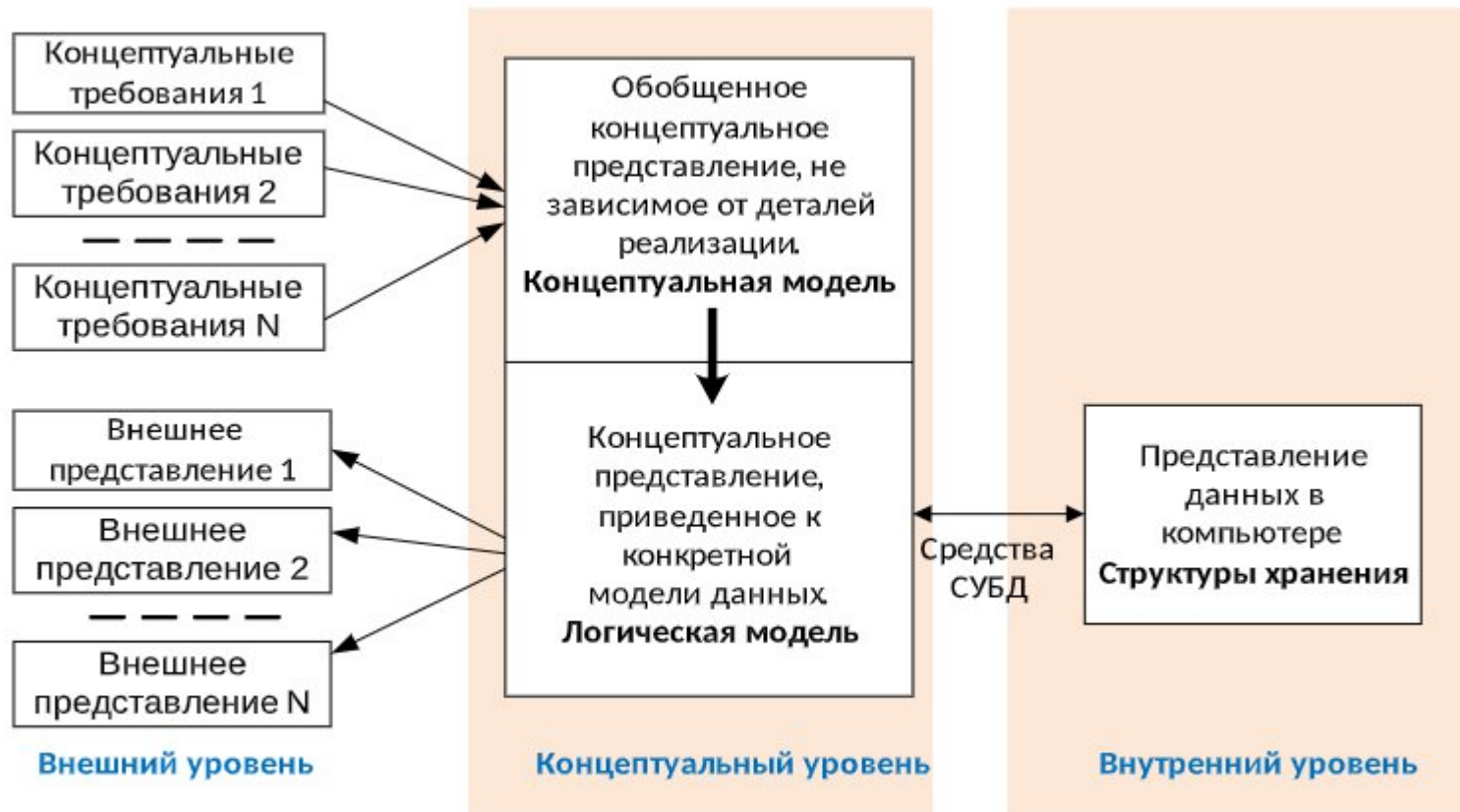


Лекция 12.

**Внутреннее устройство
базы данных**

Трехуровневая архитектура данных

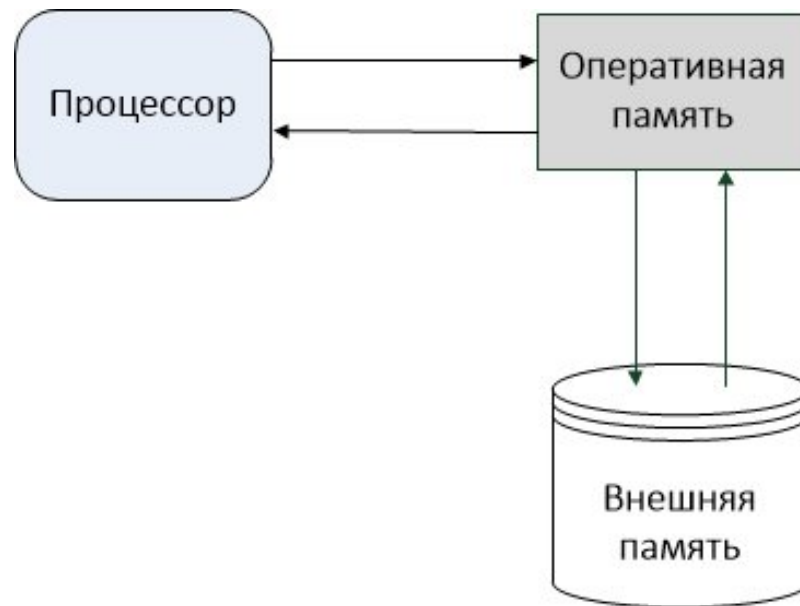


Логическая модель и внутреннее представление – для администратора базы данных.

Модели данных для оперативной и внешней памяти

Физические модели данных

База данных хранится во внешней памяти, а процессор работает только с оперативной памятью.



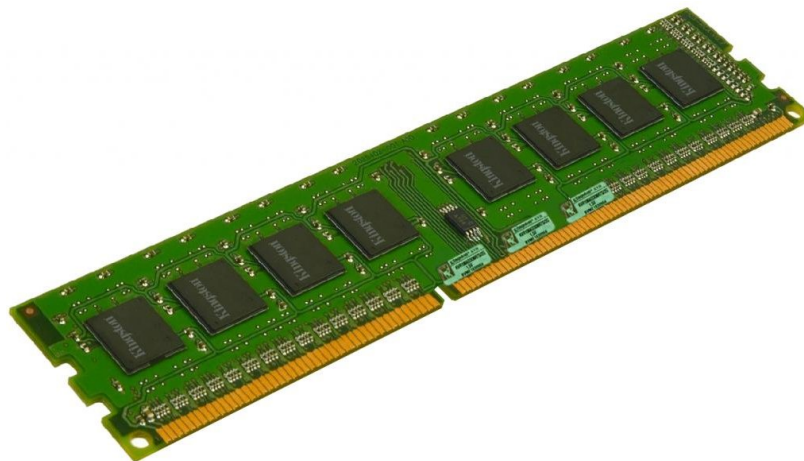
Поэтому организация данных должна учитывать:

- специфику каждого вида памяти,
- способы их взаимодействия.

Оперативная память (RAM)

Random Access Memory (RAM) - адресуемая память с произвольным доступом.

- Каждый **байт** имеет свой адрес, и контроллер памяти может прочитать/записать его **независимо**.
- Задержки доступа к любому байту — одинаковы и очень малы (наносекунды).
- Процессор выбирает нужные данные, непосредственно адресуясь к последовательности байтов, содержащих эти данные.



Внешняя память

- Внешняя память - это блочно-адресуемое устройство.
- Минимальная единица ввода/вывода - **физическая запись** объемом от 512 байт до 16 Kb.



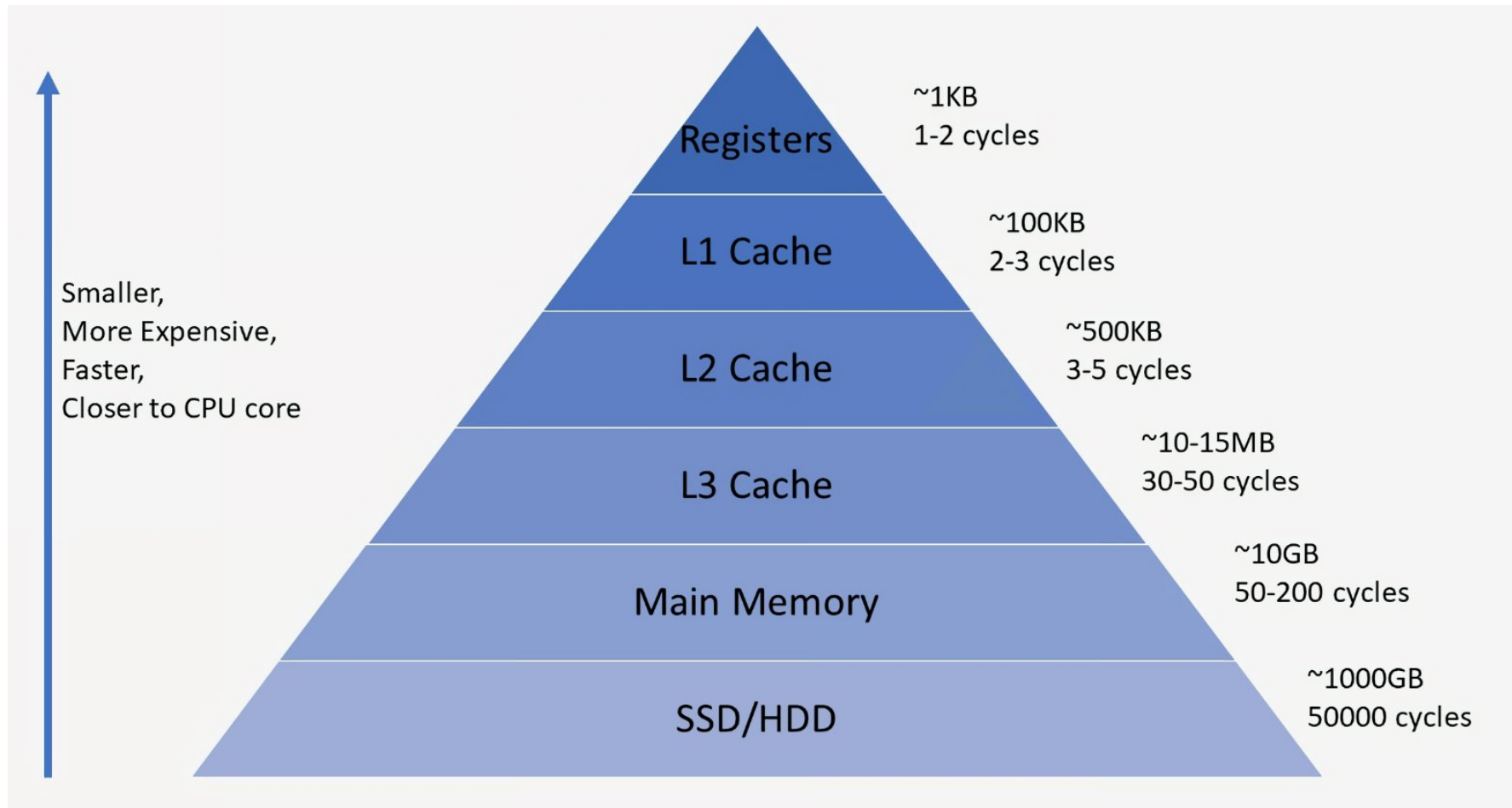
Время чтения 1 Мб данных:

RAM - 0,25 ms

SDD - 1 ms

HDD - 20 ms

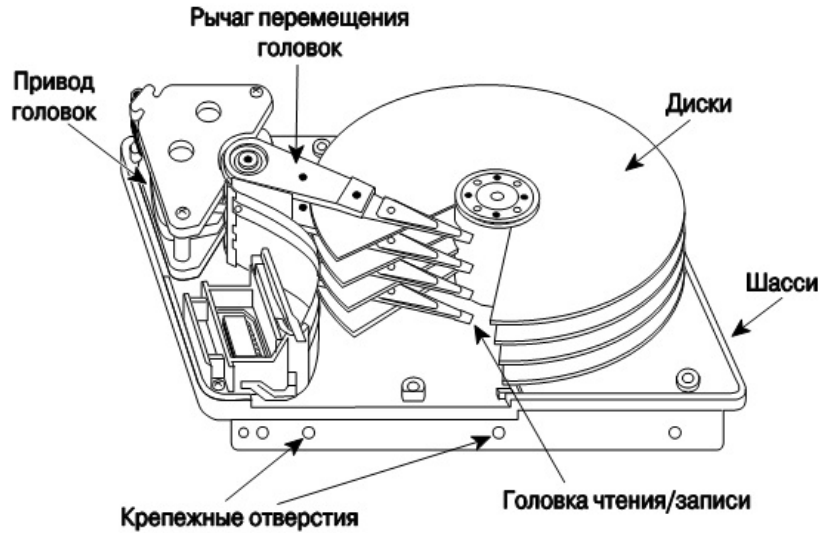
Иерархия памяти



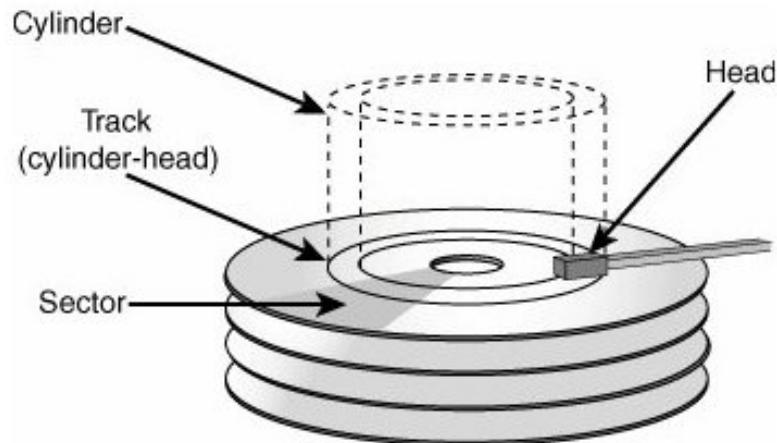
Почему операции ввода/вывода с внешней памятью реализованы как блочные, а не побайтные?

Геометрия классического HDD и адресация CHS

Устройство классического HDD



- **Цилиндр** — это множество дорожек (tracks) с одинаковым номером на всех поверхностях (сторонах) диска.
- **Головка** — это одна поверхность диска, а также устройство чтения/записи, соответствующее этой поверхности.
- **Сектор** — это наименьшая адресуемая единица данных на дорожке, представляющая собой дугу дорожки, выделенную для хранения фиксированного объема данных.



• Пример CHS-адреса

Адрес `C=1023, H=15, S=63` означает:

- Переместить головки на 1023-й цилиндр (радиус),
- Выбрать 15-ю головку (поверхность),
- Прочитать 63-й сектор на этой дорожке.

1. Физические ограничения устройств хранения

★ Магнитные диски (HDD)

- Данные записаны по дорожкам и секторам.
- Минимальная единица чтения/записи — сектор (традиционно 512 байт, сейчас часто 4096 байт — «Advanced Format»).
- Чтение одного байта физически невозможно: головка читает весь сектор целиком, даже если вам нужен один байт.
- Перемещение головки (seek) и ожидание поворота диска (rotational latency) — очень медленные операции (миллисекунды), поэтому эффективнее читать/писать большие блоки за раз.

1. Физические ограничения устройств хранения

★ SSD и флеш-память

- Хотя SSD не имеют движущихся частей, они тоже работают блоками:
 - Минимальная единица чтения — страница (обычно 4 КБ),
 - Минимальная единица стирания — блок (например, 256 КБ).
- Запись одного байта невозможна: нужно перезаписать всю страницу (а при необходимости — выполнить «копирование-обновление» всего блока).
- Поэтому SSD тоже требуют блочных операций.

2. Экономическая эффективность

Даже если бы оборудование позволяло побайтный доступ, это было бы крайне неэффективно:

- Каждая операция I/O имеет накладные расходы: команды контроллера, поиск сектора, передача данных, проверка CRC и т.д.
- Если читать по 1 байту, накладные расходы в миллионы раз превысят полезную нагрузку.
- Блочные операции амортизируют эти накладные расходы, делая I/O приемлемым по скорости и стоимости.

“Например: чтение 4 КБ за 0.1 мс = 40 МБ/с.

Чтение 1 байта за 0.1 мс = 10 КБ/с — в 4000 раз медленнее.”

3. Принцип локальности и предсказуемости

Программы и пользователи часто обращаются к смежным данным (например, читают файл последовательно или работают с записью в базе данных).

Блочная передача позволяет:

- Предзагружать (prefetch) следующие блоки,
- Кэшировать целые блоки в ОЗУ (буферный кэш),
- Уменьшать количество обращений к медленному устройству.

Это напрямую связано с принципом локальности ссылок (пространственной и временной).

4. Роль буферизации

Буферизация — не причина, а следствие и механизм адаптации к блочному I/O:

- ОС и приложения используют буферы в ОЗУ, чтобы:
 - Накапливать побайтные операции от программ (например, `fwrite()` в C),
 - Потом сбрасывать их целыми блоками на диск.
- Это скрывает блочную природу диска от программиста, позволяя писать `putc()` или `file.write("a")`, но физически на диск уйдёт целый блок.

“То есть: буферизация компенсирует несоответствие между побайтной логикой программ и блочным I/O устройств.”

✓ **Общий вывод**

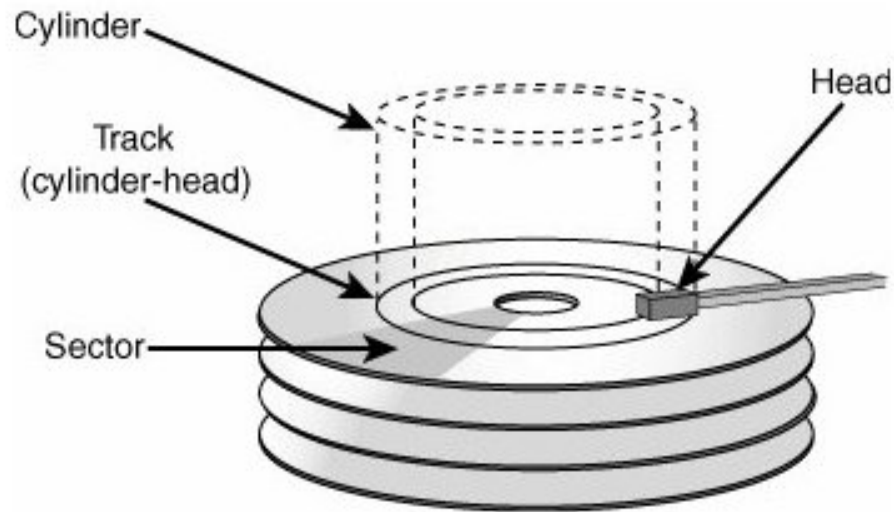
Операции ввода-вывода **блочные**, потому что:

1. **Физически** устройства хранения (HDD, SSD) работают с блоками (секторами/страницами).
2. **Экономически** побайтный доступ был бы катастрофически неэффективен.
3. **Архитектурно** блочный I/O лучше согласуется с принципами локальности и кэширования.
4. **Буферизация** — это программный механизм, позволяющий программам работать «как будто» побайтно, но **не причина** блочности, а её следствие.

Таким образом, блочная природа внешней памяти — это не историческая случайность, а неизбежное следствие физики, инженерии и эффективности.

Адресация CHS заменяется на LBA

Устройство классического HDD



♦ Пример CHS-адреса

Адрес `C=1023, H=15, S=63` означает:

- Переместить головки на 1023-й цилиндр (радиус),
- Выбрать 15-ю головку (поверхность),
- Прочитать 63-й сектор на этой дорожке.

Проблемы CHS (Cylinder-Head-Sector)

CHS — это старая модель адресации, основанная на физической геометрии диска:

- Цилиндр (C) — радиальное положение,
- Головка (H) — номер поверхности,
- Сектор (S) — позиция вдоль дорожки.

Основные недостатки CHS:

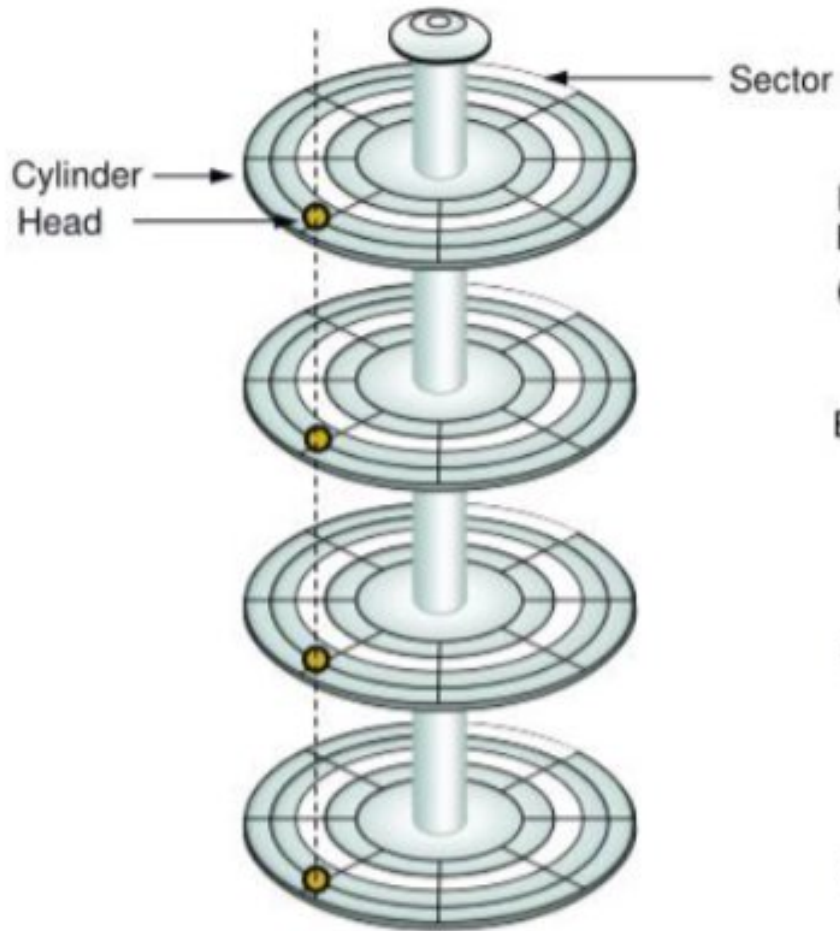
1. **Жёсткая привязка к физической геометрии**
— Но с ростом ёмкости дисков реальная геометрия перестала соответствовать простой модели (например, внешние дорожки содержат больше секторов, чем внутренние — технология зонированной записи).
2. **Ограничение объёма диска**

LBA (Logical Block Addressing — логическая блоковая адресация) — это метод адресации данных на блочных устройствах хранения (жёстких дисках, SSD и т.п.), при котором все секторы нумеруются последовательно одним целым числом, начиная с нуля.

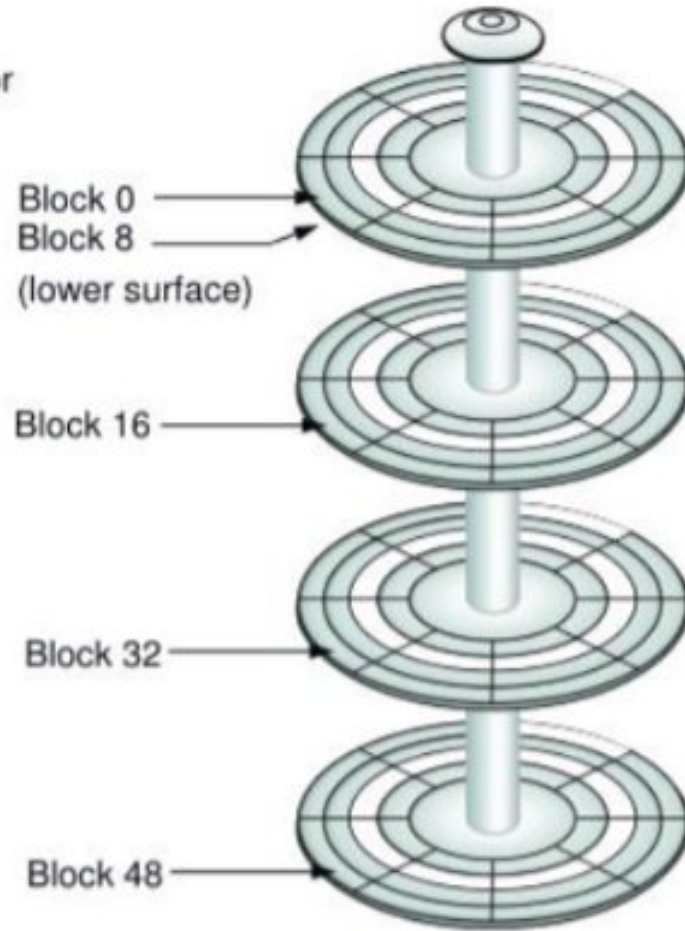
Пример:

- Сектор 0 → первый сектор на диске,
- Сектор 1 → следующий,
- ...
- Сектор 1 000 000 → миллионный сектор.

CHS и LBA



Physical Address = CHS



Logical Block Address = Block #

1. Простота и универсальность

— ОС и приложения работают с диском как с линейным массивом секторов.

— Нет нужды знать, как устроен диск внутри.

2. Масштабируемость — Современные LBA используют 48-битную адресацию (в стандарте ATA/6 и выше):

$$2^{48} \times 512 \text{ Б} \approx 128 \text{ ПБ (петабайт)}$$

— Этого хватит надолго.

3. Абстракция от физики

— Диск сам решает, где физически хранить сектор с заданным LBA.

— Это позволяет:

- Использовать `wear leveling` в SSD,
- Переназначать бэд-блоки,
- Применять зонированную запись (SMR) на HDD без ущерба для ОС.

4. Единый интерфейс для всех типов носителей

— Один и тот же протокол (например, SATA) и модель LBA работают и для HDD, и для SSD, и для виртуальных дисков.

✓ Вывод

LBA заменила CHS, потому что:

- CHS устарела из-за ограничений объёма и привязки к физической геометрии,
- LBA обеспечивает простую, масштабируемую и универсальную абстракцию поверх любого устройства хранения,
- Современные носители (особенно SSD) физически не соответствуют модели CHS.

Таким образом, LBA — это ключ к совместимости, гибкости и эффективности в современных системах хранения данных.

Дилемма блочного ввода/вывода

Как эффективно обрабатывать объёмы данных, многократно превышающие размер оперативной памяти, используя медленное устройство с минимальной единицей доступа в тысячи байт?



Эту проблему пытаются решить файловые системы

Архитектура файловой системы

Ключевая инженерная идея файловых систем

Файловая система

обеспечивает:

- удобный
- именованный
- потоковый

доступ к (очень) большим данным на

- медленных блочных устройствах

несмотря на

- критически ограниченные ресурсы оперативной памяти.

Логическая и физическая структура файлов

Для прикладной программы файл – это **именованная область внешней памяти**, в которую можно записывать и из которой можно считывать данные.



Для ФС файл – это **цепочка кластеров** (физических записей).

Кластер в файловой системе

Кластер (блок данных) — это группа одного или нескольких физических секторов на диске, объединённых файловой системой в логическую единицу выделения для хранения файлов.

- Размер кластера обычно: 512 Б, 1 КБ, 4 КБ, 16 КБ, 32 КБ и т.д
- Выбирается при форматировании диска.
- Все кластеры в разделе — одинакового размера.

Зачем нужны кластеры

1. Снижение накладных расходов

- Если бы файловая система управляла каждым сектором, таблицы размещения (например, FAT) стали бы огромными.
- Кластеры уменьшают количество «единиц учёта».

2. Совместимость с блочным I/O

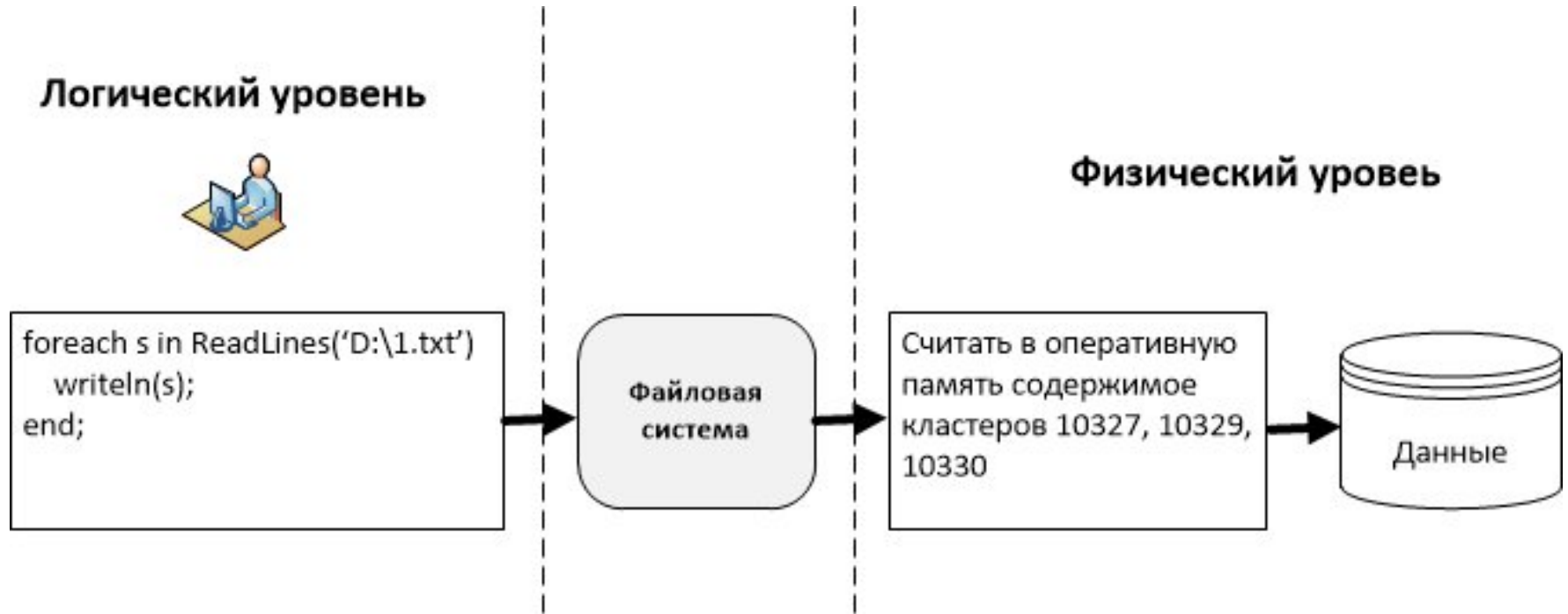
- ОС и диски работают блоками. Кластер — это логический блок, удобный для файловой системы.

3. Баланс между фрагментацией и эффективностью

- Маленькие кластеры → меньше потерь на фрагментацию, но больше служебных данных.
- Большие кластеры → меньше метаданных, но больше «пустой» памяти под мелкие файлы.

Файловая система

Пользователь/программист имеет дело только с логическими данными, не касаясь деталей фактического размещения данных.



Архитектурные особенности классических ФС

1. Минимизация использования ОЗУ

Файловые системы были спроектированы так, чтобы эффективно работать с дисковыми блоками (например, 512 байт или 4 КБ), а не с содержимым файлов. Это позволяло:

- Читать и записывать данные блоками, а не байт за байтом.
- Использовать небольшие буферы в памяти (например, один блок за раз), что критично при малом объёме ОЗУ.

2. Отсутствие осведомлённости о содержимом

Файловая система не знала, что внутри файла — текст, изображение, база данных или исполняемый код. Это упрощало её архитектуру, делало её универсальной и стабильной.

Архитектурные особенности классических ФС

3. Оптимизация под последовательный доступ

Многие ранние файловые системы (например, Unix File System, FAT) оптимизировались под последовательное чтение/запись больших файлов, что снижало фрагментацию и количество обращений к диску.

4. Метаданные минимальны

Хранились только базовые атрибуты: имя, размер, права доступа, временные метки и указатели на блоки данных. Никаких индексов, тегов, семантических связей.

Простейшая ФС FAT

C:\Документы\Мой файл.doc

Каталог файлов

Дескриптор файла 1
...
Дескриптор файла N

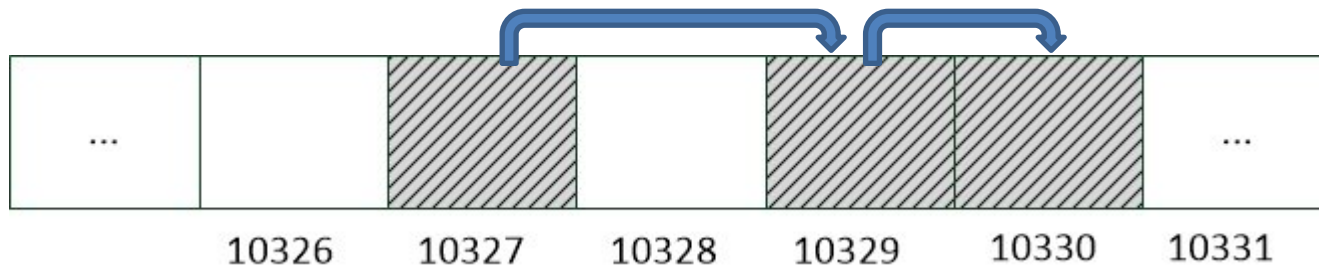
Дескриптор файла

Имя	<i>Мой файл.doc</i>
Дата создания	<i>01.02.2020</i>
Атрибуты	<i>Archive</i>
Первый кластер	<i>10327</i>
Размер	<i>9326</i>
...	...

Таблица размещения файлов (FAT)

№ кластера	Статус
10326	<i>Сбойный</i>
10327	<i>10329</i>
10328	<i>Свободный</i>
10329	<i>10330</i>
10330	<i>Конец цепочки</i>
10331	<i>Свободный</i>
...	...

Кластеры, образующие файл, организованы в **связный список**



Кластеры на диске

Чтение файла в ФС FAT

1. Найти запись файла в каталоге → получить размер и первый кластер.
2. Из BPB определить параметры диска: секторов на кластер, позиции FAT и данных.
3. Прочитать FAT (или использовать кэш FAT) для обхода цепочки.
4. Для каждого кластера:
 - Преобразовать номер кластера → начальный LBA,
 - Прочитать `SecPerClus` секторов в память.
5. Передать данные приложению как непрерывный поток.
6. Остановиться, когда прочитано нужное число байт.

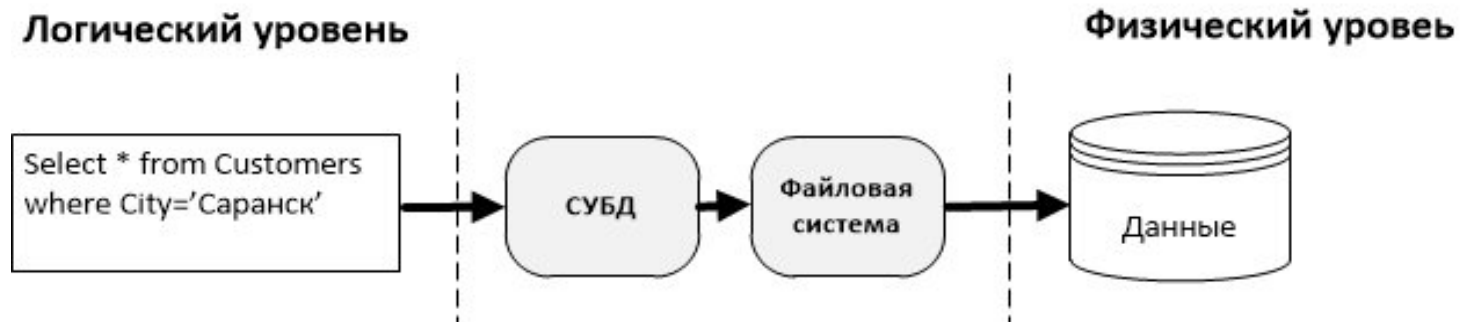
Архитектура СУБД

СУБД - абстракция над файловой системой

Работа с файлом



Работа с базой данных



СУБД как интеллектуальная надстройка над ФС

СУБД использует файловую систему как основу хранения, но добавляет высокоуровневую, логически организованную и управляемую среду для работы с данными — превращая «грудю файлов» в осмысленную, надёжную и эффективную информационную систему.

1. Хранение на уровне файловой системы

- В конечном счёте данные СУБД всё равно хранятся в файлах на диске, управляемых файловой системой (например, NTFS, ext4 и т.п.).
- Однако СУБД не просто записывает и читает файлы «как есть» — она организует их структурированно, с учётом логической модели данных (таблицы, индексы, связи и т.д.).

СУБД как интеллектуальная надстройка над ФС

СУБД добавляет множество «умных» возможностей поверх простого хранения файлов:

- **Структурирование данных:** данные организованы в таблицы, схемы, типы, связи (в реляционных СУБД) или иные логические структуры (в NoSQL).
- **Язык запросов** (например, SQL): позволяет описывать *что нужно*, а не *как это получить* — СУБД сама решает, какие файлы и как читать.
- **Оптимизация запросов:** СУБД анализирует запросы и выбирает наиболее эффективные пути доступа к данным (план выполнения).
- **Транзакционность (ACID):** обеспечивает целостность данных даже при сбоях, параллельном доступе и т.д.

СУБД как интеллектуальная надстройка над ФС

- **Индексы:** ускоряют поиск, как «оглавление» в книге — без полного перебора файлов.
- **Параллелизм и блокировки:** управляет одновременным доступом множества пользователей.
- **Безопасность:** контроль доступа на уровне объектов и операций.
- **Резервное копирование и восстановление:** встроенные механизмы защиты от потери данных.

СУБД как интеллектуальная надстройка над ФС

3. Абстракция от файлов

Пользователь или разработчик работает с логическими сущностями (таблицами, представлениями), не заботясь о том, как и где именно хранятся байты на диске. Это и есть суть «интеллектуальной надстройки».

SQLite — это идеальный пример «интеллектуальной надстройки над файловой системой»: она сохраняет простоту работы с файлами, но обеспечивает мощные возможности реляционной СУБД.

1. Бессерверная архитектура

- Не требует отдельного серверного процесса.
- Вся СУБД реализована как библиотека, встраиваемая непосредственно в приложение.

2. Хранение в одном файле

- Вся база данных (схема, таблицы, индексы, данные) хранится в одном обычном файле на диске.
- Это упрощает перенос, резервное копирование и управление базой.

3. Минималистичность и автономность

— Нет необходимости в установке, настройке или администрировании.

— Идеально подходит для встраиваемых систем, мобильных приложений, прототипирования и локального хранения.

4. Полная поддержка SQL (с оговорками)

— Поддерживает большинство стандартных SQL-возможностей: транзакции (ACID), индексы, триггеры, представления и т.д.



Дуэйн Ричард Хипп (D. Richard Hipp), 1961 г.р.

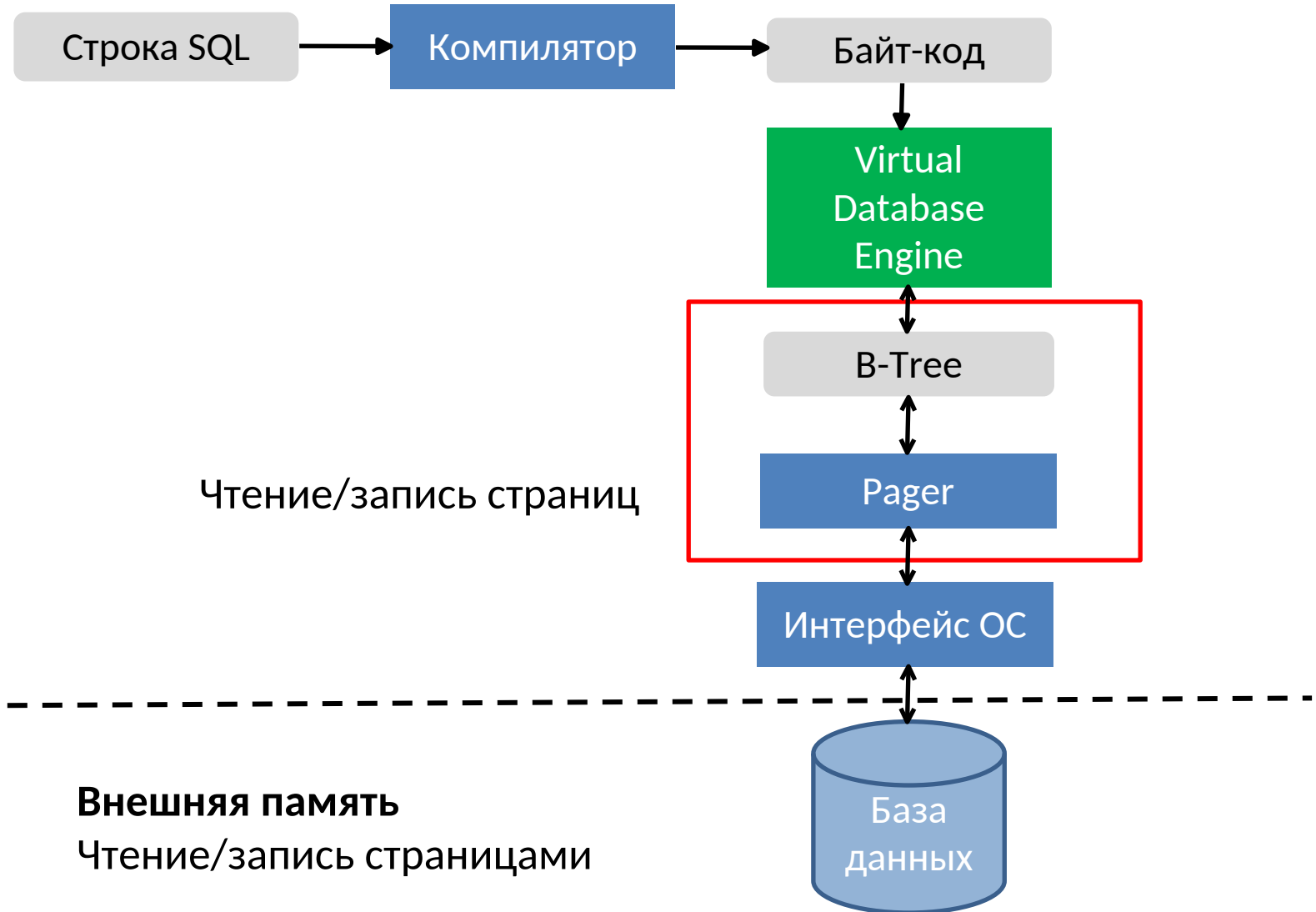
Создал СУБД SQLite, систему контроля версий Fossil,
участвовал в разработке языка TCL

«Если мы подумаем о каждом SQL-операторе как о программе, то нужно просто взять эту программу и скомпилировать её в своего рода исполняемый код. Я придумал структуру байт-кода, который фактически запускал запрос, а затем написал компилятор, который переводил SQL в этот байт-код. И вуаля: родилась СУБД SQLite»

<https://habr.com/ru/company/macloud/blog/566396/>

<https://habr.com/ru/company/macloud/blog/566540/>

Архитектура SQLite



Организация памяти в СУБД

Организация памяти для ФС и СУБД

Файловая система

- **кластеры** - логические единица хранения данных, состоящие из группы секторов диска

СУБД

- **страницы** - фиксированный блок данных, который используется для управления данными в памяти.

Сходства в организация памяти для ФС и СУБД

1. Блочная структура хранения

- **Файловая система:** выделяет и управляет данными кластерами — минимальными единицами выделения на диске (например, 4 КБ).
- **СУБД:** хранит данные в **страницах** (обычно тоже 4–16 КБ), которые являются базовыми единицами ввода-вывода и кэширования.

“Обе системы работают с **фиксированными по размеру блоками**, а не с отдельными байтами, чтобы: ”

- Снизить накладные расходы на адресацию,
- Эффективно использовать память и дисковые операции,
- Упростить управление свободным пространством.

Сходства в организация памяти для ФС и СУБД

2. Кэширование в оперативной памяти

- Файловые системы кэшируют часто используемые кластеры в буферном кэше ОС.
- СУБД имеют собственный буферный пул (**buffer pool**), в котором хранятся **страницы**, загруженные с диска.

“В обоих случаях цель — минимизировать дорогие операции чтения/записи с диска.”

3. Управление свободным пространством

- Файловая система отслеживает **свободные кластеры** (например, через битовую карту или FAT).
- СУБД отслеживает **свободные страницы** и свободное место внутри страниц (например, в PostgreSQL — через FSM, в SQL Server — через PFS).

Различия в организация памяти для ФС и СУБД

Осведомленность о данных

- ФС – нет. Считает файлы неструктурированными байтовыми потоками.
- СУБД – да. Знает структуру записей, индексов, метаданных, транзакций.

Назначение

- ФС – хранение и именованное файлов.
- СУБД – обеспечение семантически корректного, целостного и конкурентного доступа к структурированным данным.

Различия в организация памяти для ФС и СУБД

Уровень абстракции

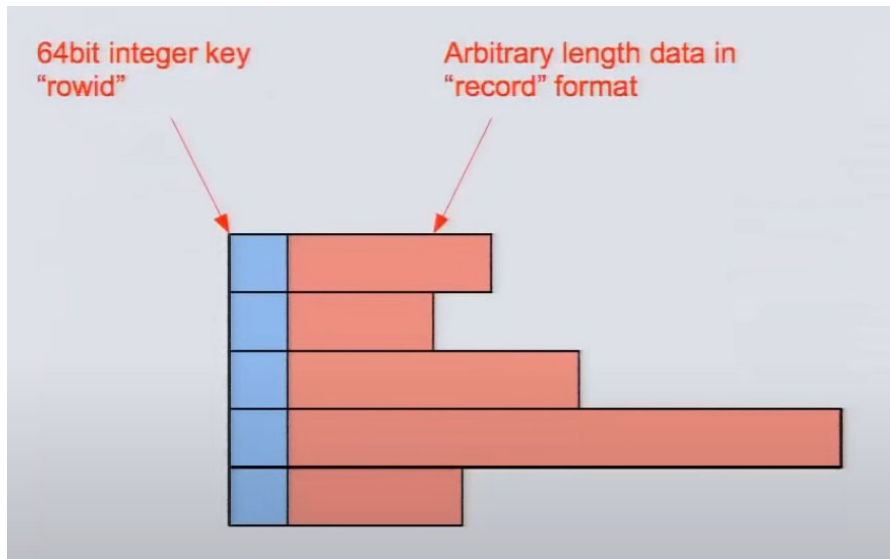
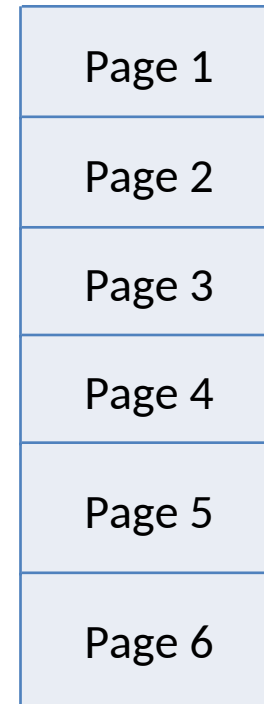
- ФС – низкий: работает с "сырыми" блоками диска.
- СУБД – высокий: страницы содержат логические структуры (таблицы, индексы, представления и т.д.).

Гибкость размещения данных

- ФС – кластеры файла могут быть фрагментированы; связность через таблицы (FAT) или указатели (ext).
- СУБД – СУБД могут размещать логически связанные данные неконтиуально, но с учётом производительности.

Хранение таблиц и индексов в СУБД SQLite

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05



Логическое представление таблицы

Физическая структура БД
(страницы размером 4096 байт
(по умолчанию) внутри одного
файла)

1. Как SQLite читает данные из страниц?
2. Как записи таблицы разнесены по страницам?

Чтение данных из определенной страницы

Шаг 1: Вычисление смещения в байтах внутри файла

```
c
```

```
1 offset = (page_number - 1) * page_size;
```

Например, если `page_size = 4096`, то:

- Страница 1 → смещение 0,
- Страница 2 → смещение 4096,
- Страница 100 → смещение 409 200.

Чтение данных из определенной страницы

Шаг 2: Вызов системного API для чтения

SQLite использует стандартные функции ОС:

- На Unix/Linux: `pread(fd, buffer, page_size, offset)`
- На Windows: `ReadFile` с указанием смещения

“Это логическое смещение в файле, а не физический адрес на диске.”

Чтение данных из определенной страницы

Шаг 3: ОС и файловая система делают остальное

- Файловая система (ext4, NTFS, APFS и т.д.) определяет, в каких кластерах (или экстентах) хранится нужный участок файла.
- Драйвер диска преобразует кластеры в LBA-адреса секторов.
- Контроллер диска читает нужные физические секторы (обычно по 512 Б или 4 КБ).

“SQLite ничего не знает о секторах, кластерах, LBA, CHS и т.п. — всё это скрыто ОС и файловой системой.”

Чтение данных из определенной страницы

- ◆ 4. Кэширование: буферный пул SQLite

Чтобы минимизировать системные вызовы, SQLite хранит недавно использованные страницы в **внутреннем кэше (page cache)**:

- Размер кэша задаётся через `PRAGMA cache_size` .
- Если страница уже в кэше — никакого I/O не происходит.
- Только при промахе кэша — вызывается `pread()` .

Какую структуру данных лучше выбрать для представления таблицы в БД?

Как разложить записи таблицы по страницам?

Критерий: минимальное количество операция чтения/записи страниц при поиске и изменении данных в таблице.

Вспоминаем файловую систему...

Односвязный список?

C:\Документы\Мой файл.doc

Каталог файлов

Дескриптор файла 1
...
Дескриптор файла N

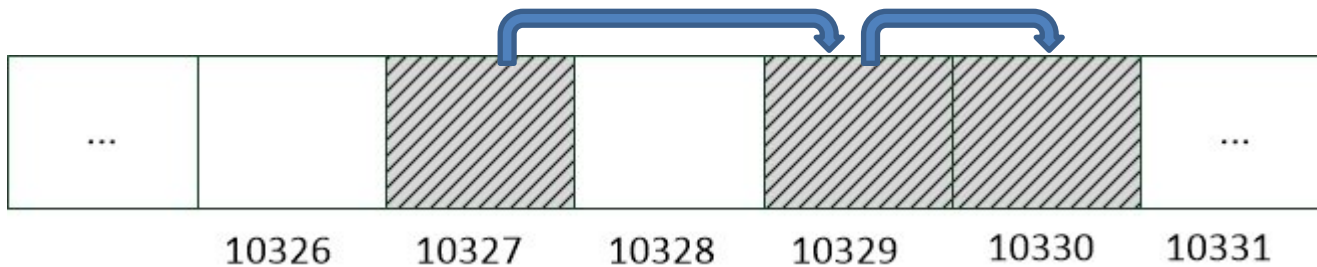
Дескриптор файла

Имя	<i>Мой файл.doc</i>
Дата создания	<i>01.02.2020</i>
Атрибуты	<i>Archive</i>
Первый кластер	<i>10327</i>
Размер	<i>9326</i>
...	...

Таблица размещения файлов (FAT)

№ кластера	Статус
10326	<i>Сбойный</i>
10327	<i>10329</i>
10328	<i>Свободный</i>
10329	<i>10330</i>
10330	<i>Конец цепочки</i>
10331	<i>Свободный</i>
...	...

Кластеры в файле организованы в **связный список**



Кластеры на диске

Страницы для таблицы в виде списка

Page 1

RowId_1	Содержимое полей записи 1
...	...
RowId_k	Содержимое полей записи k



Page 2

RowId_(k+1)	Содержимое полей записи 1
...	...
RowId_2k	Содержимое полей записи k



Page N

RowId_m	Содержимое полей записи 1
...	...
RowId_N*k	Содержимое полей записи k

Страницы для таблицы в виде списка

Оценим T – среднее число обращений к страницам.

- **Поиск записи по ключу RowId.** $T=(1+[N/k])/2=O(N)$
 - **Чтение записи.** $T=(1+[N/k])/2=O(N)$
 - **Корректировка записи.** $T=(1+[N/k])/2+1=O(N)$
 - **Удаление записи.** $T=(1+[N/k])/2+1=O(N)$

Проблемная операция: **поиск записи**

Проблемы организации таблицы в виде списка страниц

1. Линейный поиск по ключу

Если вам нужно найти запись по значению ключа (например, `user_id = 12345`), СУБД должна:

- Последовательно читать каждую страницу,
- Перебирать все записи в ней,
- Пока не найдёт нужную.

“Сложность поиска: $O(N)$, где N — количество записей (или страниц).

При миллионах записей — это неприемлемо медленно.”

Проблемы организации таблицы в виде списка страниц

2. Каждый переход — это потенциальный I/O

Если страницы не находятся в кэше (buffer pool), то каждая новая страница требует чтения с диска.

При 1 000 000 записей и 100 записях на страницу → ~10 000 страниц → до 10 000 операций I/O для одного поиска!

“Даже с SSD это будет сотни миллисекунд вместо долей миллисекунды, которые дают индексы.”

3. Нет локальности данных

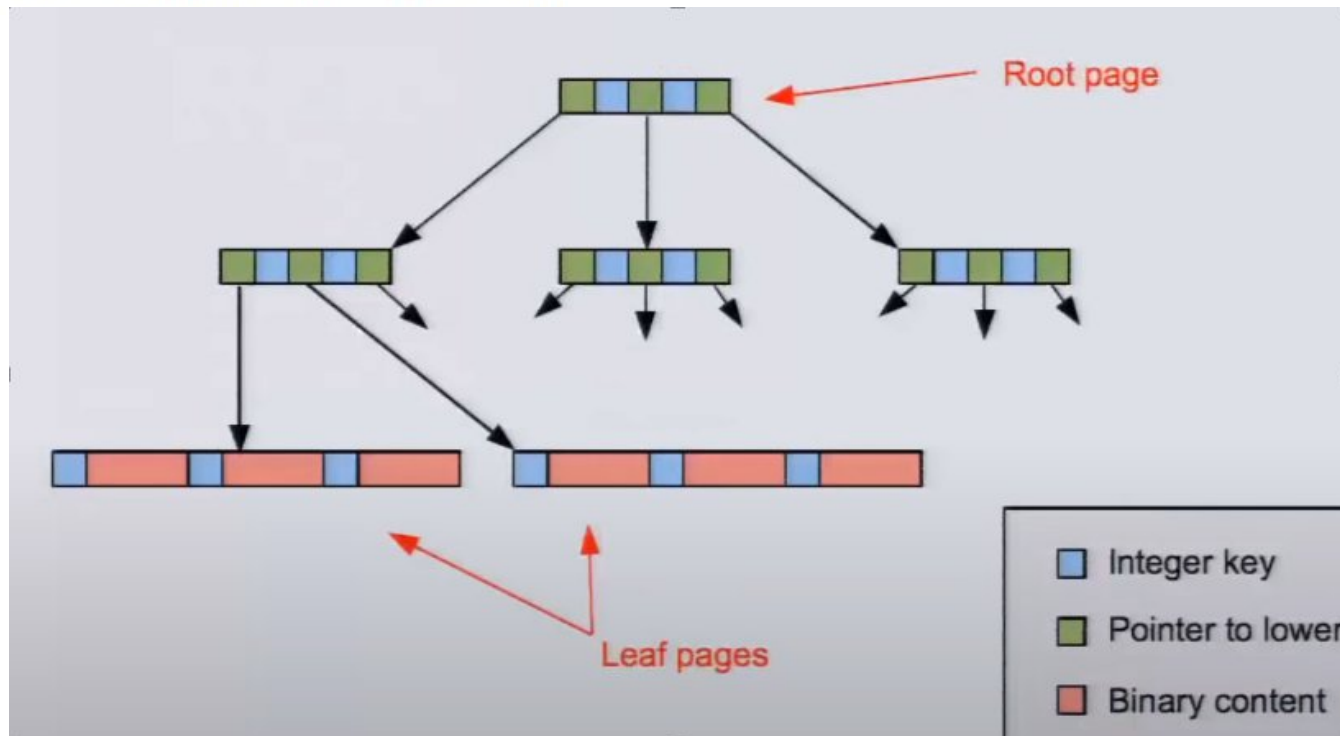
Записи с близкими ключами (например, `id = 1000` и `id = 1001`) могут оказаться в совершенно разных, несмежных страницах, что убивает эффективность кэширования и предвыборки.

Организация таблицы в виде B-дерева страниц

Структура B-дерева для страниц таблицы

B-дерево (произносится «би-дерево», от англ. *balanced tree* или *Bayer-tree*) — это самобалансирующаяся древовидная структура данных, специально разработанная для эффективной работы с внешней (медленной) памятью, например, жёсткими дисками или SSD.

Оно является основой большинства современных индексов в базах данных и каталогов в файловых системах.



Основная идея В-дерева

В отличие от бинарных деревьев (где у каждого узла максимум 2 потомка), в В-дереве узел может иметь десятки или даже тысячи потомков.

Это позволяет сделать дерево очень «низким» и «широким».

Пример:

- При хранении 1 миллиона записей:
 - Бинарное дерево: глубина ≈ 20 уровней \rightarrow до 20 обращений к диску.
 - В-дерево (со 100 детьми на узел): глубина $\approx 2-3$ уровня $\rightarrow 2-3$ обращения к диску.

💡 **Цель В-дерева — минимизировать количество операций ввода-вывода **(I/O). "

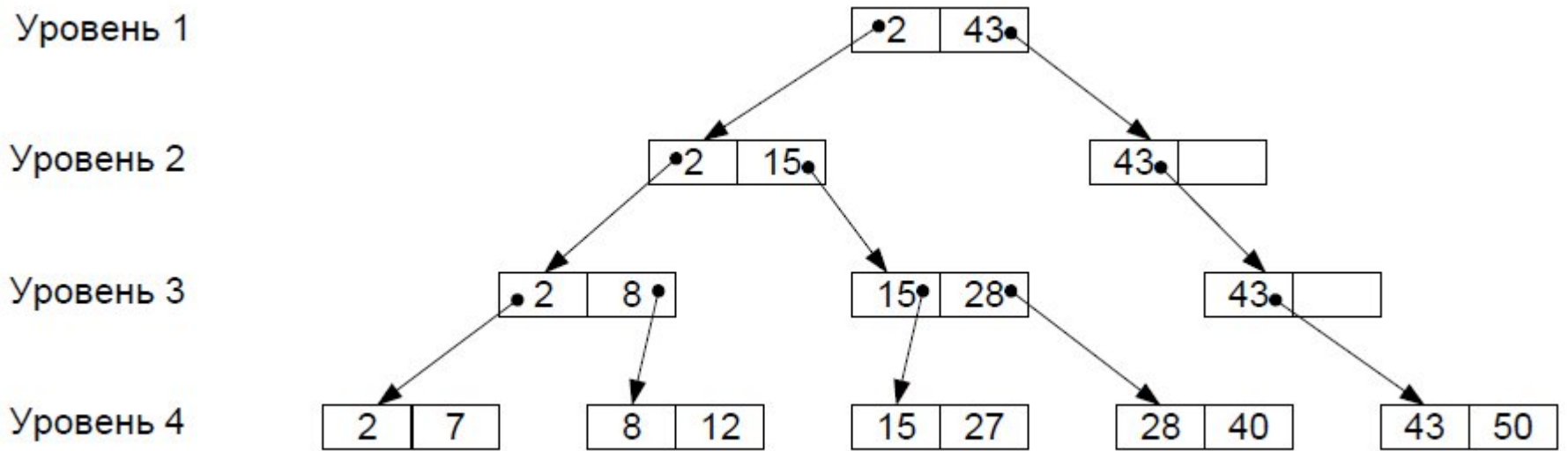
Ключевые свойства B-дерева

1. Все листья находятся на одном уровне → дерево сбалансировано.
2. Каждый узел содержит от $t-1$ до $2t-1$ ключей (где t — минимальная степень дерева).
3. Ключи в узле отсортированы, и они разделяют диапазоны значений для поддеревьев.
4. Каждый узел хранится в одной странице/блоке (например, 4 КБ), что идеально для блочного I/O.

“ ⚠ На практике чаще используется ****B⁺-дерево**** (B⁺-tree), где: ”

- Все данные хранятся **только в листьях**,
- Листья связаны между собой → эффективны диапазонные запросы.

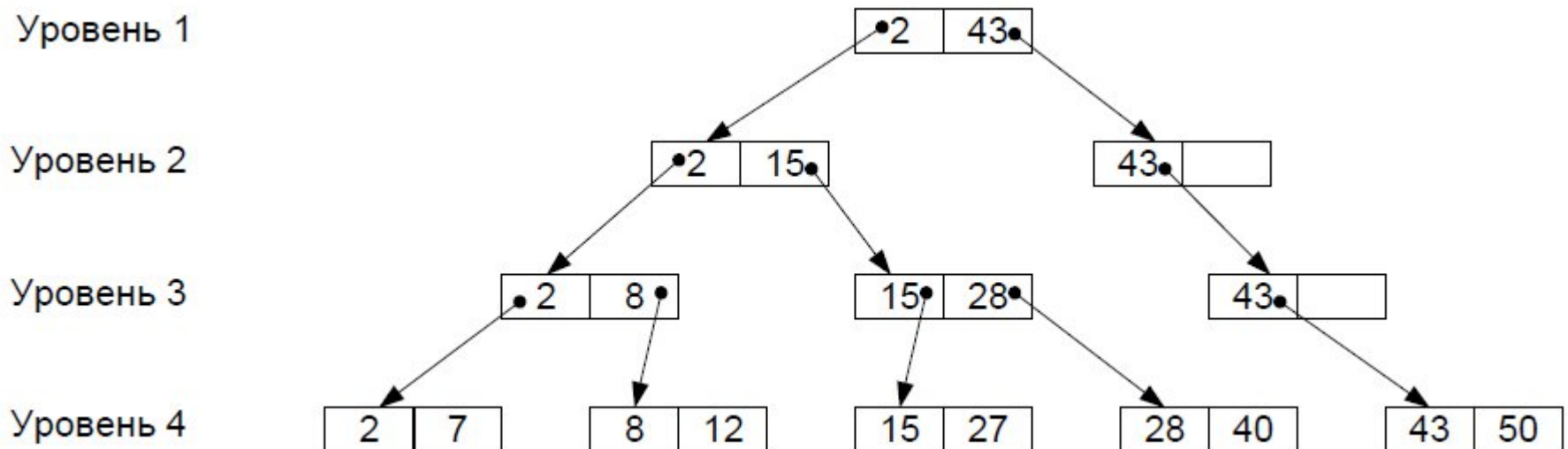
Пример B-дерева порядка $m=2$ для множества из $N=10$ элементов
(2, 7, 8, 12, 15, 27, 28, 40, 43, 50)



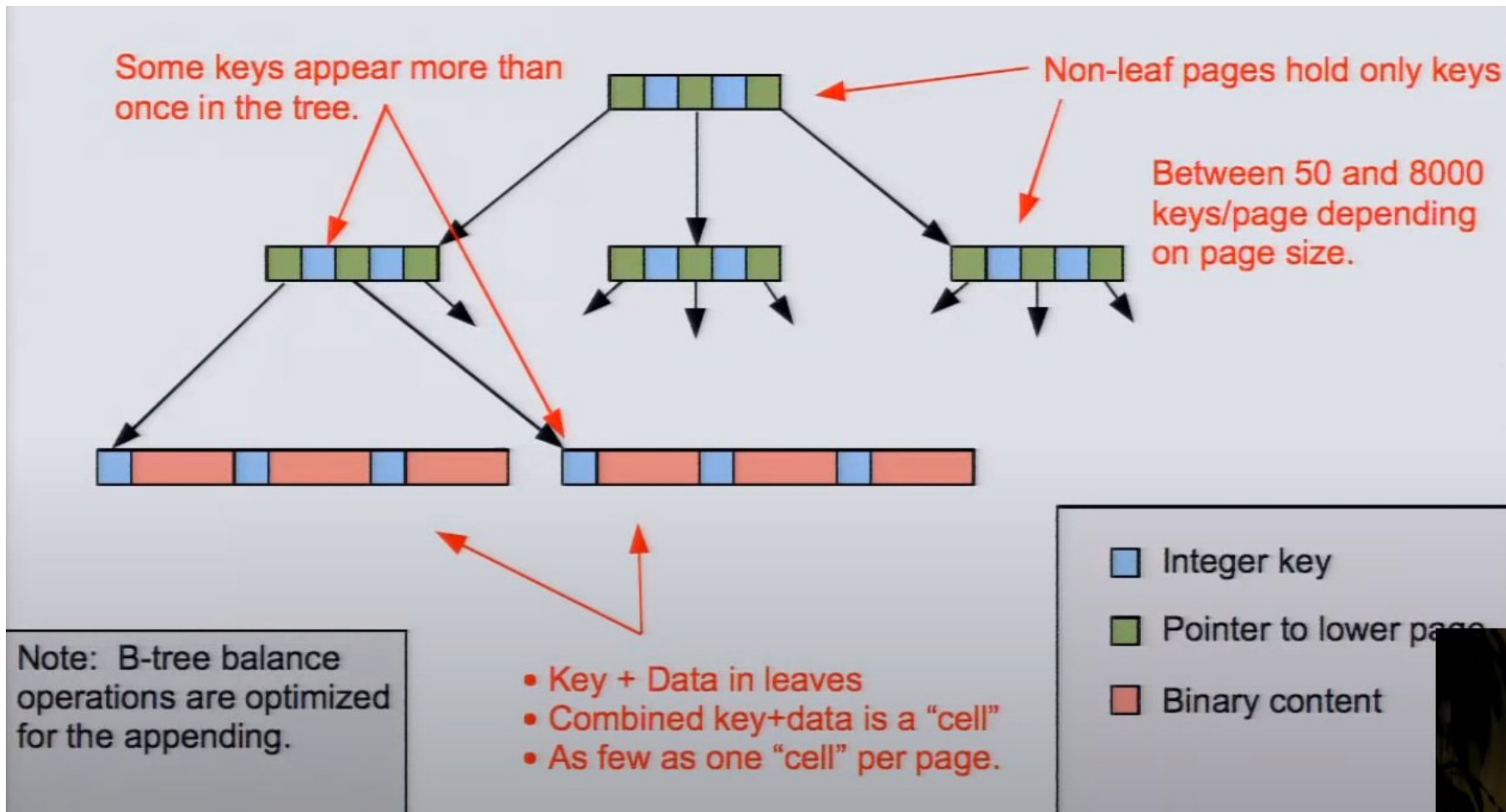
Поиск записи с заданным значением М ключа в В-дереве

1. Читаем верхний индекс.
2. Сравниваем число М со значением ключа записей индекса, начиная с большего значения. Если М больше или равно значению ключа очередной записи индекса, то по адресу связи, указанному в текущей записи, читаем блок записей индекса следующего уровня.
3. Повторяем процесс, пока не дойдем до последнего уровня.

Количество считываний узлов $T=O(\log_2 N)$



B-tree структура для страниц таблицы



Узел = страница

Преимущества B-дерева над связным списком страниц

1. Логарифмическая сложность поиска: $O(\log_k N)$

- k — степень ветвления (число потомков у узла), часто сотни или тысячи.
- Например, при 1 000 000 записей и $k = 100$ → всего 2–3 уровня дерева.
- Значит, для поиска нужно прочитать 2–3 страницы, а не 10 000.

“Это в тысячи раз быстрее, особенно при I/O.”

2. Каждый узел = одна страница

- B-дерево естественно отображается на блочную структуру диска: каждый узел дерева помещается ровно в одну страницу СУБД (обычно 4–16 КБ).
- Это минимизирует количество I/O: один узел — один I/O.

Преимущества B-дерева над связным списком страниц

3. Отсортированные ключи внутри узлов

- Ключи в каждом узле хранятся в отсортированном виде, что позволяет:
 - Быстро находить нужный диапазон с помощью бинарного поиска внутри страницы ($O(\log k)$),
 - Эффективно выполнять диапазонные запросы (`WHERE id BETWEEN 100 AND 200`).

4. Поддержка вставки/удаления без деградации

- B-дерево автоматически балансируется: при вставке/удалении оно перераспределяет ключи между узлами (разделение/слияние страниц), сохраняя сбалансированную структуру.
- Глубина дерева растёт крайне медленно.

Преимущества B-дерева над связным списком страниц

5. Локальность данных

- Записи с близкими ключами физически хранятся в соседних листьях, что:
 - Ускоряет диапазонные запросы,
 - Улучшает кэширование (один I/O — много полезных записей).

✓ Вывод

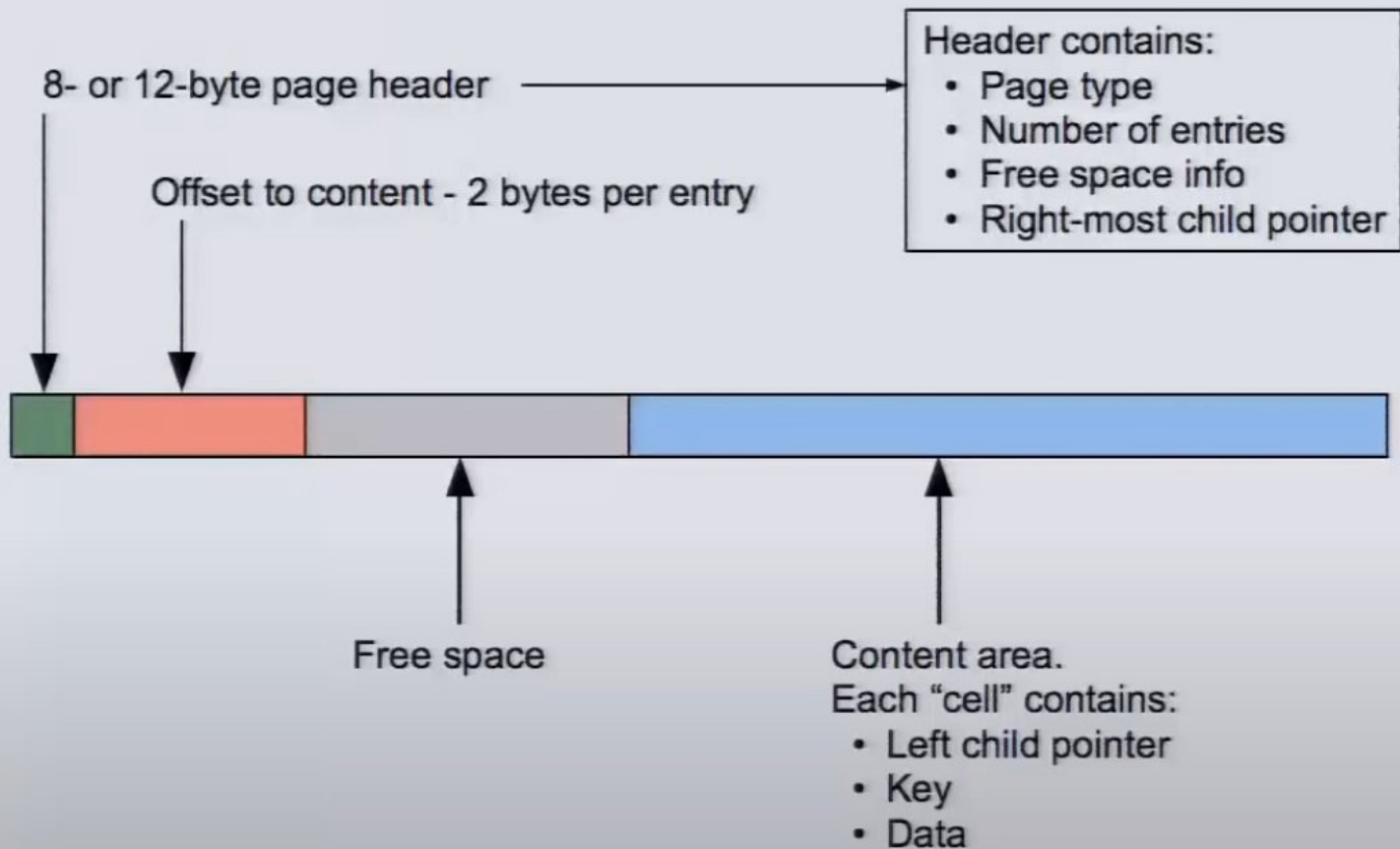
Организация страниц в виде **связного списка** приводит к **линейной зависимости времени поиска от объёма данных**, что неприемлемо для баз данных.

B-дерево решает эту проблему, обеспечивая:

- Логарифмическое время поиска,
- Минимальное количество операций ввода-вывода,
- Эффективную работу с блочной природой дисков,
- Поддержку сортировки и диапазонных запросов.

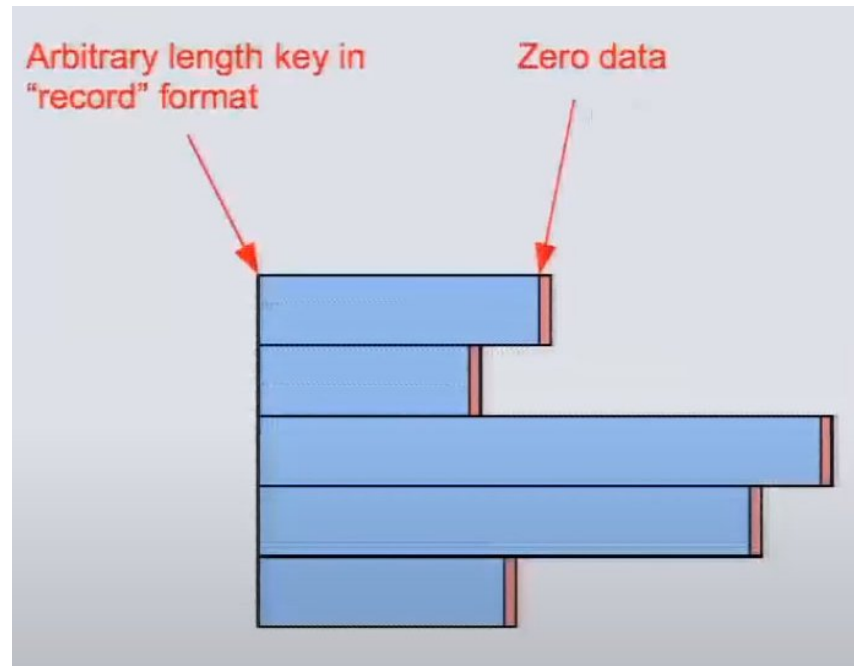
Именно поэтому **B-деревья** (и их варианты, например **B+ деревья**) стали **стандартом де-факто** для организации индексов и часто самой структуры таблиц в реляционных СУБД.

Структура страницы в B-дереве



Логическая структура индекса

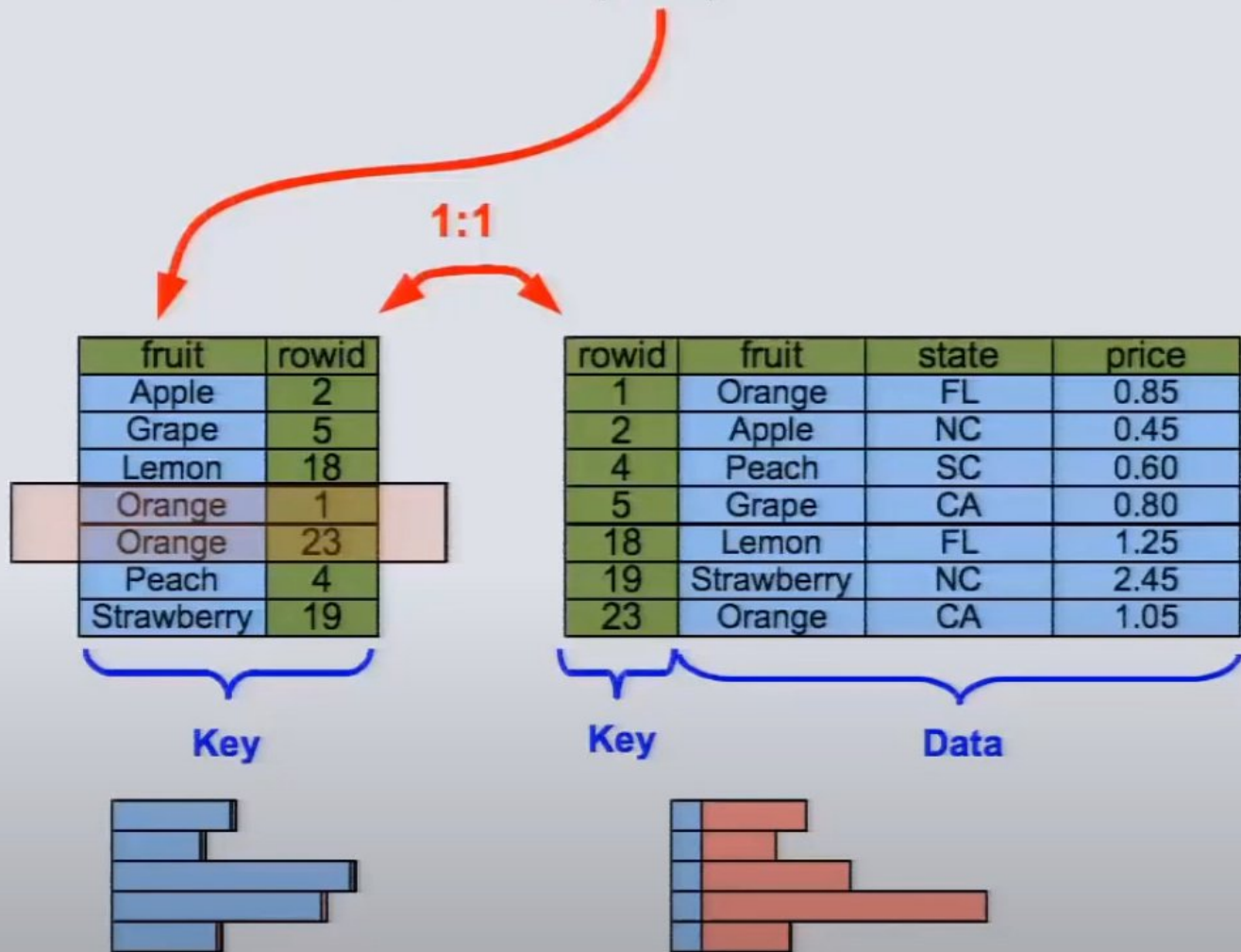
rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05



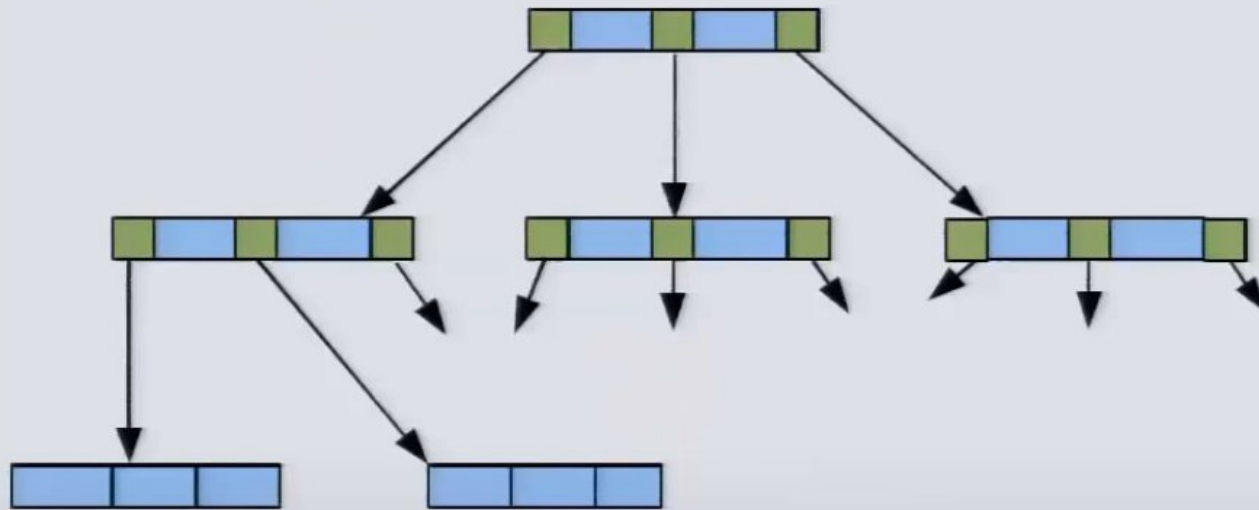
CREATE INDEX Idx1 ON fruits(fruit);

fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

CREATE INDEX idx1 ON tab(fruit)



B-tree структура для страниц индекса

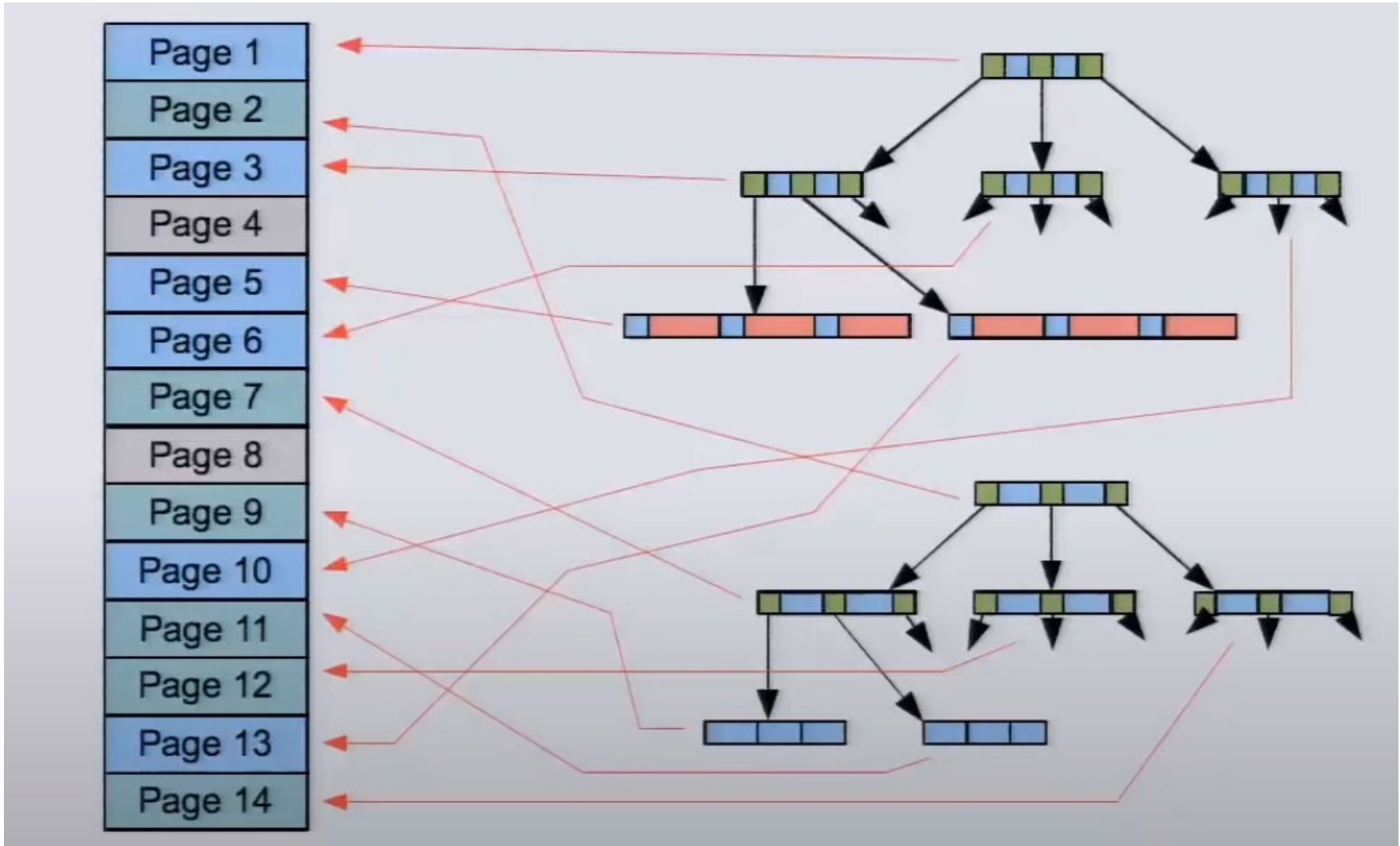


- Key only. No data. The key is the data.
- Larger binary keys, hence lower fan-out
- Each key appears in the table only once
- Minimum 4 keys per page

■ Binary key

■ Pointer to lower page

Соответствие между B-деревьями и страницами



Available pages: 1..1146

- 1: root leaf of table [sqlite_master]
- 2: root interior node of table [blob]
- 3: root interior node of index [sqlite_autoindex_blob_1]
- 4: root interior node of table [delta]
- 5: root interior node of table [rcvfrom]
- 6: root leaf of index [sqlite_autoindex_rcvfrom_1]
- 7: root leaf of table [config]
- 8: root leaf of index [sqlite_autoindex_config_1]
- 9: root leaf of table [shun]
- 10: root leaf of index [sqlite_autoindex_shun_1]
- 11: root leaf of table [private]

...

- 264: leaf of table [blob], child 201 of page 2
- 265: leaf of table [blob], child 202 of page 2
- 266: overflow 1 from cell 0 of page 268
- 267: overflow 2 from cell 0 of page 268
- 268: leaf of table [blob], child 203 of page 2

...

sqlite_master

```
CREATE TABLE sqlite_master(  
  type text,  
  name text,  
  tbl_name text,  
  rootpage integer,  
  sql text  
);
```

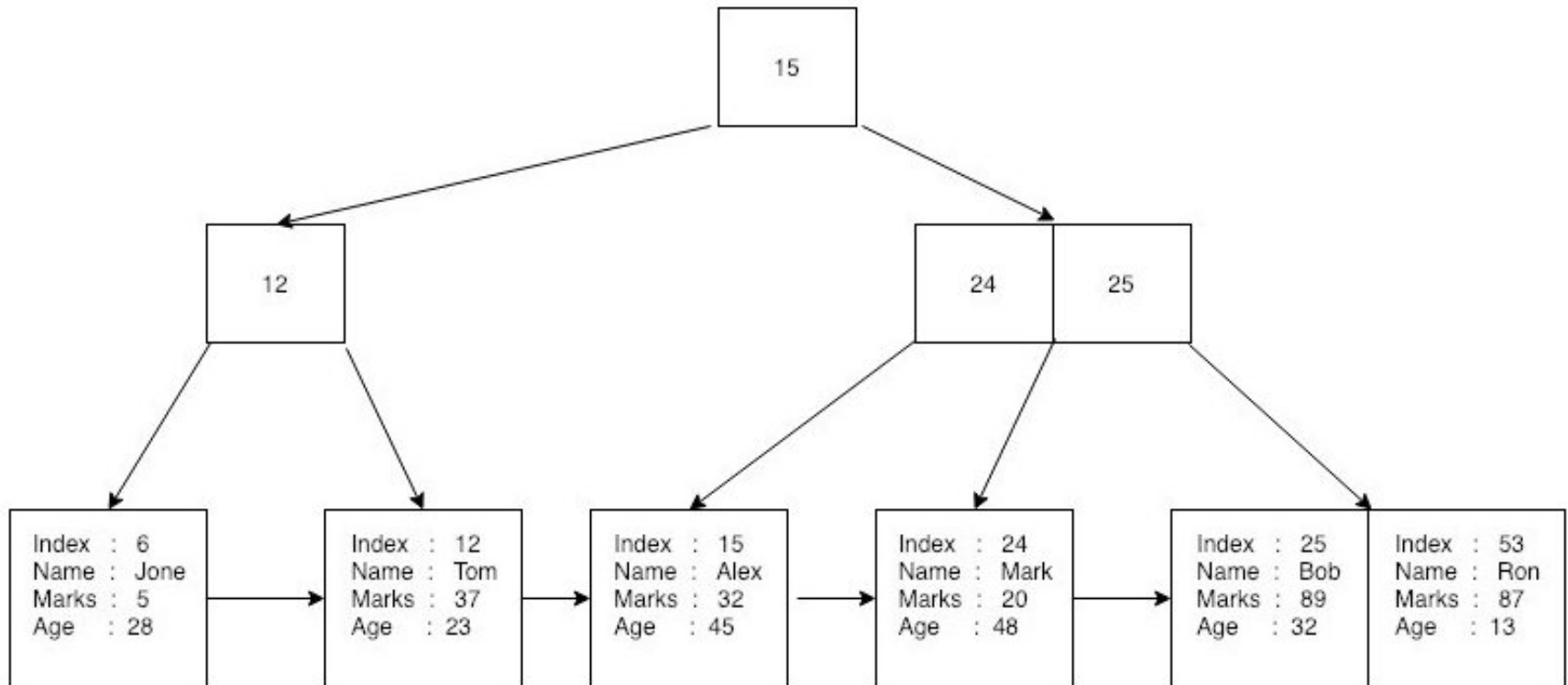
✓ *sqlite_master always rooted at page 1*

Альтернативные имена: sqlite_schema, sqlite_temp_master, sqlite_temp_master

rowid	Name	Marks	Age
6	Jone	5	28
12	Tom	37	23
15	Alex	32	45
24	Mark	20	48
25	Bob	89	32
53	Ron	87	13

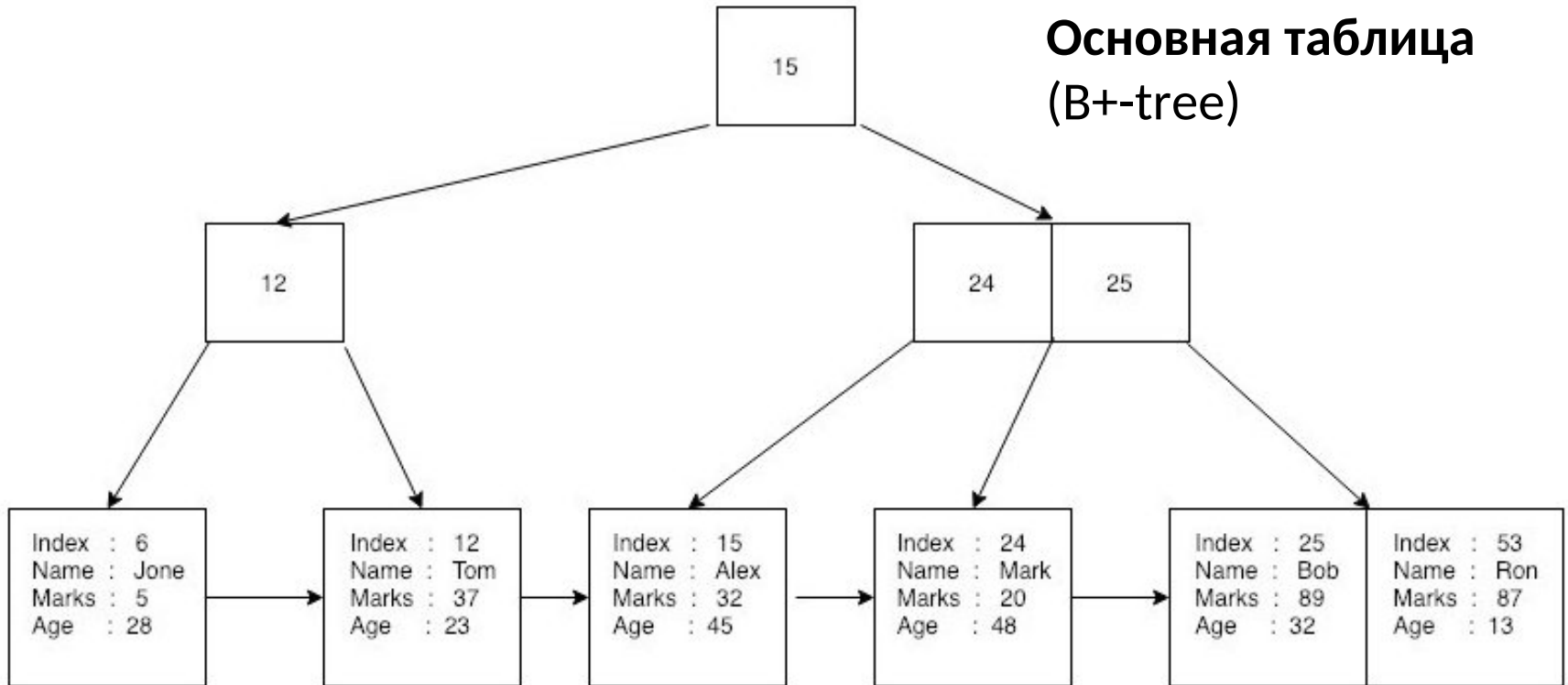
Поиск в таблице:

- Двоичный поиск по индексу
- Последовательный поиск по листьям

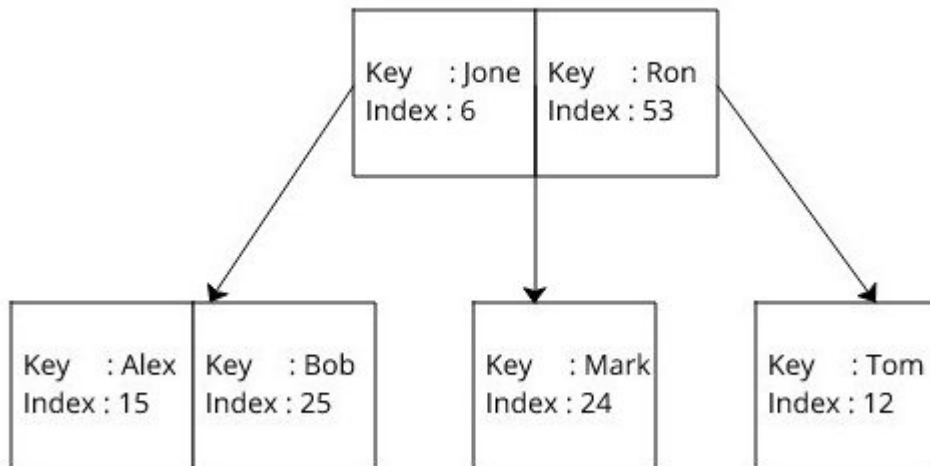


Каждый лист ссылается на следующий лист в дереве

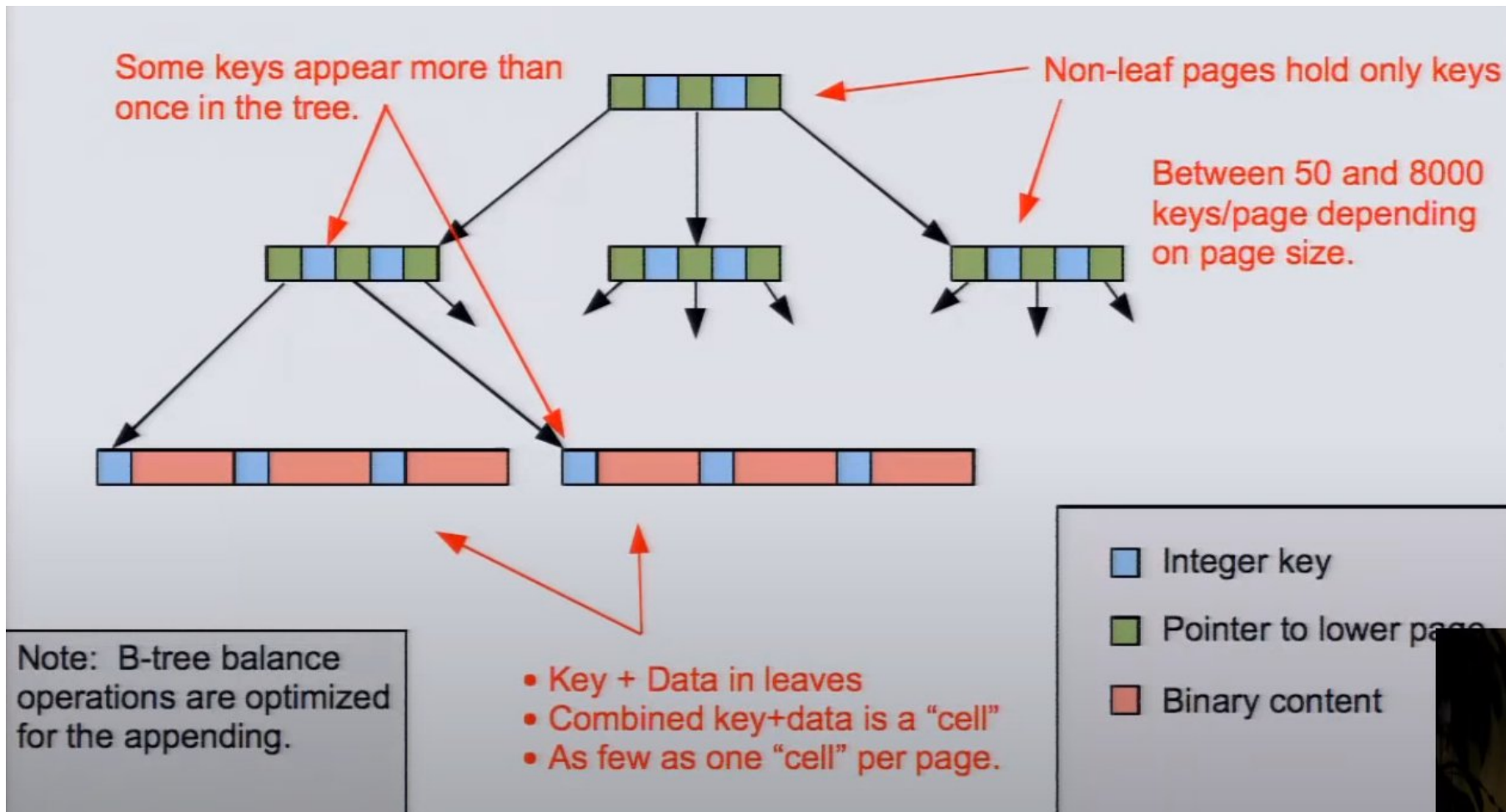
Основная таблица (B+-tree)



Индекс по имени (B-tree)

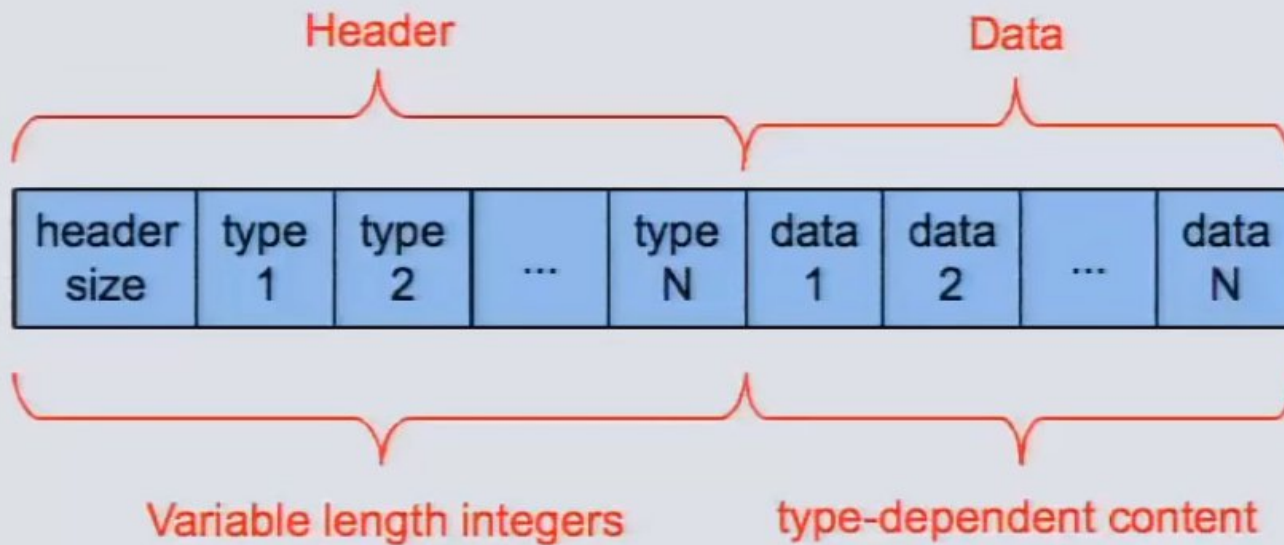


B-tree структура для страниц таблицы



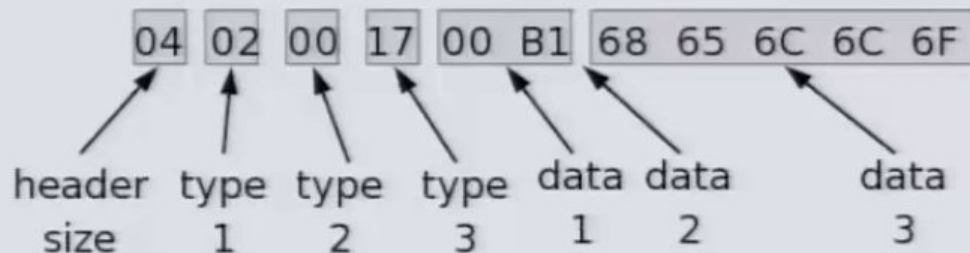
Узел = страница

Record Format



Record Format Example

```
CREATE TABLE t1(a,b,c);  
INSERT INTO t1 VALUES(177, NULL, 'hello');
```



- ✓ *Column datatypes are optional*
- ✓ *Datatypes are suggestions, not requirements.*