество. Объединение сегментов выполняется просто и эффективно, даже если размер файлов превышает объем доступной оперативной памяти. Этот подход близок к используемому в алгоритме сортировки слиянием (mergesort). Он показан на рисунке: начинаем с одновременного чтения входных файлов, просматриваем первый ключ в каждом из файлов, копируем ключ в соответствии с порядком сортировки в выходной файл и повторяем эти действия. В результате получаем новый объединенный файл сегмента, также отсортированный по ключу. А если один и тот же ключ встретится в нескольких входных сегментах? Каждый сегмент содержит все записанные в БД значения за некоторый период времени. Это значит, что все значения одного из входных сегментов должны оказаться более свежими, чем все значения другого (при условии обязательного слияния воедино соседних сегментов). Если несколько сегментов содержат один и тот же ключ, то можно взять значение из наиболее нового сегмента и отбросить значения из более старых.



Второе преимущество. Чтобы найти в файле конкретный ключ, не нужно больше хранить индекс всех ключей в оперативной памяти. Например, нам нужен ключ handiwork, но мы не знаем точного смещения этого ключа в файле сегмента. Однако нам известно смещение для ключей handbag и handsome и (благодаря сортировке) то, что handiwork должен находиться между ними. Как следствие, можно перейти по смещению для handbag и просматривать, начиная с этого места, до тех пор, пока не найдем handiwork (впрочем, можем и не найти, если ключ отсутствует в данном файле).

Нам все еще нужен индекс в оперативной памяти, чтобы узнавать смещение для некоторых ключей, но он может быть разреженным: одного ключа для каждых нескольких килобайт файла сегмента вполне достаточно, поскольку несколько килобайт можно просмотреть очень быстро.

Третье преимущество. Поскольку для выполнения запроса на чтение все равно необходимо просмотреть несколько пар «ключ — значение» из заданного диапазона, вполне можно сгруппировать эти записи в блок и сжать его перед записью на диск. Каждая запись разреженного индекса в оперативной памяти затем будет указывать на начало сжатого блока. Помимо экономии пространства на диске, сжатие также снижает использование полосы пропускания ввода/вывода.

Создание и поддержание SS-таблиц

Как же отсортировать данные по ключу? Входящие операции записи могут производиться в любом порядке.

Поддержание отсортированной структуры на диске возможно, но гораздо проще будет делать это в оперативной памяти. Существует множество доступных для использования хорошо известных древовидных структур данных, например, красно-черные деревья. При использовании этих структур можно вставлять ключи в любой последовательности и затем читать их в нужном порядке.

Теперь организуем работу подсистемы хранения следующим образом.

- При поступлении записи добавляем ее в располагающуюся в оперативной памяти сбалансированную структуру данных (например, красно-черное дерево). Это располагающееся в оперативной памяти дерево называется MemTable (от memory table — «таблица, расположенная в памяти»).
- Когда размер MemTable превышает определенное пороговое значение — обычно несколько мегабайт, — записываем его на диск в виде файла SS-таблицы. Эта операция выполняется достаточно эффективно, поскольку дерево поддерживает пары «ключ — значение» в отсортированном по ключу виде. Новый файл SS-таблицы становится последним сегментом базы данных. А пока SS-таблица записывается на диск, операции записи продолжают выполняться в новый экземпляр MemTable.
- Время от времени запускаем в фоне процесс слияния и уплотнения, чтобы объединить файлы сегментов и отбросить перезаписанные или удаленные значения

Представленная схема работает отлично. У нее есть только одна проблема: если происходит фатальный сбой БД, то записанные позже всего данные (находящиеся в MemTable, но еще не записанные на диск) теряются. Чтобы избежать этой проблемы, можно держать на диске отдельный журнал, в конец которого немедленно добавляются все записываемые данные. Сам журнал не упорядочен, но это неважно, ведь его единственное назначение — восстановление MemTable после сбоя. Всякий раз, когда MemTable записывается в SS-таблицу, соответствующий журнал можно удалять.

Создание LSM-дерева из SS-таблиц

Подсистемы хранения, основанные на принципе слияния и уплотнения отсортированных файлов, часто называются LSM-подсистемами хранения (Log-Structured Merge).

Основная идея LSM-деревьев — применение каскада SS-таблиц, объединяемых в фоновом режиме, — проста и эффективна. Она хорошо работает даже в случае, когда размер набора данных значительно превышает

доступный объем оперативной памяти. То, что данные хранятся в отсортированном виде, дает возможность эффективно выполнять запросы по диапазонам (просмотр всех ключей между установленными минимальным и максимальным значениями), а поскольку записи на диск осуществляются последовательно, LSM-дерево способно поддерживать весьма высокую пропускную способность по записи.

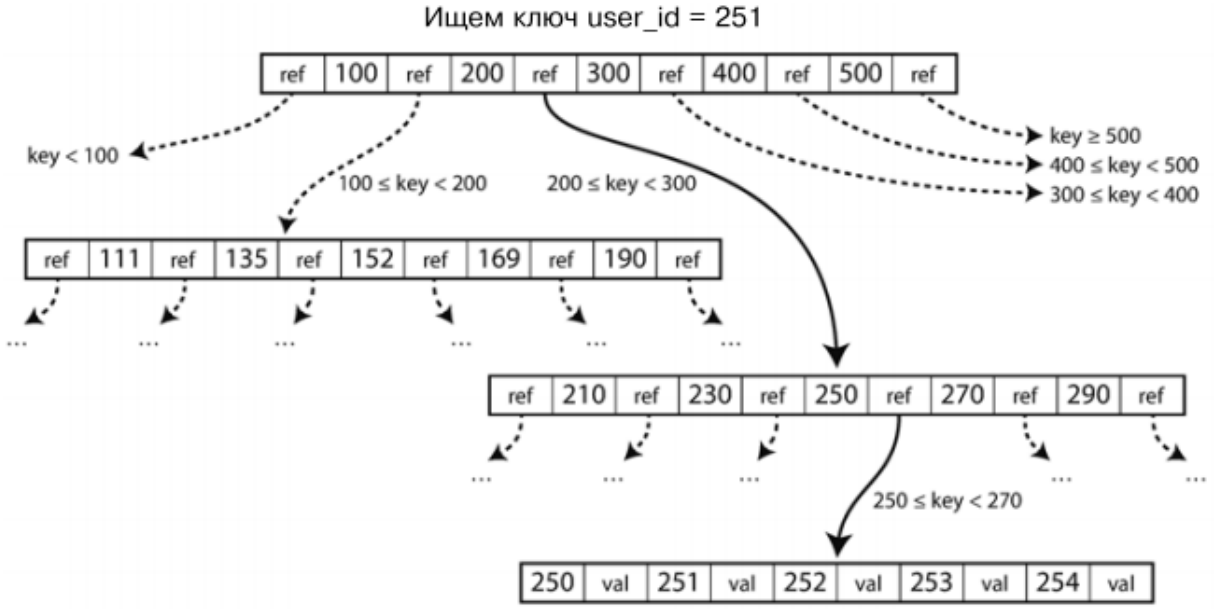
B-деревья

Наиболее широко используемая индексная структура — B-дерево. B-деревья очень хорошо выдержали испытание временем. Они остаются стандартной реализацией индексов практически во всех реляционных базах данных, да и многие нереляционные тоже их применяют.

Аналогично SS-таблицам B-деревья хранят пары «ключ — значение» в отсортированном по ключу виде, что позволяет эффективно выполнять поиск значения по ключу и запросы по диапазонам. Но на этом сходство заканчивается: конструктивные принципы B-деревьев совершенно другие.

Журналированные индексы, которые рассматривались ранее, разбивают базу данных на сегменты переменного размера и всегда записывают их на диск последовательно. В отличие от них, B-деревья разбивают БД на блоки или страницы фиксированного размера, обычно 4 Кбайт, и читают/записывают по одной странице за раз. Такая конструкция лучше подходит для нижележащего аппаратного обеспечения, поскольку диски тоже разбиваются на блоки фиксированного размера.

Все страницы имеют свой адрес/местоположение, благодаря чему одни страницы могут ссылаться на другие — аналогично указателям, но на диске, а не в памяти. Этими ссылками на страницы можно воспользоваться для создания дерева страниц, как показано на рисунке.



Одна из страниц назначается корнем B-дерева, с него начинается любой поиск ключа в индексе. Данная страница содержит несколько ключей и ссылок на дочерние страницы. Каждая из них отвечает за непрерывный диапазон ключей, а ключи, располагающиеся между ссылками, указывают на расположение границ этих диапазонов.

В случае надобности добавить новый ключ следует найти страницу, в чей диапазон попадает новый ключ, и добавить его туда. Если на странице недостаточно места для него, то она разбивается на две полупустые страницы, а родительская страница обновляется, чтобы учесть это разбиение диапазона ключей на части.

Представленный алгоритм гарантирует, что дерево останется сбалансированным, то есть глубина B-дерева с n ключами будет равна O(log n). Большинству баз данных хватает деревьев глубиной три или четыре уровня, поэтому не придется проходить по множеству ссылок на страницы с целью найти нужную (четырехуровневое дерево страниц по 4 Кбайт с коэффициентом ветвления в 500 может хранить до 256 Тбайт информации).

Количество ссылок на дочерние страницы на одной странице B-дерева называется коэффициентом ветвления

При разбиении страницы из-за ее переполнения вследствие вставки необходимо записать две новые страницы, а также перезаписать их

родительскую страницу для обновления ссылок на две дочерние страницы. Это опасная операция, ведь в случае фатального сбоя базы данных в тот момент, когда записана только часть страниц, индекс окажется поврежден.

Чтобы сделать БД отказоустойчивой, реализации B-деревьев обычно включают дополнительную структуру данных на диске: журнал упреждающей записи (writeahead log, WAL). Он представляет собой файл, предназначенный только для добавления, в который все модификации B-деревьев должны записываться еще до того, как применяться к самим страницам дерева. Когда база возвращается в норму после сбоя, этот журнал используется для восстановления B-дерева в согласованное состояние.

Поскольку B-деревья применяются уже достаточно долго, то за эти годы они претерпели немало усовершенствований.

Сравнение B- и LSM-деревьев

Хотя реализации B-деревьев в целом более совершенны, чем реализации LSM-деревьев, последние тоже представляют некоторый интерес, благодаря своей производительности. Как правило, LSM-деревья обычно быстрее при записи, а B-деревья — при чтении. Чтение выполняется медленнее на LSM-деревьях потому, что приходится просматривать несколько различных структур данных и SS-таблиц, находящихся на разных стадиях уплотнения. Однако эти оценки часто неубедительны и сильно зависят от нюансов конкретной рабочей нагрузки. Необходимо тестировать системы именно под вашей конкретной нагрузкой, чтобы сравнение было достоверным.

В книге коротко затрагиваются несколько нюансов, которые имеет смысл учесть при оценке производительности подсистемы хранения.

Автор книги затрагивает немного и другие типы индексов, например, составные и нечеткие индексы.

Обработка транзакций или аналитика?

«Транзакция» - это группа операций чтения и записи, составляющей логически единое целое. Транзакция не обязательно должна обладать свойствами ACID (Atomicity, Consistency, Isolation, Durability — атомарность,

согласованность, изоляция и сохраняемость). Обработка транзакций означает просто возможность для клиентов выполнять операции чтения и записи с низким значением задержки — в противоположность заданиям пакетной обработки, запускаемым лишь периодически (например, один раз в день).

Приложение обычно ищет с помощью индекса небольшое количество записей по какому-либо ключу. На основе вводимых пользователем данных вставляются или обновляются записи. В силу интерактивности этих приложений такой паттерн доступа получил название «обработка транзакций в реальном времени» (online transaction processing, OLTP).

Однако БД все шире используются для аналитической обработки данных (data analytics), паттерны доступа которой совершенно другие. Обычно аналитический запрос должен просматривать огромное количество записей, вычисляя сводные статистические показатели (например, количество, сумму или среднее значение) вместо возврата пользователю необработанных данных.

Эти запросы чаще всего написаны бизнес-аналитиками и вставлены в отчеты, которые руководство компании использует для оптимизации коммерческих решений (бизнес-аналитика, business intelligence). Чтобы отличать этот паттерн применения БД от обработки транзакций, его назвали аналитической обработкой данных в реальном времени (online analytical processing, OLAP).

Смысл слова online в OLAP не вполне четко определен; вероятно, речь идет не только о том, что эти запросы используются для встроенных отчетов, но и что аналитики могут задействовать OLAP-системы интерактивно для исследовательских запросов.

Свойство	Системы обработки транзакций (OLTP)	Аналитические системы (OLAP)
Основной паттерн чтения	Небольшое количество записей на один запрос, извлекается по ключу	Агрегирование по большому количеству записей
Основной паттерн записи	Произвольный доступ, операции записи с низким значением задержки на основе вводимых пользователем данных	Групповой импорт (ETL) или поток событий
В основном применяется	Конечными пользователями/ заказчиками, через веб-приложение	Штатным аналитиком, для поддержки принятия решений
Какие данные отражает	Актуальное состояние данных (текущий момент времени)	Историю событий, происходивших на протяжении отрезка времени
Размер набора данных	От гигабайтов до терабайтов	От терабайтов до петабайтов

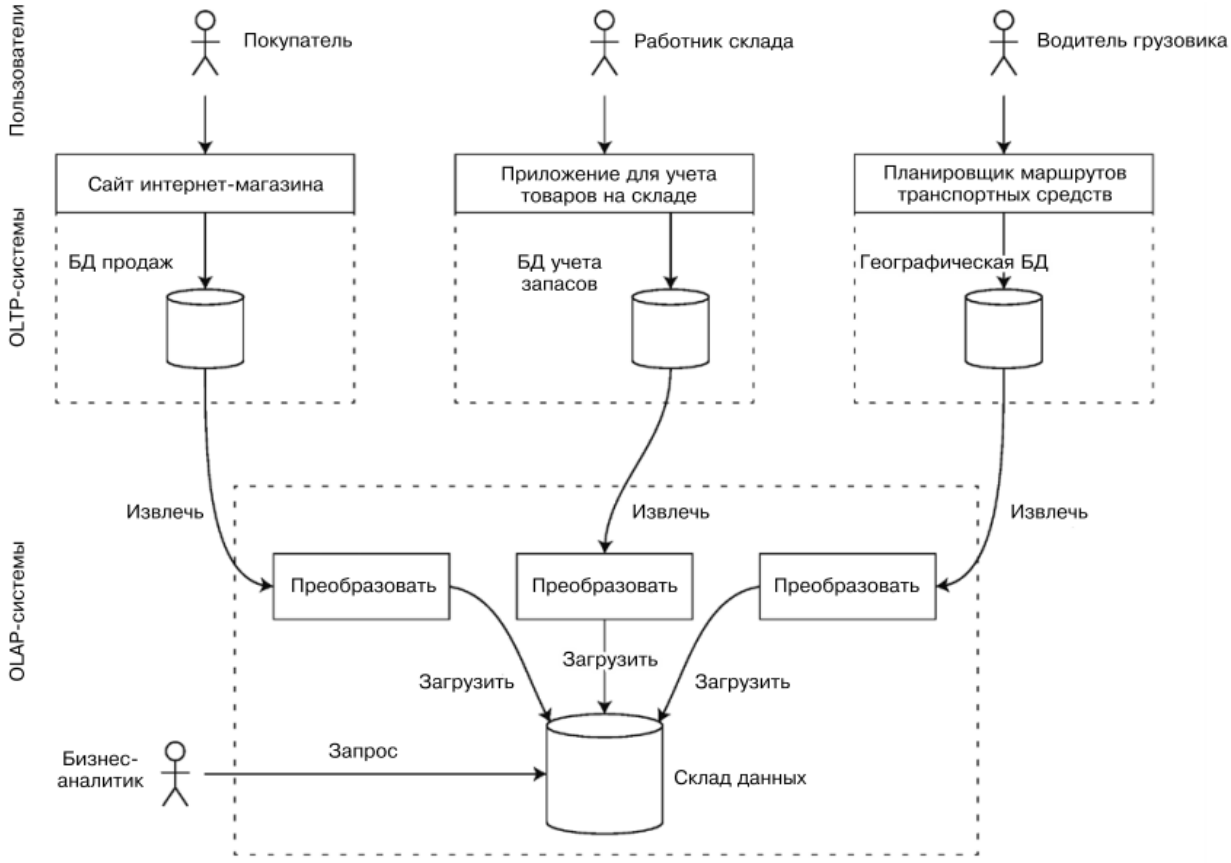
Сначала как для обработки транзакций, так и для аналитических запросов использовались одни и те же базы данных. Язык SQL оказался в этом смысле весьма гибок: он работает при OLTP-запросах ничуть не хуже, чем при OLAP-запросах. Тем не менее в конце 1980-х — начале 1990-х годов возникла такая тенденция: компании прекращали задействовать OLTP-системы для целей аналитики и выполняли анализ на отдельных БД, которые назывались складами данных (data warehouse).

Складирование данных

У предприятия могут быть десятки различных систем обработки транзакций. Все эти системы сложны и требуют команды разработчиков для поддержки, так что в итоге эксплуатируются практически автономно друг от друга.

Обычно от этих OLTP-систем ожидается высокая доступность и обработка транзакций с низкой задержкой, поскольку они зачастую критичны для работы бизнеса. Администраторы БД обычно крайне неохотно разрешают бизнес-аналитикам выполнять произвольные аналитические запросы на этих базах, ведь эти запросы зачастую оказываются ресурсоемкими, связаны с просмотром больших частей набора данных, что может отрицательно сказаться на производительности выполняемых в этот момент транзакций.

Склад данных, напротив, представляет собой отдельную БД, которую аналитики могут опрашивать так, как им заблагорассудится, не влияя при этом на OLTP-операции. Склад содержит предназначенную только для чтения копию данных из всех различных OLTP-систем компании. Данные извлекаются из баз OLTP (с помощью выполнения периодических дампов данных или непрерывного потока обновлений данных), преобразуются в удобный для анализа вид, очищаются и затем загружаются в склад. Процесс их помещения в склад известен под названием «извлечение — преобразование — загрузка» (extract — transform — load, ETL).



Большим преимуществом использования отдельного склада данных, а не выполнения запросов непосредственно к OLTP-системам является то, что склады можно оптимизировать в расчете на аналитические паттерны доступа.

Что же происходит при сохранении данных в базе и что делает база, когда позднее запрашиваются эти данные? На высоком уровне абстракции мы видим, что подсистемы хранения делятся на две широкие категории: оптимизированные для обработки транзакций (OLTP) и оптимизированные

для аналитики (OLAP). Между паттернами доступа в этих сценариях использования есть большие различия.

- OLTP-системы обычно нацелены на работу с пользователями, и это означает огромное потенциальное количество запросов. Чтобы справиться с такой нагрузкой, приложения обычно затрагивают в каждом запросе только небольшое число строк. Программы запрашивают записи с помощью определенного ключа, а подсистема хранения задействует индекс для поиска данных с соответствующим ключом. Узким местом здесь обычно становится время перехода к нужной позиции на диске.
- Склады данных и подобные им аналитические системы менее широко известны, поскольку они в основном применяются бизнес-аналитиками, а не конечными пользователями. Склады обрабатывают намного меньшее количество запросов, чем OLTP-системы, но все запросы обычно очень ресурсоемки и требуют просмотра миллионов строк за короткое время. Узким местом здесь обычно становится пропускная способность диска.

С точки зрения OLTP мы рассмотрели два различных подхода к подсистемам хранения.

- Журналированный подход, при котором допускается только дописывание данных в файлы и удаление устаревших файлов, а не обновление записанного файла. Сюда относятся: Bitcask, SS-таблицы, LSM-деревья и другие.
- Подход с обновлением на месте, при котором диск рассматривается как набор страниц заданного размера, допускающих перезапись. Крупнейший представитель этой философии — B-деревья, используемые во всех основных реляционных базах данных, а также и во многих нереляционных.

7. СПОСОБЫ ОБЪЕДИНЕНИЯ ДВУХ МНОЖЕСТВ ПО ПРЕДИКАТУ УКАЗАННЫМИ СПОСОБАМИ

Множество

Множество – это математический объект, являющийся набором, совокупностью, собранием каких-либо объектов, которые называются элементами этого множества. Или другими словами:

Множество – это не более чем неупорядоченная коллекция уникальных элементов.

Что значит неупорядоченная? Это значит, что два множества эквивалентны, если содержат одинаковые элементы.

Элементы множества должны быть *уникальными*, множество не может содержать одинаковых элементов. Добавление элементов, которые уже есть в множестве, не изменяет это множество.

Множества, состоящие из конечного числа элементов, называются *конечными*, а остальные множества – *бесконечными*. Конечное множество, как следует из названия, можно задать перечислением его элементов. Так как темой этой статьи является практическое использование множеств в Python, то я предлагаю сосредоточиться на *конечных* множествах.

Множества в Python

Множество в Python можно создать несколькими способами. Самый простой – это задать множество перечислением его элементов в фигурных скобках:

fruits = {"banana", "apple", "orange"}

Единственное ограничение, что таким образом нельзя создать пустое множество. Вместо этого будет создан пустой словарь:

wrong_empty_set = {}

print(type(wrong_empty_set))

Для создания пустого множества нужно непосредственно использовать `set()`:

correct_empty_set = set()

print(type(correct_empty_set))

Хешируемые объекты

Существует ограничение, что элементами множества (как и ключами словарей) в Python могут быть только так называемые хешируемые (Hashable) объекты. Это обусловлено тем фактом, что внутренняя реализация *set* основана на хеш-таблицах. Например, списки и словари – это изменяемые объекты, которые не могут быть элементами множеств. Большинство неизменяемых типов в Python (*int*, *float*, *str*, *bool*, и т.д.) – хешируемые. Неизменяемые коллекции, например *tuple*, являются хешируемыми, если хешируемы все их элементы.

Объекты пользовательских классов являются хешируемыми по умолчанию. Но практического смысла чаще всего в этом мало из-за того, что сравнение таких объектов выполняется по их адресу в памяти, т.е. невозможно создать два "равных" объекта.

Чтобы протокол хеширования работал без явных и неявных логических ошибок, должны выполняться следующие условия:

- Хеш объекта не должен изменяться, пока этот объект существует
- Равные объекты должны возвращать одинаковый хеш

Свойства множеств

Тип set в Python является подтипом Collection (про коллекции), из данного факта есть три важных следствия:

- Определена операция проверки принадлежности элемента множеству
- Можно получить количество элементов в множестве
- Множества являются iterable-объектами

Принадлежность множеству

Проверить принадлежит ли какой-либо объект множеству можно с помощью оператора `in`. Это один из самых распространённых вариантов

использования множеств. Такая операция выполняется в среднем за `O(1)` с теми же оговорками, которые существуют для хеш-таблиц.

Мощность множества

Мощность множества – это характеристика множества, которая для конечных множеств просто означает количество элементов в данном множестве. Для бесконечных множеств всё несколько сложнее.

Отношения между множествами

Между множествами существуют несколько видов отношений, или другими словами взаимосвязей. Давайте рассмотрим возможные отношения между множествами в этом разделе.

Равные множества

Тут всё довольно просто – два множества называются равными, если они состоят из одних и тех же элементов. Как следует из определения множества, порядок этих элементов не важен.

Непересекающиеся множества

Если два множества не имеют общих элементов, то говорят, что эти множества не пересекаются. Или другими словами, пересечение этих множеств является пустым множеством.

Подмножество и надмножество

Подмножество множества S – это такое множество, каждый элемент которого является также и элементом множества S. Множество S в свою очередь является надмножеством исходного множества.

Пустое множество является подмножеством абсолютно любого множества.

Само множество является *подмножеством* самого себя.

Операции над множествами

Рассмотрим основные операции, опредяляемые над множествами.

Объединение множеств

Объединение множеств – это множество, которое содержит все элементы исходных множеств. В Python есть несколько способов объединить множества, давайте рассмотрим их на примерах.

Добавление элементов в множество

Добавление элементов в множество можно рассматривать как частный случай объединения множеств за тем исключением, что добавление элементов изменяет исходное множество, а не создает новый объект. Добавление одного элемента в множество работает за O(1).

Пересечение множеств

Пересечение множеств – это множество, в котором находятся только те элементы, которые принадлежат исходным множествам одновременно.

При использовании оператора & необходимо, чтобы оба операнда были объектами типа set. Метод intersection, в свою очередь, принимает любой iterable-объект. Если необходимо изменить исходное множество, а не возвращать новое, то можно использовать метод intersection_update, который работает подобно методу intersection, но изменяет исходный объект-множество.

Разность множеств

Разность двух множеств – это множество, в которое входят все элементы первого множества, не входящие во второе множество.

Удаление элементов из множества Удаление элемента из множества можно рассматривать как частный случай *разности*, где удаляемый элемент – это одноэлементное множество. Следует отметить, что удаление элемента, как и в аналогичном случае с добавлением элементов, изменяет исходное множество. Удаление одного элемента из множества имеет вычислительную сложность `O(1)`.

Также у множеств есть метод `difference_update`, который принимает iterable-объект и удаляет из исходного множества все элементы iterable-объекта. Этот метод работает аналогично методу `difference`, но изменяет исходное множество, а не возвращает новое.

Симметрическая разность множеств

Симметрическая разность множеств – это множество, включающее все элементы исходных множеств, не принадлежащие одновременно обоим исходным множествам. Также симметрическую разность можно рассматривать как разность между объединением и пересечением исходных множеств.

Предикаты

Перед тем, как SQL Server приступит к поиску по индексу, он должен определить, являются ли ключи индекса подходящими для оценки предиката запроса.

Индексы по одному столбцу

С индексами по одному столбцу всё довольно просто. SQL Server может их использовать для самых простых сравнений, например, равенства и неравенства (больше чем, меньше чем, и т.д.). Более сложные выражения, такие как функции по столбцу и предикаты "LIKE" с символами подстановки, будут в таких случаях создавать трудности для использования поиска по индексу.

Например, предположим, что мы имеем индекс по одному столбцу, созданный по полю "a". Этот индекс может использоваться для поиска при следующих предикатах:

Индексы по нескольким столбцам

С индексами по нескольким столбцам дело обстоит сложнее. Для таких индексов важен порядок ключей. Этим определяется порядок сортировки индекса, и от порядка ключей зависит набор предикатов поиска, которые SQL Server сможет использовать для этого индекса.

Для того, чтобы проще понять важность порядка ключей, представьте себе телефонную книгу. Для телефонной книги походит индекс с ключами: "фамилия" и "имя". Содержание телефонной книги отсортировано по фамилии, что упрощает поиск кого-нибудь, если мы знаем его фамилию. Однако, если мы знаем только имя, очень трудно получить список людей с

необходимым нам именем. В таком случае, нам бы лучше подошла другая телефонная книга, в которой абоненты отсортированы по имени.

Точно также обстоит дело, если мы имеем индекс по двум столбцам, т.е. мы сможем использовать индекс только для предиката по второму столбцу, если указан предикат равенства для первого столбца. Даже если мы не сможем использовать индекс для удовлетворения условия предиката второго столбца, мы сможем использовать его для первого столбца. В этом случае, вводится остаточный предикат для предиката второго столбца, который будет тем же самым предикатом, который используется для просмотра.

Например, предположим, что у нас есть индекс по двум столбцам "a" и "b". Мы можем его использовать для поиска по любому из предикатов, которые применимы для индексов по одному столбцу. Кроме того, можно использовать это индекс и для поиска со следующими дополнительными предикатами:

a = 3.14 and b = 'pi'

a = 'xyzzy' and b <= 0

Для следующих ниже примеров, мы можем использовать индекс для удовлетворения условий предиката для столбца "a", но не для столбца "b". В этих случаях потребуется остаточный предикат:

a > 100 and b > 100

a like 'abc%' and b = 2

И, наконец, невозможно использовать индекс для поиска со следующим ниже набором предикатов, поскольку поиск невозможен даже по столбцу "a". В таких случаях, оптимизатор вынужден использовать другой индекс (например, такой индекс, у которого столбец "b" указан первым), или он будет использовать просмотр с остаточным предикатом.

b = 0

a + 1 = 9 and b between 1 and 9

a like '%abc' and b in (1, 3, 5)

Добавим в пример немного конкретики.

Рассмотрим следующую схему:

create table person (id int, last_name varchar(30), first_name varchar(30))

create unique clustered index person_id on person (id)

create index person_name on person (last_name, first_name)

Ниже представлены три запроса с соответствующими им текстовыми планами исполнения. Первый запрос осуществляет поиск по обоим столбцам индекса person_name. Второй запрос ищет только по первому столбцу и использует остаточный предикат, для оценки first_name. Третий запрос не может использовать поиск и использует просмотр с остаточным предикатом.

select `id` *from* `person` *where* `last_name` *= 'Doe' and* `first_name` *= 'John'*

select `id` *from* `person` *where* `last_name` *> 'Doe' and* `first_name` *= 'John'*

select `id` *from* `person` *where* `last_name` *like '%oe' and* `first_name` *= 'John'*

Внимание: Если Вы пробуете воспроизвести эти планы для этих и некоторых следующих примеров, учтите, что я использовал подсказки индексов (которые не указаны), позволяющие гарантировать получение необходимого плана запроса, поскольку я хотел проиллюстрировать эти примеры без необходимости вставки данных в таблицу.

Hash Join

Когда Вы встречаете случай использования оператора Hash Join (хэш-соединение), это говорит о наличии тяжелого запроса. В отличие то соединения Nested Loops Join, которое хорошо для относительно маленьких наборов данных, и от соединения Merge Join, которое помогает при умеренных размерах наборов данных, хэш-соединение превосходит другие типы соединений при необходимости соединения огромных наборов данных. Хэш-соединения распараллеливается и масштабируется лучше любого другого соединения и сильно выигрывает при большой производительности

информационных хранилищ (я вернусь к обсуждению параллельного выполнения запросов в следующей серии статей).

Хэш-соединение имеет много общего с соединением слиянием. Подобно соединению слиянием, для него требуются не менее одного предиката объединения по эквивалентности, оно поддерживает остаточные предикаты, а также все внешние соединения и полусоединения. В отличие от соединения слиянием, для него не требуется наличие упорядоченных входных потоков и для поддержки полного внешнего соединения требуется наличие предиката соединения по эквивалентности.

Алгоритм

Хэш-соединение выполняется в две стадии: компоновка и проба. Во время компоновки осуществляется чтение всех строк первого входного потока (часто, его называют левым потоком или потоком компоновки), хеширование строк по ключам соединения эквивалентности и создание в оперативной памяти хеш-таблицы. Во время пробы осуществляется чтение всех строк второго входного потока (часто его называют правым потоком или пробным потоком), хеширование строк этого потока по тем же ключам соединения эквивалентности, а потом осуществляется просмотр или поиск соответствий строк в хеш-таблице. Так как хеш-функции могут быть подвержены коллизиям (два разных значения ключа имеют одинаковый хэш), приходится проверять каждое потенциальное соответствие, что необходимо для гарантии того, что в этом случае действительно совпадение обусловлено соединением, а не коллизией.

Алгоритм в псевдокоде:

```
for each row R1 in the build table
   begin
      calculate hash value on R1 join key(s)
      insert R1 into the appropriate hash bucket
   end
```

```
for each row R2 in the probe table
   begin
      calculate hash value on R2 join key(s)
      for each row R1 in the corresponding hash bucket
         if R1 joins with R2
            return (R1, R2)
   end
```

Обратите внимание, что в отличие от вложенных циклов и слияния, которые немедленно начинают выдавать результат, хэш-соединение блокируют вывод до окончания компоновки. То есть вначале должны быть полностью прочитан и обработан поток компоновки, и только потом могут быть возвращены какие-либо строки. Кроме того, в отличие от других методов соединения, хэш-соединение требует предоставления памяти для хранения хеш-таблиц. Таким образом, существует предел числа параллельных хэш-соединений, которые SQL Server может поддерживать в какой то момент времени. Хотя эти ограничения не являются проблемой для информационных хранилищ, они могут оказаться нежелательными для большинства OLTP приложений.

Память и сброс на диск

До начала выполнения хэш-соединения, SQL Server пытается оценить, сколько памяти потребуется для хеш-таблиц. Выполняется оценка кардинальности элементов для размера потока компоновки, базирующаяся на ожидаемом среднем размере строк, что позволяет оценить требующийся объем и конфигурацию памяти. Для уменьшения задействуемой хэш-соединением памяти, в качестве таблицы потока компоновки выбирается самая маленькая из двух таблиц. После этого выполняется попытка резервирования такого объёма памяти, который бы гарантировал успешное выполнение хэш-соединения.

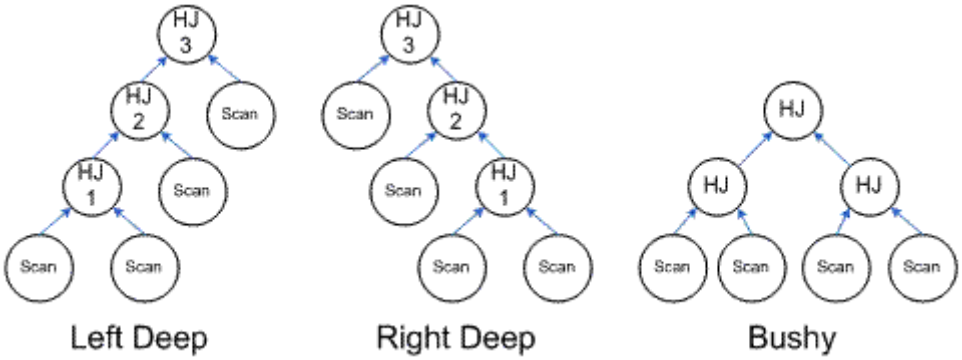
Что же случается, если хэш-соединению будет предоставлено меньше памяти, чем было запрошено, или если оценка оказалось слишком низкой? В этих случаях, на стадии компоновки хэш-соединение может выйти за пределы памяти. Если хэш-соединение вышло за пределы памяти, начинается сброс на диск небольшого процента от всей хеш-таблицы (во временный объект tempdb). Хэш-соединение следит, какие разделы хеш-таблицы все еще находятся в памяти, а какие были сброшены на диск. Поскольку каждая новая строка читается из таблицы компоновки, можно легко проверить, является ли она хэшем раздела в оперативной памяти, или она находиться в разделе на диске. Если это хэш в оперативной памяти, всё происходит по обычной схеме. Если это хэш на дисковом разделе, осуществляется запись строки на диск. Этот процесс позволяет управлять превышением памяти и сбросом разделов на диск, и он может занимать много времени, до тех пор, пока стадия компоновки не будет закончена.

Подобный процесс исполняется во время стадии пробы. Для каждой новой строки из таблицы пробного потока выполняется проверка, является ли она хэшем раздела в оперативной памяти или на диске. Если это хэш раздела в оперативной памяти, берутся пробы из хеш-таблицы, генерируются соответствующие условиям соединения строки, и после этого строка отбрасывается. Если это хэш раздела на диске, осуществляется запись строки на диск. Как только будет завершен первый проход таблицы пробного потока, процесс возвращается поочерёдно к другим разделам, которые были сброшены на диск, считывает в память строки потока компоновки, восстанавливает хеш-таблицу для каждого из разделов, и затем считывает соответствующее разделы пробного потока, завершая соединение.

Сравнение Left Deep, Right Deep и Bushy - деревьев хэш-соединений

Эти термины относятся к форме плана исполнения запроса, что проиллюстрировано на этом рисунке:

Форма дерева соединения особенно важна для хэш-соединений, поскольку от этого зависит потребление памяти.



В густом слева дереве, выходной поток одного хэш-соединения является входным потоком компоновки следующего хэш-соединения. Поскольку хэш-соединения должны получить весь входной поток компоновки до того, как начнётся стадия проб, в густом слева дереве активными одновременно могут быть только смежные пары хэш-соединений. Например, на показанном выше рисунке всё начинается с построения хеш-таблицы для HJ1. Когда HJ1 перейдёт на стадию проб, становится возможным использовать выходной поток HJ1, который нужен для формирования хеш-таблиц для HJ2. Когда HJ1 прошёл стадию проб, используемая для хеш-таблиц память может быть освобождена. Только после этого можно начинать стадию проб для HJ2 и строить хеш-таблицу для HJ3. Таким образом, HJ1 и HJ3 никогда не будут активны одновременно и поэтому могут использовать одно и то же распределение памяти. Полный объем необходимой памяти займёт: max(HJ1 + HJ2, HJ2 + HJ3).

В густом справа дереве, выходной поток одного хэш-соединения является входным пробным потоком следующего хэш-соединения. Все хэш-соединения должны полностью сформировать свои хеш-таблицы до того, как начнётся стадия проб. Все хэш-соединения активны одновременно и не могут использовать одну и ту же память совместно. Когда начинается стадия проб, поток строк заполняет дерево хэш-соединений, без использования блокировок. Таким образом, полный объем необходимой памяти будет равен: HJ1 + HJ2 + HJ3.

8. ПОНЯТИЕ И ВИДЫ ИНДЕКСОВ В БАЗАХ ДАННЫХ

Индексирование баз данных — это техника, повышающая скорость и эффективность запросов к базе данных. Она создаёт отдельную структуру данных, сопоставляющую значения в одном или нескольких столбцах таблицы с соответствующими местоположениями на физическом накопителе, что позволяет базе данных быстро находить строки по конкретному запросу без необходимости сканирования всей таблицы. Применяются разные типы индексов, однако они занимают пространство и должны обновляться при изменении данных. Важно тщательно продумывать стратегию индексирования базы данных и регулярно её оптимизировать.

Как базы данных создают индексы

Индексирование базы данных обычно выполняется при помощи алгоритма, определяющего, как должен создаваться и храниться индекс. Конкретный процесс создания индекса может варьироваться в зависимости от типа используемой системы базы данных, однако в целом общие этапы выглядят так:

1. Определение столбца или столбцов в таблице базы данных, которые нужно индексировать. Обычно они определяются по тому, какие столбцы чаще всего используются в запросах или поисках.

2. Выбор алгоритма индексирования, подходящего для типа индексируемых данных. Например, индексы в виде B-деревьев обычно используются для индексирования строковых или числовых данных, а полнотекстовые индексы — для индексирования текстовых данных.

3. Применение алгоритма индексирования к выбранным столбцам, что создаёт структуру данных, сопоставляющую значения в столбцах с местоположениями соответствующих записей таблицы.

4. Сохранение индекса в отдельной структуре данных, обычно в другой части диска или в памяти, чтобы доступ к ней был более эффективным, чем к соответствующим табличным данным.

5. Обновление индекса в случае добавления новых записей, удаления или изменения записей в таблице.

Создание индекса может существенно улучшить производительность запросов к базе данных и операций поиска, поскольку оно позволяет системе базы данных находить соответствующие записи быстрее и эффективнее. Однако индексирование также может обладать и недостатками, например, увеличение требований к объёму хранилища и замедление выполнения операций вставки и обновления, поэтому перед созданием индекса следует взвесить плюсы и минусы.

Алгоритмы индексирования

Как говорилось выше, существует множество алгоритмов индексирования, используемых для оптимизации скорости операций получения данных при помощи создания индексов столбцов таблиц. Вот некоторые из самых популярных алгоритмов индексирования баз данных:

- B-дерево
- Bitmap-индекс
- Хэш-индекс
- GiST (Generalized Search Tree, обобщённое поисковое дерево)
- Полнотекстовый индекс

Каждый алгоритм индексирования имеет свои сильные и слабые стороны; давайте рассмотрим их по порядку.

B-дерево

B-дерево — это структура данных самобалансирующегося дерева, которая часто используется в качестве алгоритма индексирования в базах данных. Каждый узел дерева состоит из набора ключей и указателей на дочерние узлы; хранение данных осуществляется в иерархической структуре. Деревья B-узлов упорядочены таким образом, что позволяют быстро выполнять поиск, вставку и удаление данных.

Самое большое преимущество алгоритма B-дерева заключается в минимизации количества дисковых операций ввода-вывода, необходимых для

доступа к данным, потому что в B-дереве все узлы-листья находятся на одном уровне, а каждый узел может хранить множество ключей и указателей. Количество ключей и указателей, которое может храниться в узле, определяется параметром, называемым «порядок» дерева.

Алгоритм B-дерева работает следующим образом:

1. Инициализация: при создании B-дерева создаётся пустой корневой узел. Параметр, задающий максимальное количество ключей («порядок»), которые могут храниться в каждом узле, управляет B-порядком дерева.

2. Вставка: при добавлении нового узла в B-дерево алгоритм сначала подыскивает подходящий узел-лист, в который нужно вставить ключ. B-дерево разделяет заполненный узел-лист на два новых узла и перемещает медианный ключ в родительский узел. Пока не достигнут корневой узел, процесс разделения может распространяться по дереву. Благодаря этой процедуре дерево остаётся сбалансированным, а узлы-листья находятся на одинаковой высоте.

3. Удаление: когда ключ удаляется из B-дерева, алгоритм ищет узел, который изначально хранил ключ. Если узел-лист хранил ключ, то ключ извлекается и узел может нуждаться в перебалансировке. Алгоритм удаляет предшествующий или последующий лист после листа-узла, удалив ключ с ним, если ключ обнаружен не в узле-листе.

4. Поиск: в процессе поиска ключа в B-дереве алгоритм начинает с корневого узла и рекурсивно движется к веткам, пока не найдёт нужный узел-лист. Метод поиска сравнивает искомый ключ с ключами, содержащимися в каждом узле, а затем использует соответствующий указатель для перехода к дочернему узлу, в котором может находиться ключ. Этот процесс продолжается, пока не будет найден искомый ключ или пока не будет определено, что он отсутствует в дереве.

Однако B-деревья обладают некоторыми недостатками:

- Излишняя трата ресурсов: B-деревья задействуют большой объём излишнего пространства, поскольку каждый узел в B-дереве содержит указатель на родительский и дочерний узлы.
- Сложность: алгоритмы, используемые для вставки, удаления и поиска данных в B-дереве, сложнее по сравнению с другими структурами данных. Это усложняет реализацию и поддержку B-деревьев.
- Медленные обновления: обновление данных в B-дереве может быть относительно медленным по сравнению с другими структурами данных. Каждая операция обновления требует множества операций доступа к диску, и этот процесс может быть медленным для больших B-деревьев.

Bitmap-индексирование

Bitmap-индексирование — это методика индексирования данных, использующая битовые карты (bitmap) для обозначения наличия или отсутствия значения в таблице. Это успешная техника индексирования для таблиц с низкой кардинальностью, где количество уникальных значений в столбце довольно мало по сравнению с общим количеством строк.

Bitmap-индексирование может быть очень эффективным для столбцов с низкой кардинальностью, поскольку битовые карты крайне компактны и их можно быстро сканировать для извлечения данных. Bitmap-индексы очень удобны для применения в хранилищах данных, где необходимо быстро сканировать огромные объёмы данных. Кроме того, они полезны для баз данных, в которых много операций чтения, но мало обновлений или вставок.

- Для создания bitmap-индекса столбца для каждого уникального значения столбца создаётся отдельный bitmap. Каждый bitmap имеет длину, равную количеству строк в таблице.
- Если значение присутствует в строке, соответствующему биту в bitmap присваивается значение 1, а если оно отсутствует, то присваивается значение 0. (Представьте таблицу, где столбец «Gender» имеет два уникальных значения, например, «Male» и «Female». Если этот столбец имеет bitmap-индекс, можно создать два bitmap, длина каждого из которых равна количеству

строк в таблице. Когда в строке встречается «Male» или «Female», соответствующий бит в bitmap «Male» или «Female» получает значение 1, и наоборот. В случае отсутствия значения «Male» или «Female» соответствующему биту присваивается значение 0.)

- Чтобы выполнить запрос при помощи bitmap-индекса, соответствующие в запросе значения bitmap комбинируются при помощи побитовых операторов AND, OR и NOT. (например, если мы хотим найти все строки, где «Gender» равно «Male» И «Age» больше 30, нам сначала нужно получить bitmap «Male» и bitmap «Age > 30» из соответствующих индексов. Затем мы комбинируем эти два bitmap при помощи побитового оператора AND и получаем окончательный bitmap только с единицами в тех позициях, где оба условия истинны. Затем окончательный bitmap используется для получения из таблицы строк, удовлетворяющих запросу.)

Bitmap-индексы имеют множество недостатков, и в том числе:

- Большой размер: Bitmap-индексы могут быть большими, особенно при работе с крупными датасетами. Из-за этого они могут оказаться менее эффективными, чем другие методики индексирования.
- Столбцы с высокой кардинальностью: Bitmap-индексы неэффективны для столбцов с высокой кардинальностью, где количество уникальных значений очень высоко. В таких случаях bitmap-индексы могут становиться очень большими и не помещаться в памяти.
- Смещённое распределение данных: если данные смещены, у нескольких значений может быть гораздо более высокая частота, чем у других, и bitmap-индексы окажутся неэффективными. Это вызвано тем, что bitmap для наиболее частых значений становятся очень большими и могут доминировать в индексе.

Хэш-индекс

Хэш-индекс — это разновидность методики индексирования баз данных, использующая хэш-функцию для сопоставления ключей индекса с

местоположениями соответствующих записей данных. Это быстрый метод индексирования для запросов точного соответствия в одном столбце.

Сопоставление ключей индекса с местоположениями соответствующих записей данных позволяет выполнять быстрый поиск и вставки за постоянное время O(1). Однако этот метод плохо работает с запросами диапазонов или частичными совпадениями и может страдать от коллизий, с которыми можно справляться при помощи различных техник разрешения коллизий.

Чтобы объяснить, как работает хэш-индекс, давайте рассмотрим пример. Допустим, у нас есть таблица базы данных, содержащая информацию о сотрудниках, в том числе, номера их пользовательских ID. Мы хотим создать хэш-индекс столбца пользовательских ID, чтобы получить возможность быстрого поиска данных пользователей на основании номера их ID.

1. Мы создадим хэш-функцию, получающую на входе пользовательский ID и генерирующую на выходе уникальный хэш-код. Хэш-функция должна быть спроектирована таким образом, чтобы генерировать равномерно распределённое множество хэш-кодов для равномерного распределения записей по корзинам в файле индекса. На практике хэш-функция может использовать для генерации хэш-кода различные методики, например, модульную арифметику или побитовые операции.

2. Мы создаём файл хэш-индекса, содержащий набор корзин (bucket), каждая из которых соответствует уникальному хэш-коду сгенерированному хэш-функцией. Каждая корзина содержит указатель на файл базы данных, содержащий записи для этого хэш-кода.

3. При выполнении запроса к значению запроса применяется хэш-функция для генерации хэш-кода. Затем хэш-код используется для нахождения соответствующей корзины в файле хэш-индекса. Записи с одинаковым хэш-кодом хранятся в одной корзине, поэтому мы можем просто просканировать записи в этой корзине и найти совпадающую запись/записи. Если присутствуют коллизии (то есть несколько записей с одинаковым хэш-

кодом), то для их разрешения можно использовать техники наподобие создания цепочек или открытой адресации.

4. Чтобы вставить новую запись в хэш-индекс, мы применяем к значению ключа записи хэш-функцию, чтобы сгенерировать его хэш-код, а затем вставляем запись в соответствующую корзину в файле хэш-индекса. Если коллизии отсутствуют, вставку можно выполнить за постоянное время O(1), так как нам нужно всего лишь вычислить хэш-код и вставить запись в корзину. Если коллизии есть, нам может потребоваться проделать дополнительные операции, например, вставку записи в связанный список в корзине или проверку других корзин, пока не будет найден свободный слот.

Хэш-индексы также имеют множество недостатков, в том числе:

- Ограниченные возможности поиска: хэш-индексы предназначены для обработки только поисков равенства (например, «найти все записи, где столбец A равен значению»). Они плохо подходят для запросов диапазонов или сортировки.
- Коллизии: хэш-индексы могут иметь коллизии, при которых несколько ключей соответствуют одному хэш-значению. Это может привести к снижению производительности, поскольку базе данных нужно будет выполнять дополнительные операции для разрешения коллизий.
- Непредсказуемые требования к размеру хранилища: размер хэш-индекса невозможно предугадать, так как он зависит от количества уникальных значений в индексируемом столбце. Это усложняет планирование требований к размеру хранилища.

GiST

GiST (Generalized Search Tree, обобщённое поисковое дерево) — это техника индексирования баз данных, которая может использоваться для индексирования сложных типов данных, например, геометрических объектов, текста или массивов. Это сбалансированная древовидная структура, состоящая из узлов с множественными дочерними узлами. Каждый узел описывает диапазон или множество значений и связан с предикативной

функцией, проверяющей, принадлежит ли значение диапазону или множеству. Предикативная функция зависит от типа индексируемых данных и может быть подстроена под разные типы данных.

Чтобы проиллюстрировать принцип работы индекса GiST, рассмотрим пример индексирования пространственных данных. Допустим, у нас есть таблица базы данных, содержащая информацию о городах, в том числе их названия и координаты в формате широты и долготы.

1. Зададим множество предикатов и функций преобразования, специфичных для индексируемого типа пространственных данных. В данном случае мы должны задать предикат, проверяющий, находится ли заданная точка в ограничивающем прямоугольнике, описанном узлом в индексе, и функцию преобразования, преобразующую точку в набор ключей на основании её позиции в ограничивающем прямоугольнике.

2. Создаём файл индекса GiST, состоящий из множества узлов, каждый из которых описывает ограничивающий прямоугольник, охватывающий диапазон координат.

Корневой узел описывает весь диапазон координат в таблице базы данных, а каждый дочерний узел описывает подмножество этого диапазона.

Каждый узел связывается с предикативной функцией и функцией преобразования, специфичными для индексируемого типа пространственных данных.

3. При выполнении запроса значение в запросе преобразуется при помощи функции преобразования в набор ключей.

Затем ключи сравниваются с предикатами, связанными с каждым узлом индекса, начиная с корневого узла.

Поиск продолжается вниз по дереву и выбирает дочерний узел, содержащий значение из запроса.

Процесс повторяется, пока не будет достигнут узел-лист, содержащий элементы индекса, соответствующие значению в запросе.

4. Для вставки в индекс нового города координаты города сначала при помощи функции преобразования преобразуются в набор ключей.

Затем ключи вставляются в соответствующие узлы индекса, начиная с корневого узла.

Если узел заполнен, выполняется операция разделения для создания двух новых узлов и ключи распределяются между узлами.

GiST имеет несколько недостатков, которые нужно учитывать:

1. Сниженная скорость вставок и обновлений: структуры индексирования GiST могут быть сложнее, чем традиционные структуры индексирования, что может привести к снижению скорости операций вставки и обновления.

2. Больше дискового пространства: структуры индексирования GiST могут требовать больше дискового пространства, чем другие методики индексирования, поскольку хранят дополнительную информацию для поддержки различных типов поиска.

3. Подходит не для всех типов данных: GiST оптимизирован под индексирование сложных типов данных, например, пространственных данных, однако может быть не лучшим выбором для индексирования более простых типов данных, например, целочисленных значений или строк.

4. Повышенные затраты на поддержку: из-за сложности реализации индексы GiST требуют больше обслуживания по сравнению с традиционными индексами.

Полнотекстовый индекс

Полнотекстовое индексирование — это методика индексирования баз данных, используемая для повышения эффективности поиска текстовых запросов. Это особый вид индекса, спроектированный для работы с текстовыми данными. В отличие от традиционных индексов, хранящих значения отдельных столбцов, полнотекстовый индекс хранит текстовое содержимое одного или нескольких столбцов как множества слов или токенов.

Эти слова или токены используются при выполнении поискового запроса для быстрого нахождения релевантных строк.

Полнотекстовое индексирование способно существенно улучшить производительность текстовых поисковых запросов, особенно при работе с большими объёмами текстовых данных. Однако оно требует дополнительного дискового пространства и вычислительных ресурсов, а также тщательной настройки параметров индексирования для обеспечения оптимальной производительности.

Процесс полнотекстового индексирования состоит из нескольких этапов:

1. Токенизация: текстовое содержимое индексируемого столбца разбивается на отдельные слова или токены, которые затем сохраняются в индекс. При создании полнотекстового индекса система базы данных сначала анализирует текстовое содержимое индексируемых столбцов, а затем разбивает его на отдельные слова или токены. Этот процесс называется токенизацией, он может включать в себя фильтрацию игнорируемых слов (например, «the», «and», «or») и выделение корней (редуцирование слов до их базовой формы).

2. Индексирование: затем токены индексируются при помощи специальной структуры данных, например, B-дерева или инвертированного индекса. Структура индекса обеспечивает возможность эффективного поиска и извлечения строк, содержащих указанные токены.

3. Построение и выполнение запросов: система базы данных использует полнотекстовый индекс для поиска строк, содержащих релевантные токены. В процессе поиска токены запроса сопоставляются с индексированными токенами и извлекаются строки, соответствующие запросу. Результаты поиска можно ранжировать на основании их релевантности запросу, который вычисляется при помощи алгоритмов наподобие TF-IDF (term frequency-inverse document frequency).

Полнотекстовое индексирование имеет некоторые недостатки:

1. Сниженная скорость индексирования и поиска: полнотекстовое индексирование может быть более сложным, чем другие техники индексирования, что может приводить к снижению скорости индексирования и поиска, особенно в больших базах данных со множеством текстовых полей.

2. Подходит не для всех типов данных: полнотекстовое индексирование лучше всего подходит для баз данных, содержащих большие объёмы текстовых данных. Оно может и не быть наиболее эффективной техникой для баз данных, по большей мере, для содержащих числовую или другую нетекстовую информацию.

3. Зависимость от языка: полнотекстовое индексирование может быть не очень эффективно для многоязычных баз данных, поскольку требует отдельных индексов для каждого языка и может оказаться неспособным справиться с нюансами различных языков и систем

Индексирование баз данных — критически важная технология, повышающая эффективность запросов к базам данных. Оно заключается в создании специальных структур данных, обеспечивающих эффективный поиск и извлечение данных на основании одного или нескольких столбцов таблицы. Для оптимизации запросов под различные типы данных и сценарии использования применяются разные типы алгоритмов индексирования, например, B-деревья, bitmap-индексы, хэш-индексы и GiST-индексы.

9. СТОИМОСТНЫЕ ОПТИМИЗАТОРЫ В СУБД

Эвристический подход

Для начала коротко рассмотрим эвристический подход. Здесь решение о выборе плана запроса принимается на основании некоторых заранее предложенных правил.

К примеру, таких:

- Выборка и проекции должны быть выполнены как можно раньше в дереве запроса

- При объединении таблиц необходимо в первую очередь выполнять наиболее строгие операции объединения

Важно, что эти правила основываются на общих идеях и теоретически не обязаны давать наиболее эффективные результаты. Но такой подход довольно прост в реализации и потому с завидной периодичностью используется даже в ведущих СУБД.

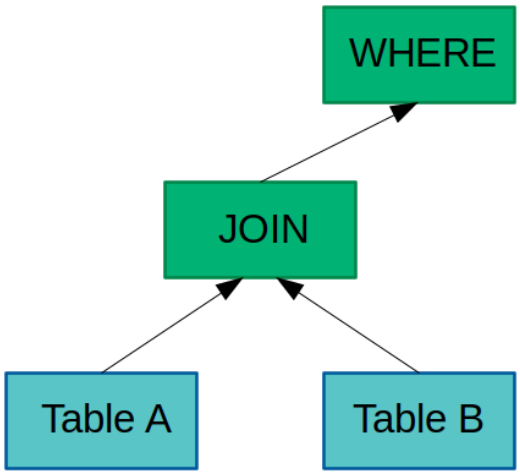
Стоимостной подход

Перейдем к более интересному, на мой взгляд, подходу — стоимостному. Его идея заключается в оценке эффективности плана выполнения на основании некоторого параметра. Этим параметром может быть время выполнения, количество физических обращений к диску, размеры промежуточных выборок из БД и т.д.

Первое, что делает при таком подходе оптимизатор - строит дерево запроса. Разберемся, что это такое.

Деревом запроса называется древовидная структура, в узлах которой расположены логические операторы, соответствующие отдельным операциям запроса.

Другими словами, нечто подобное:



К каждому такому дереву можно применить трансформацию — логическую или физическую. Логические трансформации порождают новые деревья посредством изменения структуры исходных, а физические заменяют логические операторы на их конкретные реализации, называемые физическими операторами, не меняя структуру дерева. Так, например, логический оператор JOIN можно заменить на физические LOOP JOIN или MERGE JOIN.

Трансформации производятся на основании некоторого набора правил, состоящих из шаблона и подстановки. Шаблон показывает, к какому выражению данное правило может быть применено, а подстановка — что будет получено в результате изменений.

Разобравшись с тем, что такое дерево поиска, введем ещё два определения: логическая эквивалентность и область поиска.

Логически эквивалентными называются деревья запроса, дающие одинаковый результат на выходе. Логически эквивалентные деревья запроса в совокупности с их физическими реализациями образуют группу эквивалентных запросов.

Чтобы сократить количество логически эквивалентных выражений, их можно преобразовывать в так называемые групповые выражения. В каждом узле дерева такого выражения находится логический оператор, принимающий на вход группы эквивалентных запросов, а не просто отдельные запросы.

Областью поиска называется множество всевозможных логически эквивалентных деревьев запроса. Исходной областью поиска является логическое дерево, полученное в результате парсинга искомого запроса. При замене каждого из узлов дерева на его логические эквиваленты область поиска расширяется. Конечной областью поиска будет таким образом являться множество всех эквивалентных искомому запросу деревьев поиска. А точнее, одно дерево, каждый из узлов которого будет являться полной эквивалентной группой аналогичного узла в первоначальном дереве. Согласен, звучит сложно, но теория — она такая.

Итак, с определениями разобрались, теперь перейдём непосредственно к оценке стоимости плана. Она состоит из трех этапов: оценки кардинальности, применения стоимостной модели и перечисления планов. Рассмотрим каждый в отдельности.

Этап 1. Оценка кардинальности

Для тех кто не знал или забыл, напомню, что кардинальностью называется размер выборки, возвращаемой некоторым выражением. Чаще всего под размером понимается количество полученных в результате запроса кортежей, но иногда речь может идти и о количестве страниц. Очевидно, что при наличии готовой выборки посчитать кардинальность не составит труда, но можно ли как-то провести оценку, не выполняя сам запрос?

Да, и для этого существуют разные подходы. Два наиболее популярных из них — оценка на основании выборочной совокупности и использование профилей.

Первый вариант заключается в анализе выборки «в миниатюре». При таком подходе, к примеру, соединяют два небольших набора кортежей вместо соединения двух полных таблиц. Полученная в результате оценка затем экстраполируется на полноценное соединение.

Во втором случае используется некоторая дополнительная информация о таблицах БД, к примеру, размер, количество уникальных атрибутов, наиболее часто встречающиеся атрибуты и т.д. На основании этих данных можно «прикинуть» размер результата того или иного запроса. Подобные оценки грубы, и для их уточнения часто прибегают к использованию гистограмм — информации о распределении данных одного домена в таблице. При построении гистограмм рассматриваемый домен разбивается на интервалы, и затем для каждого из них вычисляется количество попавших в него значений. Определять размеры интервалов тоже можно по-разному: строить их равными по диапазону или по количеству значений.

Саму оценку кардинальности проводят в таком случае при помощи другого свойства запросов — селективности. Селективностью называется

отношение количества строк, возвращаемых запросом, к общему количеству строк в таблице. То есть, если в таблице user содержатся записи о 100 людях, 25 из которых младше 50, а 75 старше, то селективность запроса SELECT * FROM user WHERE age > 50 будет равняться 75/100=0.75

Вернёмся к оценке кардинальности. Предположим, что таблица user имеет 1000 записей, и согласно гистограмме распределения количество пользователей с параметром age=20 равняется 100, а количество пользователей с параметром sex=“male“ равняется 500.

Тогда запрос SELECT * FROM user WHERE age=20 AND sex='male' будет оценен следующим образом: age=20 даст селективность 100/1000=0.1, а sex='male' соответственно 500/1000=0.5. Итоговая селективность составит 0.1*0.5=0.05. Кардинальность данного запроса в таком случае может быть оценена как 1000*0.05=50 записей.

Ну, а если данных о распределении значений атрибутов какого-то столбца нет, то можно прибегнуть к ряду допущений, например, предположить, что значения атрибутов распределены равномерно.

Этап 2. Применение стоимостной модели

Стоимостная модель задает отображение полученного дерева плана на некоторое число, которое будет обозначать его итоговую стоимость. В свою очередь стоимостью плана называется сумма стоимостей всех входящих в него операторов. Стоимость оператора же может зависеть от множества факторов. Часть из них определена заранее и зависит, к примеру, от оборудования, на котором развернута СУБД. А часть может вычисляться динамически — например, оценка кардинальности оператора, наполненность буферов или наличие свободных потоков для параллельного выполнения.

Этап 3. Перечисление планов

Перечислением планов называется выбор наиболее оптимальной последовательности выполнения соединений. Важно помнить про время построения плана при использовании слишком большого числа соединений

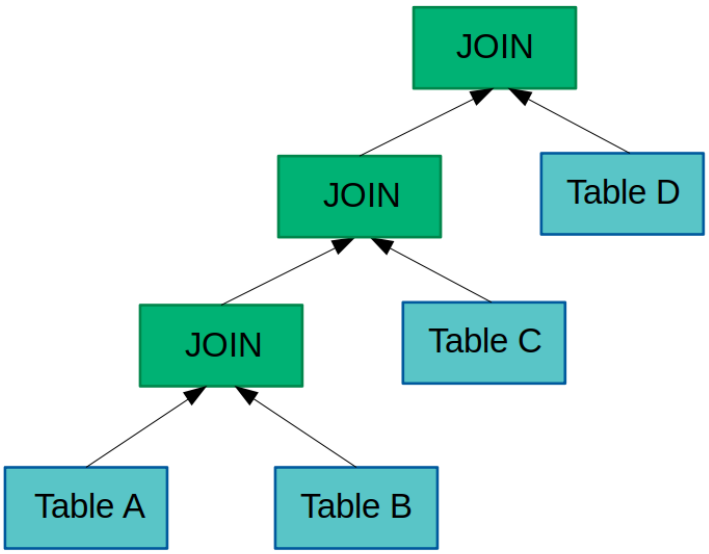
таблиц в одном запросе. Чаще всего эта проблема решается посредством применения ряда эвристик в ущерб оптимальности итогового плана.

Существует два наиболее часто применяемых алгоритма перечисления: прохождение по дереву плана снизу вверх при помощи динамического программирования и прохождение по дереву плана сверху вниз при помощи мемоизации.

Первый подход: снизу вверх

Первым алгоритмом, который мы рассмотрим, является прохождение по дереву снизу вверх при помощи динамического программирования. Здесь для вычисления оптимального плана на *i-м* уровне дерева вычисляются все планы на *i-1-м* уровне дерева, и из них выбирается наилучший. Очевидно, что при таком подходе выбор оптимального плана выродится в полный перебор всех вариантов, поэтому для использования на практике его нужно модифицировать. Так П.Г.Сэлинджер предложила рассматривать только левосторонние деревья, у которых при этом элементы на левой ветви имеют минимальную оценку кардинальности.

Примерно такие:



Чтобы понять, почему это работает, рассмотрим пример. Предположим, что у нас есть три таблицы B, C и D. Пусть A — таблица, полученная в результате соединения таблиц C и D.

Допустим, что некоторая операция contains возвращает true, если в таблице, переданной первым аргументом, есть кортеж, который можно естественным образом соединить с переданным во втором аргументе кортежем (совпадающие по именам атрибуты равны).

Попробуем сначала выполнить соединение A с B, а потом B с A. В первом случае процесс соединения можно описать следующим псевдокодом:

```
for a in A:
 if (contains(B, a)):
 return a x b
```

Если предположить, что метод contains для поиска искомого кортежа по очереди проверит каждую запись в таблице B, то приведенная операция очевидно будет иметь квадратичную сложность. Но поскольку B постоянно присутствует в базе данных, для нее можно построить индекс, что позволит быстрее выполнить поиск, и асимптотическая сложность задачи понизится.

Во втором случае процесс можно представить следующим (очень отличающимся) псевдокодом:

```
for b in B:
 if (contains(A, b)):
 return b x a
```

Казалось бы, рассуждения здесь должны быть аналогичными, но это не так. Поскольку таблица A является промежуточным результатом вычисления, она не будет содержать заранее определенных индексов, а это в свою очередь означает, что сложность операции в таком случае будет всегда квадратичной вне зависимости от наличия индексов в таблице *B*. Значит в теории такие деревья вообще не стоит рассматривать — зачем, если вариант заведомо неэффективный?

Второй подход: сверху вниз

Другой подход, называющийся мемоизацией, заключается в построении дерева, узлами которого являются группы операций, и построение с их помощью итогового результата. Для этого каждому оператору в группе присваивается оценка стоимости и на каждом уровне выбирается один «победитель» — наиболее оптимальный оператор из своей группы. Набор групп, рассматриваемых в процессе отбора оптимального результата, называется MEMO. Во многих современных оптимизаторах результаты выбора оптимального варианта для каждого узла запоминаются, чтобы уменьшить пространство поиска в дальнейшем.

10.ЛАБОРАТОРНЫЕ ЗАНЯТИЯ

Примерный перечень тем

1. Создание и использование массивов и связанных списков в python
2. Создание и использование хэш-таблиц в python
3. Работа с балансированными деревьями в python
4. Работа с LSM деревьями в python
5. Знакомство с базами данных в python
6. Работа с базовыми СУБД типа ключ-значение в python
7. Реализация алгоритмов слияния в python
8. Работа с индексами в python
9. Итоговая лекция, обсуждение всего изученного, ответы на вопросы

11.КОНТРОЛЬНАЯ РАБОТА

Примерный перечень тем

1. Алгоритмы и структуры данных

Примерные задания

1. Имеется некоторая структура данных, в которую заносятся упорядоченные по убыванию символы. Считывание данных из этой структуры даёт результат: F, E, D, C, B, A. Чем является эта структура данных?

— Связный список
— Дерево
— Граф
— Очередь
— Стек

2. Имеется упорядоченный массив целых чисел из 9 элементов. Сколько операций сравнения потребуется при двоичном поиске для нахождения искомого ключа, если он находится в точно в середине массива?

— 0
— 9
— 1
— 5
— 8

3. Имеется двоичное дерево (не являющееся деревом поиска), содержащее целые числа. Восходящий просмотр дерева даёт следующий результат: 2, 4, 6, 8, 10, 12, 14. Какой узел является корнем дерева?

— 14
— 2
— 8
— 10
— 6

4. Какая сортировка из следующих является самой неэффективной?

— Пузырьковая
— Отбором
— Шелла
— Вставками
— Быстрая

5. Имеется неупорядоченный массив целых чисел. Для нахождения ключа используется двоичный поиск. Гарантируется ли в этом случае истинность результата поиска?

— Да
— Гарантируется при условии, что значение ключа не превышает размера массива
— Гарантируется при условии, что в процедуре поиска используется цикл for

— Нет

— Гарантируется при условии, что в процедуре поиска используется цикл while

6. Какие операции над элементами характерны для очередей и стеков?

— Поиск элемента и сортировка

— Занесение элемента, извлечение элемента и просмотр

— Занесение элемента и извлечение элемента

— Занесение элемента, извлечение элемента, просмотр, сортировка и удаление текущего элемента

— Занесение элемента, извлечение элемента и очистка

7. Какая структура данных используется для сохранения и восстановления содержимого регистров общего назначения центрального процессора при вызове процедур?

— Стек

— Двоичное дерево

— Список

— Очередь

— Таблица

12.ДОМАШНЯЯ РАБОТА

Примерный перечень тем

1. Разработка базы данных для хранения и извлечения информации о книгах

Примерные задания

Практический проект на Python, который студенты могут создать в рамках данного модуля, может быть связан с разработкой базы данных для хранения и извлечения информации о книгах. Студенты могут использовать алгоритмы и структуры данных, изученные в данном модуле, для реализации индексов, таких как хэш-индекс и LSM-дерево, а также базовых СУБД типа “ключ-значение”, таких как Redis и MongoDB.

Студенты могут разработать веб-приложение, которое позволит пользователям добавлять книги в базу данных, искать их по ключевым словам

и категориям, а также просматривать информацию о каждой книге, включая ее название, автора, жанр и количество страниц.

Цель работы:

В результате такого проекта студенты приобретут практические навыки работы с базами данных и СУБД, научатся использовать различные алгоритмы и структуры данных для эффективного хранения и извлечения данных, а также улучшат свои навыки программирования на Python.

Задание и требования:

Студенты могут реализовать алгоритмы объединения множеств, такие как сортированное слияние и объединение по хэшу, для оптимизации поиска и извлечения информации из базы данных. Они могут также разработать систему кэширования для ускорения доступа к данным, используя LRU-кэш или другие алгоритмы.

Работа должна быть оформлена в виде python скрипта и отправлена в форму для приема работы. Задание индивидуально

13.ВОПРОСЫ ДЛЯ ЗАЧЕТА

1. Различия, составляющие реляционные СУБД: компоненты и процесс обработки запроса, основные этапы. Основные элементы оптимизатора.

2. Применение операций выборки и проекции. Различные подходы к осуществлению операции соединения. Исполнение операции агрегации.

3. Понятие переписывания запроса. Основные типы преобразований. Понимание оптимизации запросов. Примеры классических систем в оптимизации запросов. Понятие канонического представления и его пример. Логическая и физическая алгебра. Трансформационный оптимизатор.

4. Задачи оптимизатора. Операторные деревья: линейные, лево-линейные, и кустистые деревья. Граф соединений и его смысл. Алгоритм построения кустистых деревьев. Оптимизация сложных запросов.

5. История гистограмм. Значение использования гистограмм в СУБД. Определение и примеры гистограмм. Основные понятия гистограммы: спред,

площадь и частота. Гистограммы с постоянной шириной. Примеры серийных гистограмм. Экспериментальное исследование различных типов гистограмм.

6. Классификация гистограмм по параметру источника и параметру сортировки. Гистограммы равной глубины: алгоритм построения и примеры. Класс серийных гистограмм: определение, описание v-optimal и spline-based.

7. Распределенные СУБД: история и цель. Прозрачность в распределенных СУБД. Фрагментация и репликация в распределенных СУБД. Аспекты распределенных СУБД: автономность, распределенность, гетерогенность.

8. Основные типы распределенных СУБД: клиент-серверные, P2P СУБД, гетерогенные мультибазы.

9. Оптимизация в общем для распределенных СУБД. Два типа стоимостных моделей для оптимизации запросов в распределенных средах, с примером. Выполнение запросов в распределенных СУБД.

10. Выполнение запросов в клиент-серверных распределенных СУБД, применяемые стратегии. Оптимизация в клиент-серверной системе. Двухэтапный подход. Выполнение запросов в гетерогенных распределенных СУБД. Статистическая информация в гетерогенных системах.

11. Представление данных на диске: слотированная страница. История колоночных систем. Два фактора, стимулирующие популярность колоночного подхода. Предшественники колоночных систем. Два подхода к построению колоночных систем. Общая архитектура колоночных систем.

12. Сжатие в колоночных системах: алгоритмы и преимущества легковесных схем. Операции над сжатыми данными. Отложенная и прямая материализация в агрегированных запросах без соединений.

13. OLAP и OLTP: схемы "звезда" и "снежинка". Выполнение соединений с помощью стратегий отложенной и прямой материализации. Проблемы прямой материализации и доступа к диску. Невидимое соединение: алгоритм и преимущества.

14. Критика реляционной модели. Объектно-ориентированные БД и объектно-реляционные БД. Структура данных в ООБД и ОРБД. Запросы в ООБД и ОРБД. Архитектура объектных систем и основные вопросы. Особенности оптимизации в объектных системах.

15. Объектные серверы и страничные серверы: факты и рассуждения. Управление буфером. Управление OID: стратегии POID и LOID. Типы LOID, генерация LOID, перевод LOID в POID. Хардварное и программное Pointer Swizzling. Понятие выражения пути и соображения по оптимизации.

16. Задача настройки СУБД. Автоматическая настройка СУБД: история и актуальность. Подходы к настройке физического уровня. Формальная постановка задачи и ее три компонента. Решение и классификация методов решения. Процедурные и стоимостные методы.

17. Горизонтальное фрагментирование данных.

18. Для чего применяется кластеризация записей.

19. Что такое B-дерево и B+-дерево. Многомерное индексирование.

20. Задача расщепления в R-дереве.

21. KD-дерево. Построение KD-дерева.

22. Графы в базах данных.

14.УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ

Электронные ресурсы (издания)

1. Алексеев, В. Е. Графы и алгоритмы. Структуры данных. Модели вычислений / В.Е. Алексеев, В.А. Таланов. - М.: Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2018. - 320 с.

2. Бабенко, М. А. Введение в теорию алгоритмов и структур данных / М.А. Бабенко. - М.: МЦНМО, 2016. - 243 с.

3. Белов, В. В. Алгоритмы и структуры данных. Учебник / В.В. Белов, В.И. Чистякова. - М.: КУРС, Инфра-М, 2016. - 240 с.

4. Вирт Алгоритмы и структуры данных / Вирт, Никлаус. - М.: СПб: Невский Диалект; Издание 2-е, испр., 2018. - 352 с.

5. Роберт, Седжвик Алгоритмы на C++. Анализ структуры данных. Сортировка. Поиск. Алгоритмы на графах. Руководство / Седжвик Роберт. - М.: Диалектика / Вильямс, 2016. - 697 с.

6. Чиняков, Н. А.; Реляционные базы данных для социальных исследований на примере лаборатории позитивной психологии ВШЭ: выпускная бакалаврская работа : студенческая научная работа.; , Москва; 2018; https://biblioclub.ru/index.php?page=book&id=491998 (Электронное издание)

7. Сидорова, Н. П.; Базы данных: практикум по проектированию реляционных баз данных : учебное пособие.; Директ-Медиа, Москва, Берлин; 2020; https://biblioclub.ru/index.php?page=book&id=575080 (Электронное издание)

8. Сидорова, Н. П.; Информационное обеспечение и базы данных: практикум по дисциплине «Информационное обеспечение, базы данных» : учебное пособие.; Директ-Медиа, Москва, Берлин; 2019; https://biblioclub.ru/index.php?page=book&id=500238 (Электронное издание)

9. Жуков, Р. А.; Базы данных: учебно-методическое пособие по дисциплине «Базы данных» для направления подготовки 38.03.05 «Бизнес-информатика» (бакалавриат) : учебно-методическое пособие.; Директ-Медиа, Москва, Берлин; 2019; https://biblioclub.ru/index.php?page=book&id=566814 (Электронное издание)

10. Гудов, А. М.; Базы данных и системы управления базами данных. Программирование на языке PL/SQL : учебное пособие.; Кемеровский государственный университет, Кемерово; 2010; https://biblioclub.ru/index.php?page=book&id=232497 (Электронное издание)

11. Дьяков, И. А.; Базы данных. Язык SQL : учебное пособие.; Тамбовский государственный технический университет (ТГТУ), Тамбов;

2012; https://biblioclub.ru/index.php?page=book&id=277628 (Электронное издание)

12. Королева, , О. Н., Мажукин, , В. И.; Базы данных : курс лекций.; Московский гуманитарный университет, Москва; 2012; http://www.iprbookshop.ru/14515.html (Электронное издание)**Профессиональные базы данных, информационно-справочные системы**

1. Единое окно доступа к образовательным ресурсам. Раздел Информатика и информационные технологии http://window.edu.ru/catalog

2. Интернет-Университет Информационных Технологий http://www.intuit.ru/

3. Веб-сервис для хостинга IT-проектов и их совместной разработки Github http://www.github.ru

Материалы для лиц с ОВЗ

Весь контент ЭБС представлен в виде файлов специального формата для воспроизведения синтезатором речи, а также в тестовом виде, пригодном для прочтения с использованием экранной лупы и настройкой контрастности.

Базы данных, информационно-справочные и поисковые системы

1. ЭБС Университетская библиотека онлайн «Директ-Медиа» http://www.biblioclub.ru/

2. eLibrary ООО Научная электронная библиотека http://elibrary.ru