Формализация информации и Big Data

«02.03.03 - Математическое обеспечение и администрирование информационных систем направленность разработка и администрирование информационных систем»

http//vikchas.ru

Тема 1.Формализация информации

Лекция 7«Алгоритмы работы современных систем хранения информации»

Часовских Виктор Петрович
д-р техн. наук, профессор кафедры ШИиКМ

ФГБОУ ВО «Уральский государственный экономический университет»

Екатеринбург 2023

ОБЩИЕ ПОЛОЖЕНИЯ

Распределенные системы управления базами данных являются неотъемлемой частью большинства предприятий и подавляющего большинства программных приложений. Приложения отвечают за логику и взаимодействие с пользователем, в то время как системы управления базами данных (СУБД) обеспечивают целостность, согласованность и избыточность данных.

В 2000 году было лишь несколько разновидностей баз данных, большинство из которых были реляционными и в силу этого мало чем отличались друг от друга. Конечно, это не означает, что эти базы данных были совершенно одинаковыми, однако они имели много сходства в плане функциональности и сценариев использования.

Некоторые из этих баз данных ориентировались на ***горизонтальное масштабирование*** — повышение производительности и увеличение емкости за счет запуска нескольких экземпляров базы данных, действующих как единый логический блок: Gamma Database Machine Project, Teradata, Greenplum и многие другие.

Горизонтальное масштабирование и сегодня остается одним из самых востребованных свойств баз данных, что объясняется растущей популярностью облачных сервисов. Зачастую проще развернуть новый экземпляр и добавить его в кластер, чем производить ***вертикальное масштабирование,*** перенося базу данных на более крупную и мощную машину. Миграция часто требует больших затрат времени и усилий, что, в свою очередь, может приводить к простоям.

Примерно в 2010 году начал формироваться новый класс баз данных *с* ***конечной согласованностью*** и стали набирать популярность такие термины, как *NoSQL* и ***большие данные***. За последние 15 лет сообщество разработчиков ПО с открытым исходным кодом, крупные интернет-компании и поставщики баз данных создали так много баз данных и инструментов, что можно легко запутаться, пытаясь разобраться в их специфических особенностях и сценариях использования.

В 2007 году разработчики компании Amazon опубликовали статью с описанием системы хранения данных Dynamo, которая произвела столь сильное впечатление на сообщество разработчиков баз данных, что за короткое время на свет появилось множество вариантов и реализаций этой системы. Наиболее известными среди них были такие системы хранения, как Apache Cassandra, разработанная компанией Facebook, Project Voldemort от компании LinkedIn и Riak от бывших разработчиков компании Akamai.

Сегодня ситуация в области систем хранения данных снова меняется: на смену хранилищам типа «**ключ–значение**», NoSQL и системам с конечной согласованностью стали приходить более масштабируемые и производительные базы данных, способные выполнять сложные поисковые запросы с более высокой гарантией согласованности.

Рассмотрим алгоритмы и концепции, используемые во всех типах СУБД, с акцентом на ***подсистему хранения данных*** и на компонентах, отвечающих за распределение.

Наиболее существенные различия между разными СУБД касаются используемых **методов хранения** и **методов распределения данных.**

Основной задачей любой СУБД является надежное хранение данных и обеспечение доступа к ним со стороны пользователей.

Мы используем базы данных в качестве основного источника данных, помогающего обмениваться данными между различными частями приложений.

Подсистема хранения данных

База данных представляет собой модульную систему, включающую в себя несколько составных частей: **транспортный уровень**, который принимает запросы, **обработчик запросов**, который выбирает наиболее эффективный способ выполнения запросов, **подсистему выполнения**, которая производит выполнение операций, и **подсистему хранения**.

Подсистема хранения данных (или ядро СУБД) — это программный компонент СУБД, отвечающий за хранение, извлечение и управление данными в памяти и на диске, предназначенный для работы с постоянной, долговременной памятью каждого узла.

Существует множество различных подсистем хранения, использующих разные структуры данных и реализованных на разных языках, начиная с таких низкоуровневых языков, как C, и заканчивая такими высокоуровневыми языками, как Java. В то же время все подсистемы хранения сталкиваются с одинаковыми проблемами и ограничениями.

СУБД разделяют на три основные категории:

Базы данных для обработки транзакций в реальном времени (online transaction processing, OLTP)

Они обрабатывают большое количество поступающих со стороны пользователя запросов и транзакций. Запросы часто бывают предопределенными и с коротким жизненным циклом.

Базы данных для аналитической обработки данных в реальном времени (online analytical processing, OLAP)

Они обрабатывают сложные агрегаты данных. OLAP-системы часто используются для аналитики и построения хранилищ данных и способны обрабатывать сложные произвольные специальные запросы с длительным жизненным.

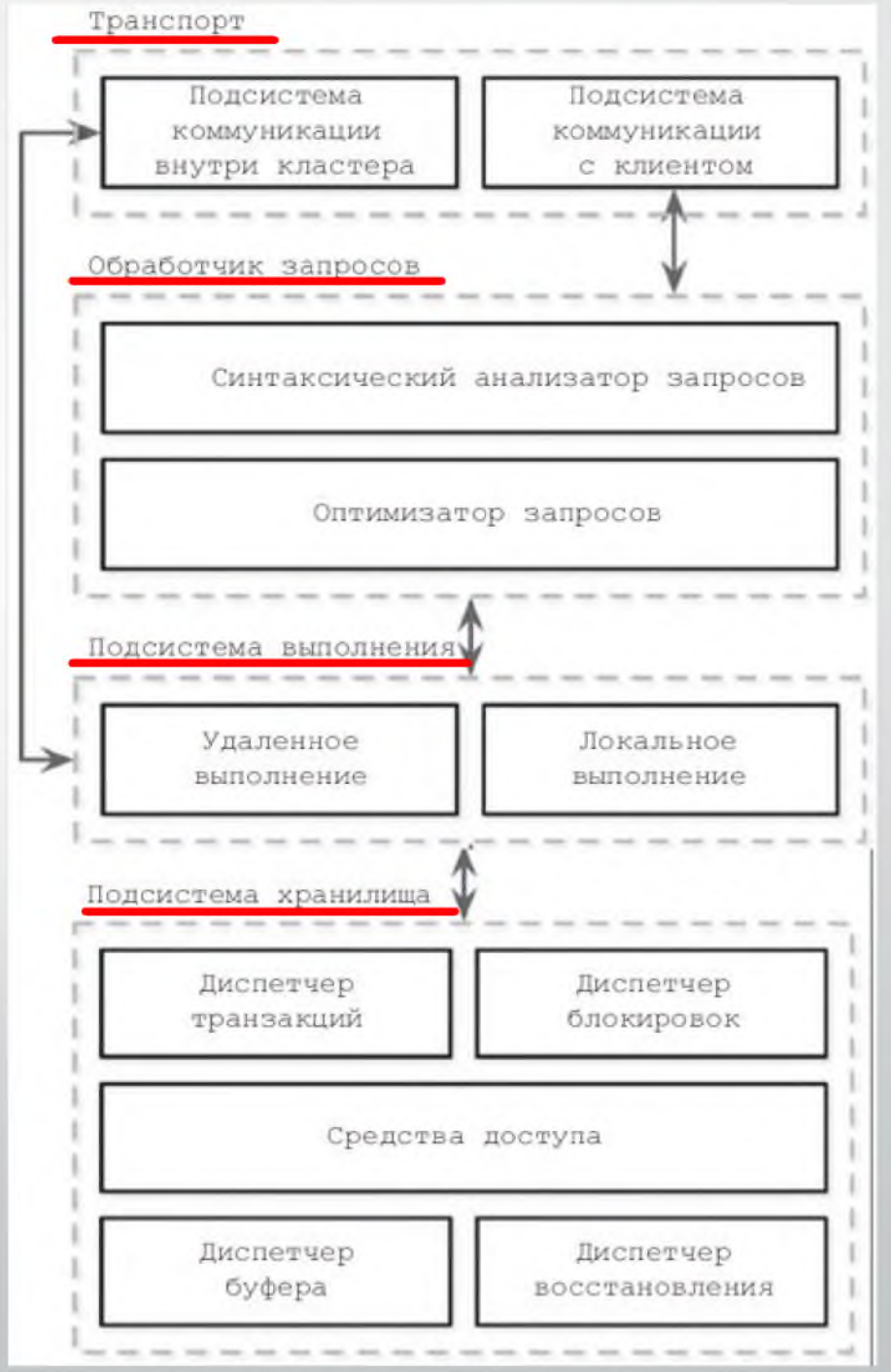
Гибридная транзакционно-аналитическая обработка (hybrid transactional and analytical processing, HTAP)

Эти базы данных сочетают в себе свойства хранилищ OLTP и OLAP.

Архитектура СУБД

Не существует общего шаблона для проектирования СУБД.

Каждая база данных строится немного по-разному, а границы компонентов довольно трудно обнаружить и определить. Даже если эти границы существуют на бумаге (например, в проектной документации), в коде кажущиеся независимыми компоненты могут оказаться связанными из-за оптимизации производительности, обработки граничных случаев или применения определенных архитектурных решений.



СУБД используют ***модель «клиент — сервер»,*** в которой экземпляры СУБД (*узлы*) играют роль серверов, а экземпляры приложений — роль клиентов.

Запросы клиентов поступают через ***транспортную подсистему***. Поступающие запросы чаще всего оформлены на каком-либо языке запросов. Транспортная подсистема также отвечает за связь с другими узлами кластера баз данных.

После получения запроса транспортная подсистема передает его обработчику запросов, который анализирует, интерпретирует и проверяет его. Далее проводятся проверки управления доступом, так как для их полного выполнения необходима интерпретация запроса.

Анализируемый запрос передается оптимизатору запросов, который сначала устраняет невозможные и избыточные части запроса, а затем пытается найти наиболее эффективный способ его выполнения на основе внутренней статистики (мощность индекса, приблизительный размер пересечений и т. д.) и размещения данных (какие узлы в кластере содержат данные и какие затраты требуются для их передачи).

Оптимизатор обрабатывает реляционные операции, необходимые для разрешения запросов, обычно представленные в виде дерева зависимостей, и проводит оптимизации, такие как упорядочение индексов, оценка мощности и выбор средств доступа.

Запрос обычно представлен в виде плана выполнения (или плана запроса): последовательности операций, которую необходимо выполнить для того, чтобы результаты запроса считались полными. Поскольку один и тот же запрос может быть выполнен с помощью различных планов выполнения, которые могут отличаться по эффективности, оптимизатор выбирает наиболее эффективный план из всех доступных.

План выполнения обрабатывается подсистемой выполнения, которая собирает результаты выполнения локальных и удаленных операций. Удаленное выполнение обычно сводится к записи и чтению данных на других узлах кластера, а также выполнению репликации.

Локальные запросы (поступающие непосредственно от клиентов или от других узлов) выполняются подсистемой хранения данных, которая включает в себя несколько компонентов с четко заданными функциями:

Диспетчер транзакций

Производит планировку транзакций и гарантирует, что они не оставят базу данных в логически несогласованном состоянии.

Диспетчер блокировок

Блокирует объекты базы данных для выполняемых транзакций, чтобы конкурентные операции не нарушали физическую целостность данных.

Средства доступа (структуры для хранения данных)

Управляют доступом и организацией данных на диске. Средства доступа включают в себя неупорядоченные файлы и такие структуры хранения, как B-деревья или LSM-деревья.

Диспетчер буферов

Кэширует страницы данных в памяти.

Диспетчер восстановления

Ведет журнал операций и восстанавливает состояние системы в случае сбоя.

Резидентные и дисковые СУБД

СУБД хранят данные в оперативной памяти или на диске.

Резидентные СУБД (или *СУБД в оперативной памяти*) хранят данные *главным образом* в оперативной памяти, а диск используют для восстановления и ведения журнала.

Дисковые СУБД хранят *большую часть* данных на диске и используют память для кэширования содержимого диска или в качестве временного хранилища.

Системы обоих типов в той или иной степени используют диск, но резидентные базы данных хранят содержимое почти исключительно в ОЗУ

Доступ к памяти был и остается на несколько порядков быстрее доступа к диску, поэтому есть смысл использовать память в качестве основного хранилища. Такой подход становится все более экономически целесообразным, поскольку цены на память снижаются. Однако цены на оперативную память по-прежнему остаются высокими по сравнению с постоянными устройствами хранения данных, такими как твердотельные накопители и жесткие диски.

Резидентные СУБД отличаются от дисковых не только основной средой хранения данных, но и тем, как они организованы и какие структуры данных и методы оптимизации они используют.

Использование памяти в качестве основного хранилища данных в таких базах данных обусловлено главным образом высокой производительностью, сравнительно низкими затратами на доступ и высокой гранулярностью доступа.

Основными ограничивающими факторами роста количества резидентных баз данных являются непостоянство (т. е. недостаточная долговечность) оперативной памяти и высокая стоимость.

Колоночные и строчные СУБД

Большинство СУБД хранит некоторый *набор записей*, состоящий из *столбцов* и *строк таблиц*.

Поле находится на пересечении столбца и строки и содержит одно значение некоторого типа.

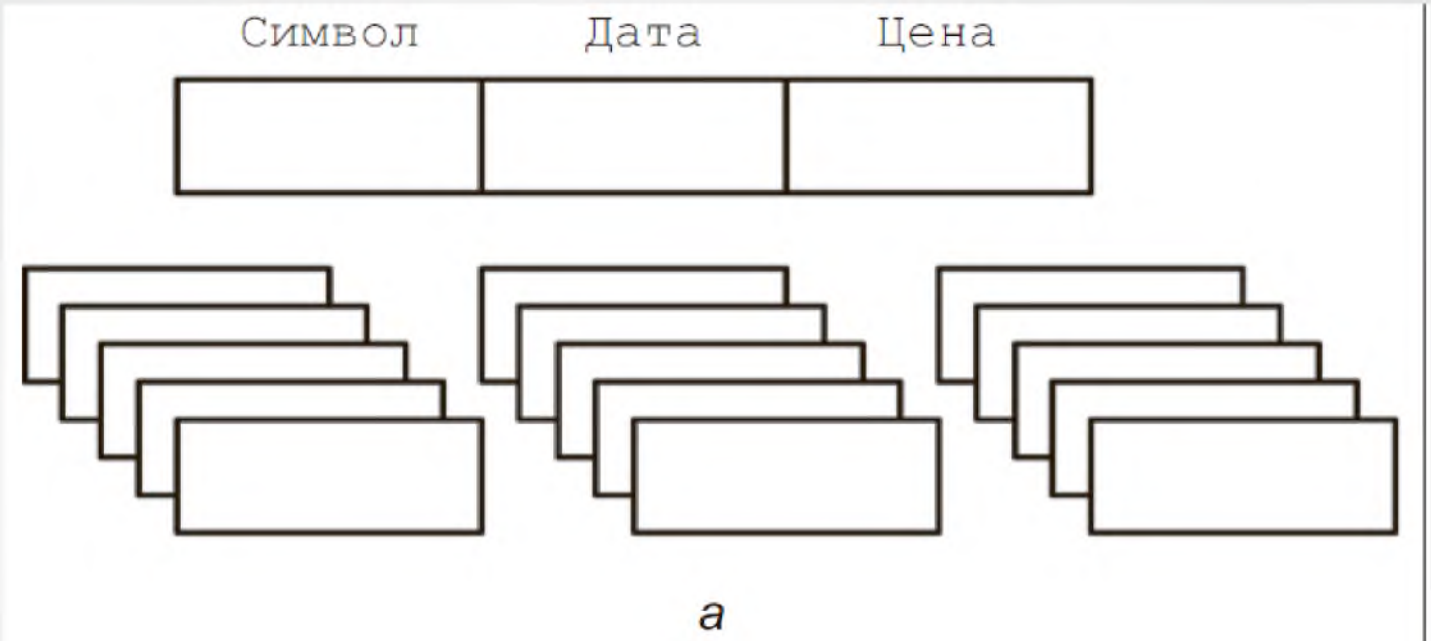
Поля, относящиеся к одному столбцу, обычно имеют один и тот же тип данных. Например, если мы определим таблицу, содержащую записи пользователей, все имена будут иметь один и тот же тип и находиться в одном и том же столбце. Набор значений, логически относимых к одной и той же записи (обычно идентифицируемой ключом), образует строку.

Один из способов классификации баз данных сводится к их разделению в зависимости от того, как данные сохраняются на диске: по строкам или по столбцам.

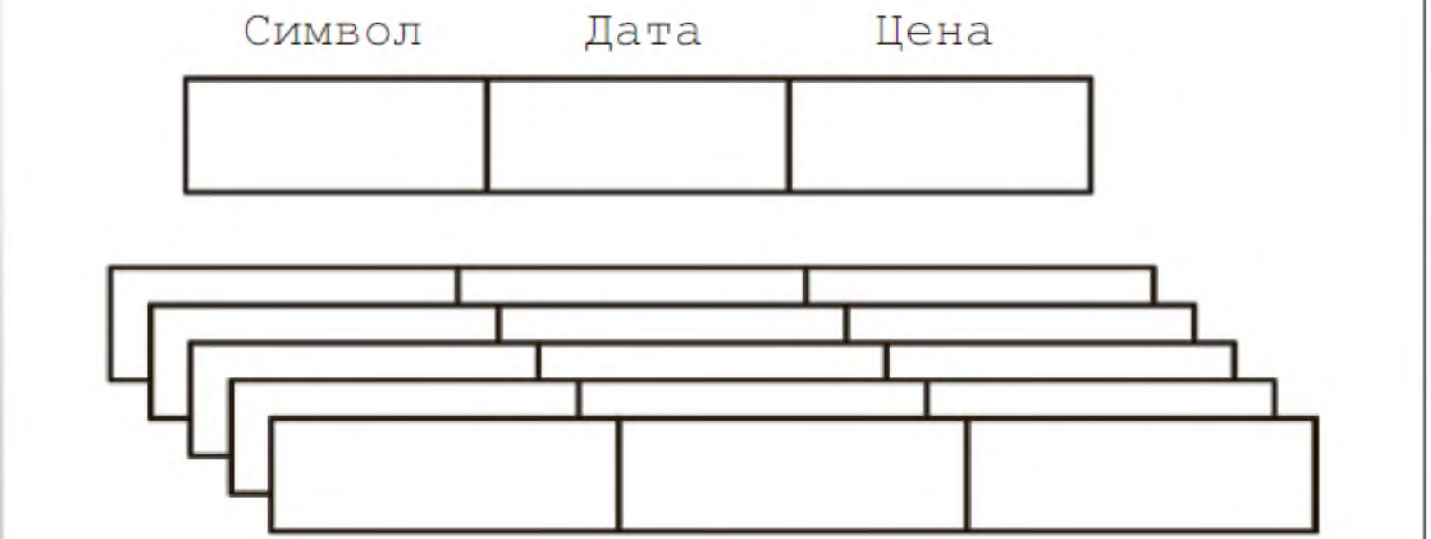
Данные таблиц могут разбиваться либо по горизонтали (когда вместе сохраняются значения, относящиеся к одной строке), либо по вертикали (когда вместе сохраняются значения, относящиеся к одному столбцу

Компоновка данных в колоночных(а) и строчных хранилищах(б)

а - SAP IQ, Google Bigtable, Vertica



а



б

Введение в B-деревья

Одной из самых популярных структур хранения данных является **B-дерево**.

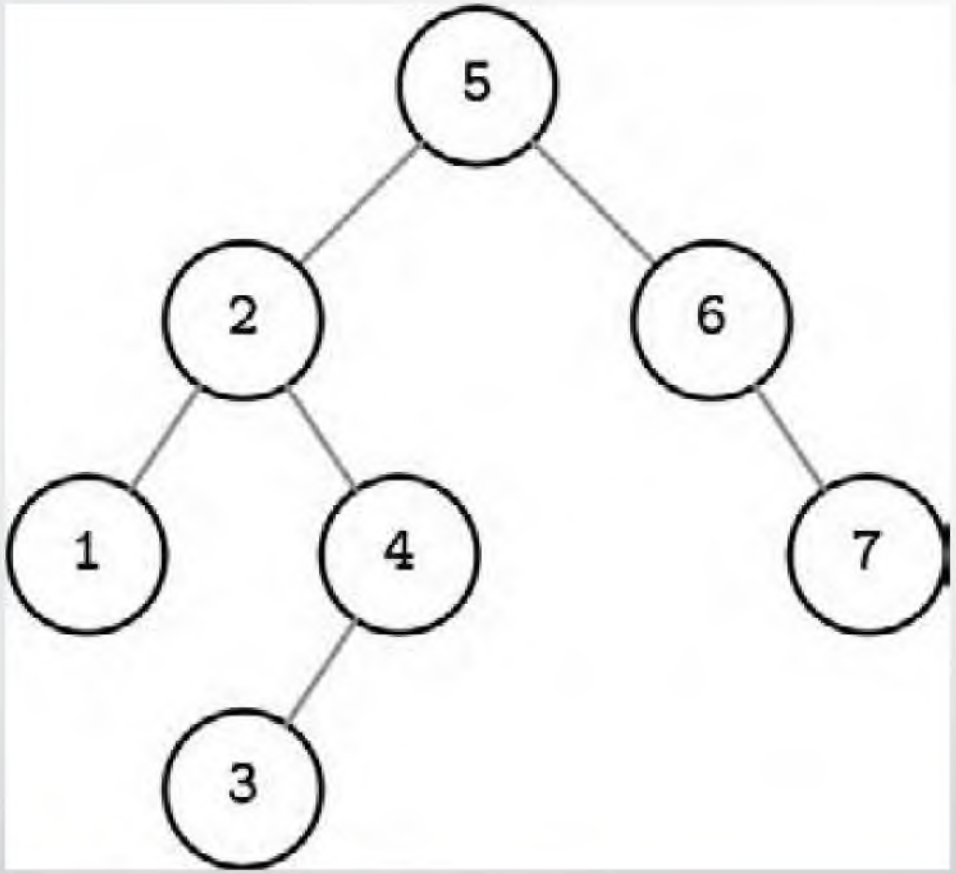
Многие СУБД с открытым исходным кодом основаны на B-деревьях, и за прошедшие годы они доказали, что охватывают большинство сценариев использования.

B-деревья не являются недавним изобретением: они были введены Рудольфом Байером и Эдвардом М. Маккрейтом еще в 1971 году и с годами приобрели популярность. К 1979 году существовало уже довольно много вариантов B-деревьев.

Двоичные деревья поиска

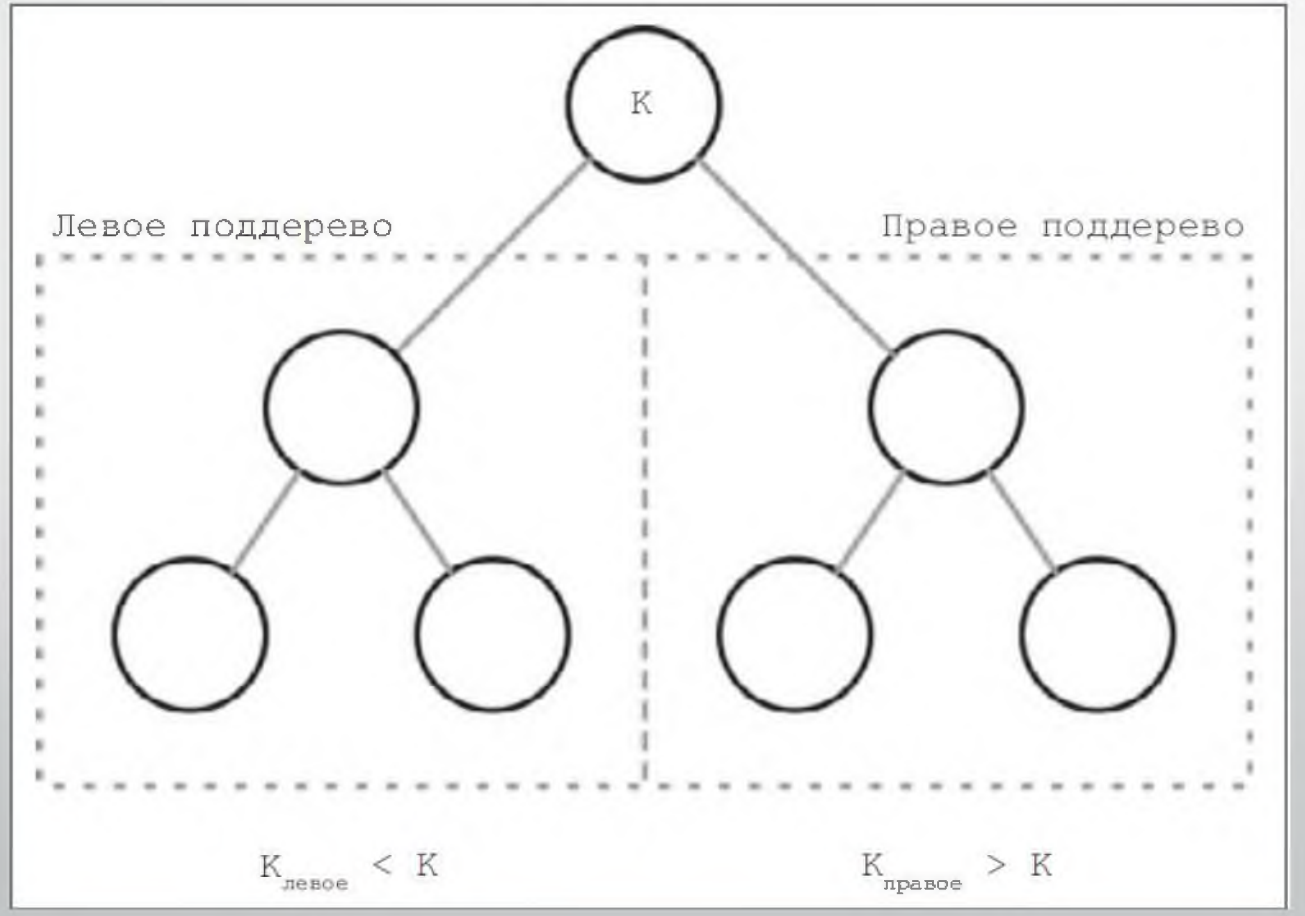
Двоичное дерево поиска — это упорядоченная структура данных в **оперативной памяти**, используемая для эффективного поиска значений по ключу. Двоичные деревья состоят из нескольких узлов. Каждый узел дерева представлен ключом, значением, связанным с этим ключом, и двумя указателями на потомков (отсюда и название — двоичное).

Двоичные деревья начинаются с одного узла, называемого корневым. При этом у дерева может быть только один корень.



Каждый узел разделяет пространство поиска на левое и правое поддеревья.

При этом ключ узла *больше* любого ключа, хранящегося в его левом поддереве, и *меньше* любого ключа, хранящегося в его правом поддереве



Следуя по левым указателям от корня дерева вниз до листового уровня (где узлы уже не имеют потомков), можно найти узел, содержащий наименьший ключ в дереве и связанное с ним значение.

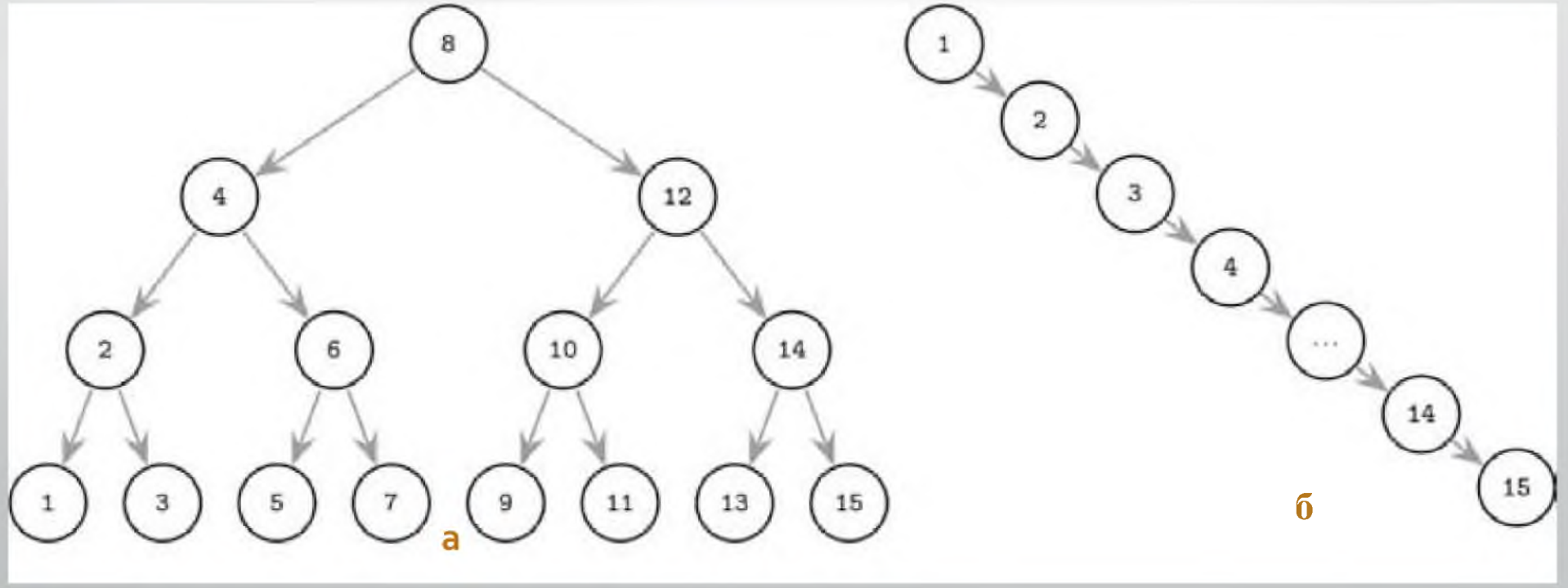
Аналогичным образом, следуя по правым указателям, можно найти узел, содержащий самый большой ключ в дереве и связанное с ним значение. Значения можно хранить в любых узлах дерева.

Поиск начинается с корневого узла и может завершиться до достижения нижнего уровня дерева, если искомый ключ будет найден на более высоком уровне.

Балансировка деревьев

Операции вставки не следуют какой-либо определенной схеме, поэтому вставка элементов может привести к тому, что дерево станет несбалансированным (т. е. одна из его ветвей будет длиннее другой).

В наихудшем случае мы можем в итоге получить «вырожденное» (или «патологическое») дерево (б), которое больше похоже на связанный список и вместо желаемой логарифмической сложности (а) дает нам линейную.



Дерево должно быть сбалансировано: даже если и маловероятно, что все элементы окажутся на одной стороне дерева, то по крайней мере некоторые из них обязательно окажутся там, что значительно замедлит поиск.

Сбалансированное дерево определяется как дерево, имеющее высоту log2N, где N — общее число элементов в дереве, а разница в высоте между двумя поддеревьями не превышает единицы.

Без балансировки теряется преимущества производительности структуры двоичного дерева поиска и позволяем операциям вставки и удаления определять форму дерева.

В сбалансированном дереве следование указателю на левый или правый узел сокращает пространство поиска в среднем вдвое, поэтому сложность поиска логарифмическая: **O($\log_2 N$).**

Использование заглавной буквы **O** (или так называемая O-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

Примеры

O(n) — линейная сложность.

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

O(log n) — логарифмическая сложность.

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

O(n2) — квадратичная сложность.

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n * n, т. е. n2.

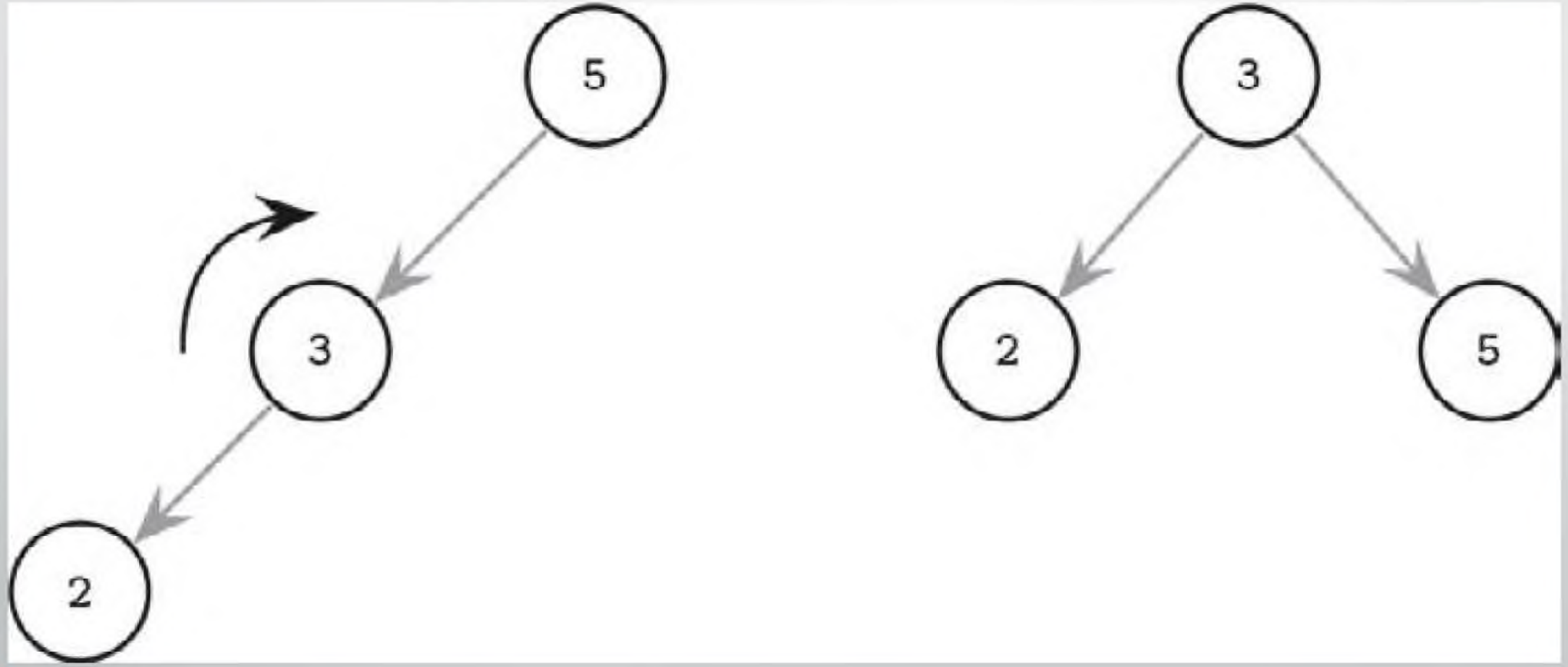
Если дерево не сбалансировано, в худшем случае сложность возрастает до O(N), так как мы можем оказаться в ситуации, когда все элементы расположатся на одной стороне дерева.

Вместо того чтобы добавлять новые элементы в одну из ветвей дерева и делать ее длиннее, в то время как другая остается пустой (как показано в примере слайда 20 (б)), дерево балансируется после каждой операции. Балансировка выполняется путем реорганизации узлов таким образом, чтобы минимизировать высоту дерева и сохранить количество узлов на каждой стороне в конкретных пределах.

Один из способов сохранения сбалансированности дерева сводится к выполнению операции поворота после добавления или удаления узлов.

Если операция вставки оставляет ветвь несбалансированной (два последовательных узла в ветви имеют только одного потомка), мы можем повернуть узлы вокруг среднего узла. В примере, слайд 24, во время поворота средний узел (3) выступает в качестве оси вращения и смещается на один уровень выше; при этом его родитель становится его правым потомком.

Операция поворота



Дисковые структуры

Ранее мы в общих чертах обсудили резидентные и дисковые хранилища. Такое же различие можно провести и для конкретных структур данных: некоторые из них лучше подходят для использования на диске, а некоторые лучше работают в оперативной памяти.

Не каждая структура данных, удовлетворяющая требованиям к занимаемому пространству и сложности, может быть эффективно использована для хранения на диске. Структуры данных, используемые в базах данных, должны быть адаптированы с учетом ограничений персистентных носителей (*структуры данных, которые при внесении в них каких-то изменений сохраняют все свои предыдущие состояния и доступ к этим состояниям*).

Дисковые структуры данных часто используются, когда объемы информации настолько велики, что хранение всего набора данных в памяти невозможно или непрактично. В каждый момент может быть кэширована только часть данных, а остальные должны храниться на диске таким образом, чтобы к ним можно было эффективным образом обращаться.

Жесткие диски

Большинство традиционных алгоритмов были разработаны в то время, когда наиболее распространенным персистентным носителем данных были вращающиеся диски, что существенно повлияло на их свойства. Позже новые разработки в области носителей данных, такие как флэш-накопители, вдохновили новые алгоритмы и модификации существующих, использующие возможности нового оборудования. В наши дни появляются новые типы структур данных, оптимизированные для работы с энергонезависимыми хранилищами с байтовой адресацией.

Вращающиеся диски повышают затраты на случайное считывание, поскольку они требуют вращения диска и механического движения головки для размещения головки чтения/записи в нужном месте. Однако после выполнения этой затратной части операции чтение или запись непрерывной последовательности байтов требует сравнительно небольших затрат.

Наименьшей единицей передаваемой информации у вращающегося накопителя является сектор, поэтому при выполнении любой операции нужно прочитать или записать как минимум один целый сектор. Размеры секторов обычно варьируют от 512 байт до 4 КБ.

Позиционирование головки — самая затратная часть операции, выполняемой на жестком диске. Это одна из причин, по которой мы часто слышим о положительных эффектах *последовательного ввода-вывода*, т. е. чтения и записи на диске последовательных сегментов памяти.

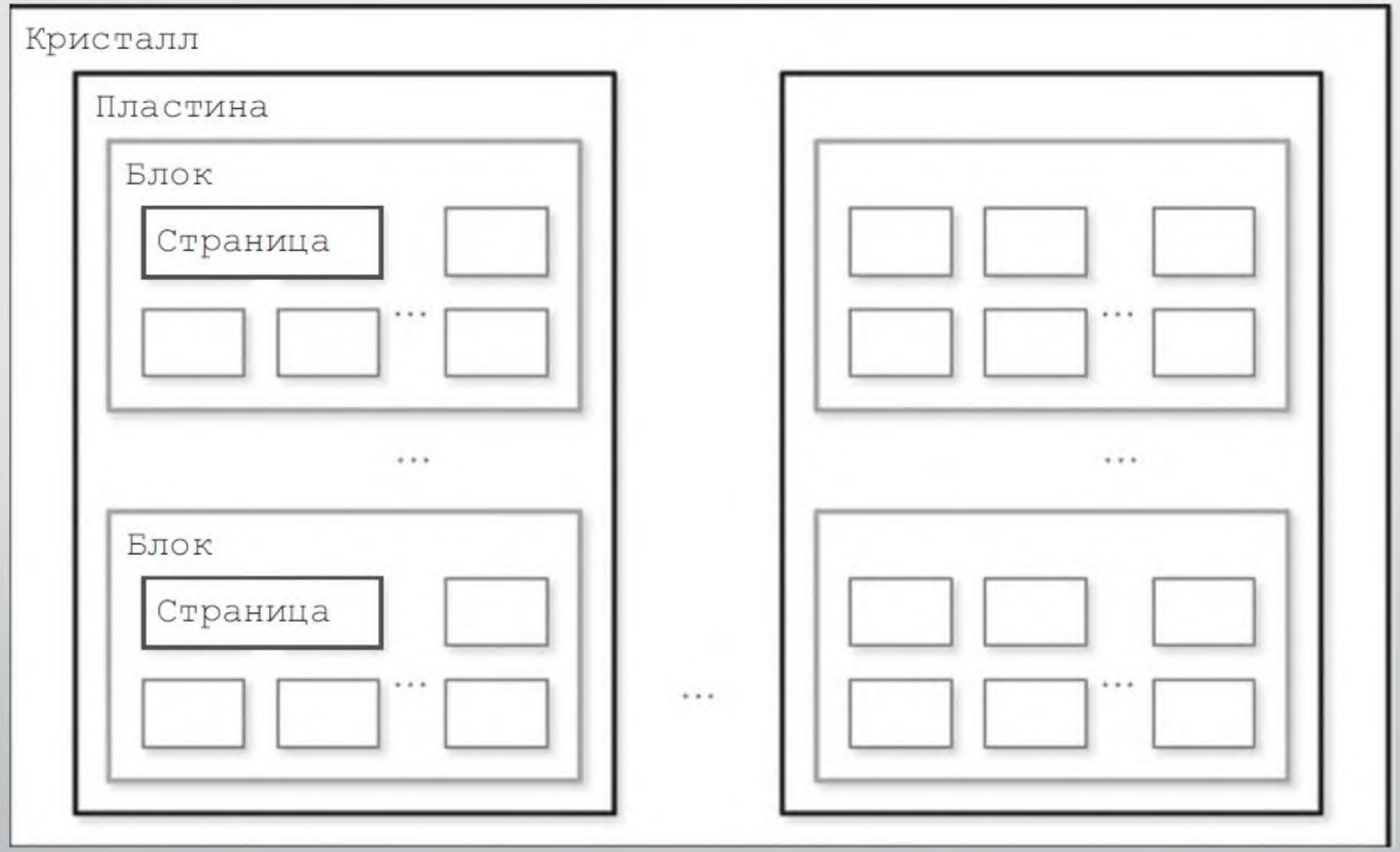
Твердотельные накопители

Твердотельные накопители (SSD) не имеют движущихся частей: нет диска, который вращается, или головки, которую необходимо позиционировать для чтения. Типичный твердотельный накопитель состоит из *ячеек памяти*, соединенных в *строки* (обычно от 32 до 64 ячеек на строку), строки объединяются в *массивы*, массивы — в *страницы*, а страницы — в *блоки*.

В зависимости от используемой технологии ячейка может содержать один или несколько битов данных. Размер страниц может быть разным в зависимости от устройства, но находится в диапазоне от 2 до 16 Кб. Блоки, как правило, содержат от 64 до 512 страниц. Блоки организованы в *пластины*, и, наконец, отдельные пластины составляют *кристалл*. Твердотельные накопители могут иметь один или несколько кристаллов.

Минимальная единица, которую можно записать (запрограммировать) или прочитать, — это страница. Однако мы можем вносить изменения только в пустые ячейки памяти (т. е. в те, которые были стерты до записи). Минимальный объект для стирания — это не страница, а блок, содержащий несколько страниц, поэтому его часто называют *блоком стирания*. Страницы в пустом блоке должны записываться последовательно.

Схема организации SSD



Часть контроллера флэш-памяти, отвечающая за сопоставление идентификаторов страниц с их физическим местоположением, отслеживание пустых, записанных и отброшенных страниц, называется слоем трансляции флэш-памяти (Flash Translation Layer, FTL).

Данный слой также отвечает за операцию сборки мусора, во время которой он находит блоки, которые можно безопасно стереть.

Некоторые блоки все еще могут содержать актуальные страницы. В этом случае FTL перемещает актуальные страницы из этих блоков в новые места и изменяет идентификаторы страниц, чтобы они указывали на их местоположение. После этого он стирает неиспользуемые блоки, делая их доступными для записи.

Поскольку в обоих типах устройств (жестких дисках и твердотельных накопителях) мы адресуем области памяти, а не отдельные байты, в большинстве операционных систем имеется абстракция ***блочного устройства***. Она скрывает внутреннюю структуру диска и буферизует операции ввода-вывода, поэтому, когда мы считываем из блочного устройства *одно слово*, полностью считывается *весь содержащий его блок*. Это ограничение нельзя игнорировать и необходимо всегда учитывать при работе с дисковыми структурами данных.

В отличие от жестких дисков, в твердотельных накопителях нет особой разницы между произвольным и последовательным вводом-выводом, поскольку разница в задержках между произвольным и последовательным считыванием не так велика.

Несмотря на то что сборка мусора обычно является фоновой операцией, ее последствия могут отрицательно сказаться на производительности записи, особенно в случае записи в произвольном порядке и без выравнивания записей по размеру и смещению блока на диске.

Запись только полных блоков и объединение последовательных операций записи в один и тот же блок может помочь сократить количество требуемых операций ввода-вывода.

Благодарю за внимание!