

# Занятие 4

## Индексы, партиционирование, MRP-системы, функции, триггеры

Бояр Владислав

Индексы

# Индекс

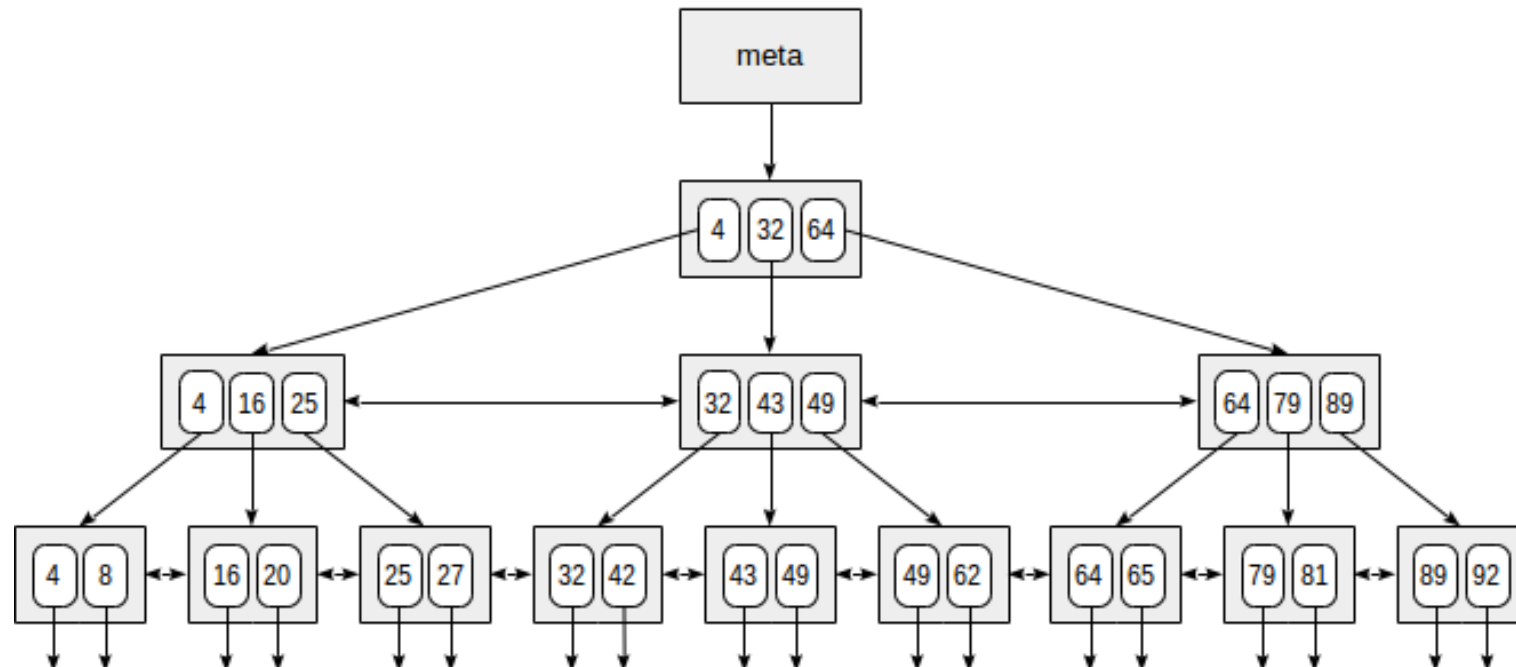
- **Индекс** - объект БД, предназначенный для ускорения доступа к данным;
- **Создание индекса** - одно из основных средств оптимизации запросов;
- Индекс, ускоряя доступ к данным, взамен требует затрат на своё содержание:
  - индекс занимает место не диске;
  - любая операция над проиндексированными данными (вставка, обновление, удаление) приводит к перестраиванию индекса в этой же транзакции.

# Индекс B-tree (бинарное дерево)

- Наиболее часто используемый тип индекса;
- Используется для данных, которые можно отсортировать;
- Позволяет производить поиск данных наподобие оглавления в телефонном справочнике;
- При любом способе сканирования (индексном, исключительно индексном, по битовой карте) метод доступа b-tree возвращает упорядоченные данные.

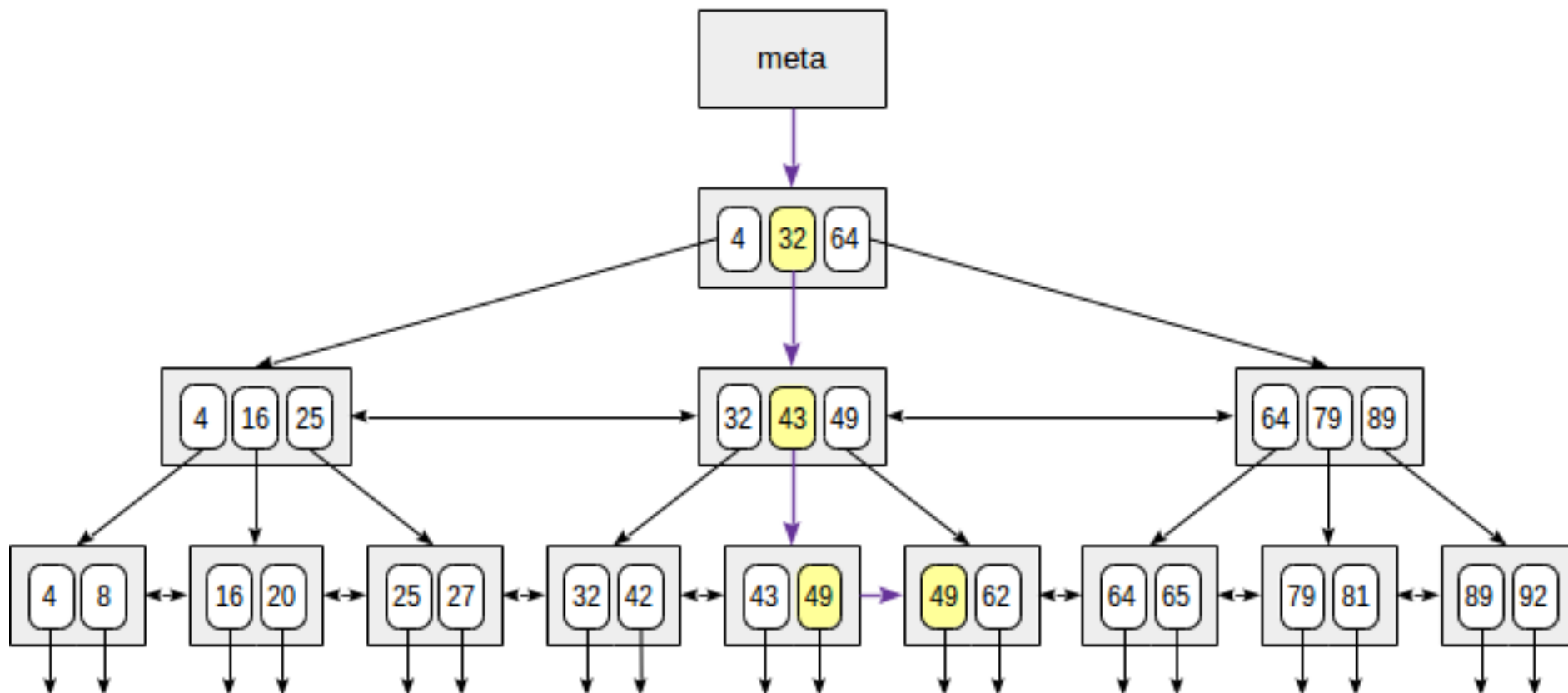
# Пример индекса по одному полю с целочисленными ключами

В самом начале файла находится метастраница, которая ссылается на корень индекса. Ниже корня расположены внутренние узлы; самый нижний ряд — листовые страницы. Стрелочки вниз символизируют ссылки из листовых узлов на строки таблицы (TID - **t**uple **i**dentifier).



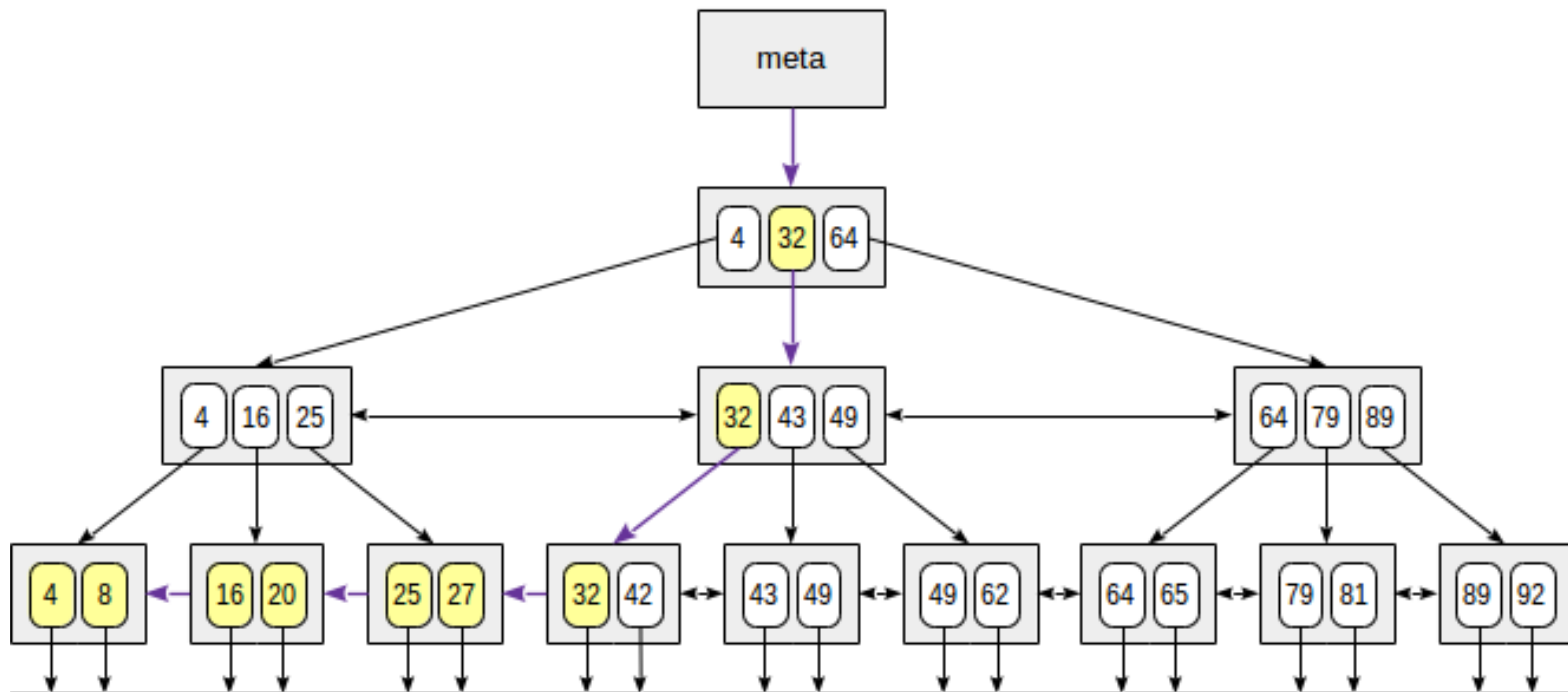
# Поиск по равенству

- Индексированный атрибут = 49



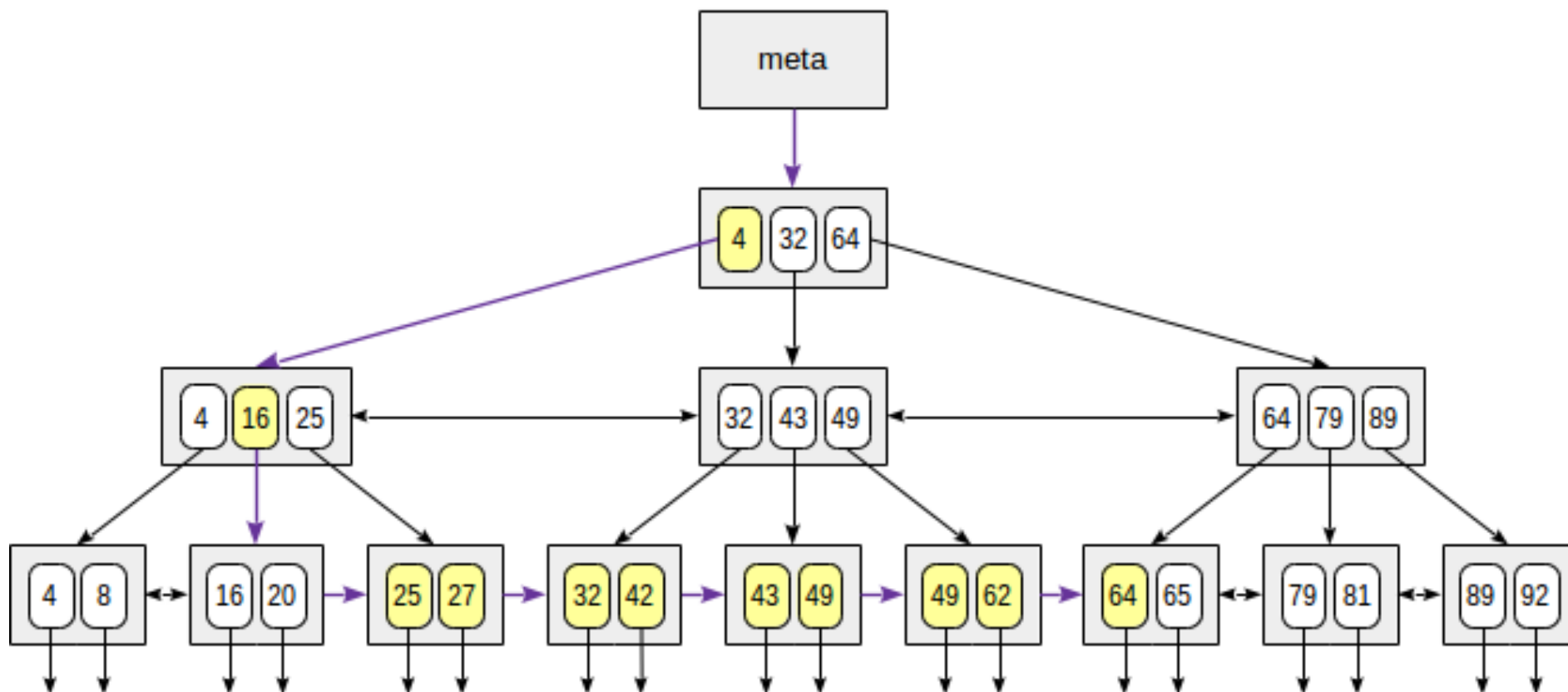
# Поиск по неравенству

- Индексированный атрибут  $\leq 35$



# Поиск по диапазону


- $23 \leq \text{Индексированный атрибут} \leq 64$





# Работа с индексами при масштабных изменениях данных

Если предстоит крупная вставка или обновление таблицы (более 100 тыс. записей), то оптимальнее:

- Удалить все индексы
  - Произвести вставку/обновление записей
  - Создать индексы на новых данных
- 

# Почему нельзя создать индексы на все атрибуты?

- Индексы занимают дисковое пространство (если сделать индексы по всем полям, то они занимают больше места, чем исходная таблица);
- Индексы утяжеляют операции над проиндексированными данными;

# Операции с индексом

Создание индекса:

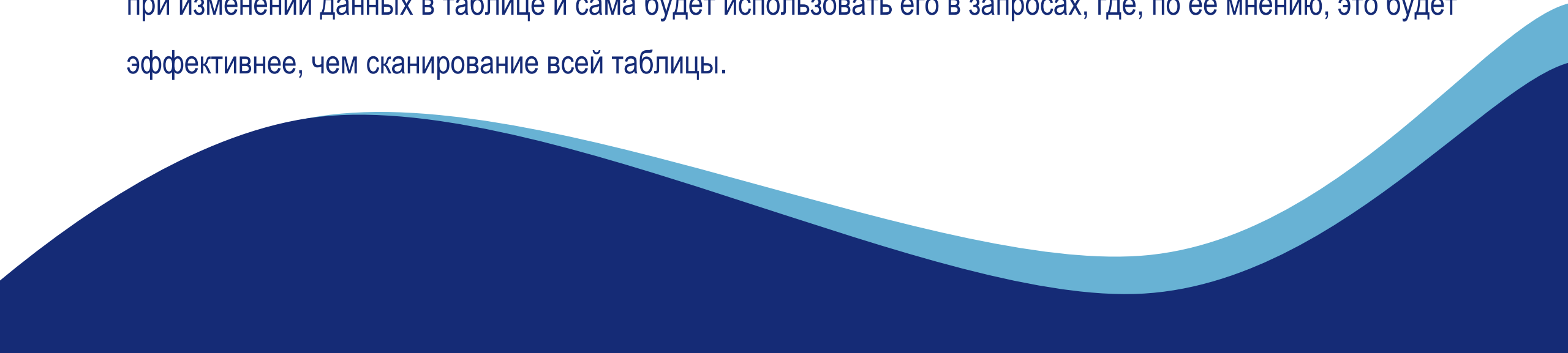
```
CREATE INDEX table_name_column_name_index ON table_name (column_name);
```

Название индекса может быть произвольным.

Удаление индекса:

```
DROP INDEX table_name_column_name_index;
```

Когда индекс создан, никакие дополнительные действия не требуются: система сама будет обновлять его при изменении данных в таблице и сама будет использовать его в запросах, где, по её мнению, это будет эффективнее, чем сканирование всей таблицы.




# Операции с индексом

Индекс можно создавать по нескольким полям:


```
CREATE INDEX table_1_c1c2_idx ON table_1 (c1, c2);
```

Актуально, если эти поля часто используются в операциях объединения или они часто одновременно участвуют в запросах в качестве фильтра:

```
SELECT c_name FROM table_1 WHERE c1 = ... AND c2 = ... ;
```



# Индекс и сортировки

- Помимо поиска строк для выдачи в результате запроса, индексы также могут применяться для сортировки строк в определённом порядке.
  - Это позволяет учесть предложение ORDER BY в запросе, не выполняя сортировку дополнительно.
  - Из всех типов индексов, которые поддерживает PostgreSQL, сортировать данные могут только В-деревья — индексы других типов возвращают строки в неопределённом, зависящем от реализации порядке.
  - Особый случай представляет ORDER BY в сочетании с LIMIT: при явной сортировке системе потребуется обработать все данные, чтобы выбрать первые n строк, но при наличии индекса, первые n строк можно получить сразу, не просматривая остальные вовсе.
- 

# Партиционирование



# Партиционирование

- Партиционирование (секционирование) – разбиение одной большой логической таблицы на несколько меньших физических таблиц (партиций / секций).
- Является способом ускорения доступа к данным.
- Применяется, когда:
  - таблица содержит много данных (десятки млн и более);
  - запросы чаще всего производятся к определённым данным (к примеру, с фильтрацией по дате);

# Партиционирование

stackoverflow.questions_2018		
Creation_date	Title	Tags
2018-03-01	How do I??	Android
2018-03-01	When Should?	Linux
2018-03-02	This is great!	Linux
2018-03-03	Can this?	C++
2018-03-02	Help!!	Android
2018-03-01	What does?	Android
2018-03-02	When does?	Android
2018-03-02	Can you help?	Linux
2018-03-02	What now?	Android
2018-03-03	Just learned!	SQL
2018-03-01	How does!	SQL



20180301	stackoverflow.questions_2018_partitioned		
	Creation_date	Title	Tags
	2018-03-01	How do I??	Android
	2018-03-01	When Should?	Linux
	2018-03-01	What does?	Android
20180302	2018-03-01	How does!	SQL
	Creation_date	Title	Tags
	2018-03-02	This is great!	Linux
	2018-03-02	Help!!	Android
	2018-03-02	When does?	Android
20180303	2018-03-02	Can you help?	Linux
	2018-03-02	What now?	Android
	Creation_date	Title	Tags
20180303	2018-03-03	Can this?	C++
	2018-03-03	Just learned!	SQL



# Партиционирование

- Партиционированная таблица — это виртуальная таблица, в которой нет строк.
- Партиции — это обычные таблицы, связанные с партиционированной.
- Каждая партиция хранит подмножество строк, определяемое значениями ключа партиционирования.
- Строки вставляются в соответствующую партицию на основе ключа партиционирования.
- Если в строке обновляется значение ключа партиции и он больше не соответствует значениям партиции, строка перемещается в другую партицию.

# Плюсы партиционирования

- Если данные запрашиваются из определённой партии (к примеру, за определённый месяц/год), то поиск данных в этой партии быстрее, чем поиск по индексу;
- Нет минусов индексов (дополнительное место на диске, тяжелые операции вставки и обновления);
- Добавить/удалить партию быстрее, чем производить последовательную вставку или удаление;

# Особенности партиционирования

- Не путать с PARTITION BY в оконных функциях 😊
- К партиции можно обращаться как к обычной таблице;
- Партиций не должно быть очень много (несколько тысяч) или очень мало (две);
- Первичные ключи в партиционированных таблицах не поддерживаются;
- На партиционированные таблицы не могут ссылаться внешние ключи;
- Преобразовать обычную таблицу в партиционированную и наоборот нельзя;
- В партиционированную таблицу можно добавить в качестве партиции обычную или партиционированную таблицу с данными;
- Можно удалить партицию из партиционированной таблицы и превратить её в отдельную таблицу.

# Виды партиционирования

- **RANGE (по диапазону)** - таблица партиционируется по «диапазонам», определённым по ключевому столбцу. Диапазоны не должны пересекаться друг с другом. Например, можно секционировать данные по диапазонам дат или по диапазонам идентификаторов.
- **LIST (по списку)** - таблица партиционируется с помощью списка, явно указывающего, какие значения ключа должны относиться к каждой партиции (например, список аэропортов).
- **HASH (по хэшу)** - таблица партиционируется по определённым модулям и остаткам, которые указываются для каждой партиции. Каждая партиция содержит строки, для которых значение ключа разбиения, разделённое на модуль, равняется заданному остатку. Используется, когда нужно равномерно распределить строки по партициям, а таблица не имеет подходящего ключа партиционирования.

# Партиционирование по диапазону

Создание партиционированной таблицы:

```
CREATE TABLE sales (  
  id int,  
  product text,  
  sale_date date  
) PARTITION BY RANGE (sale_date);
```

Создание партиций:

```
CREATE TABLE sales_y2024m01 PARTITION OF sales  
  FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');  
CREATE TABLE sales_y2024m02 PARTITION OF sales  
  FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');  
CREATE TABLE sales_default PARTITION OF sales  
  DEFAULT;
```

# Партиционирование по списку

Создание партиционированной таблицы:

```
CREATE TABLE books (  
  id int,  
  author text,  
  genre text  
) PARTITION BY LIST (genre);
```

Создание партиций:

```
CREATE TABLE books_novel PARTITION OF sales  
  FOR VALUES IN ('novel');  
CREATE TABLE books_detective PARTITION OF sales  
  FOR VALUES IN ('detective');  
CREATE TABLE sales_default PARTITION OF sales  
  DEFAULT;
```

# Партиционирование по хэшу

Создание партиционированной таблицы:

```
CREATE TABLE books (  
    code        char(5),  
    title       varchar(40),  
    delivery_date date,  
    genre       varchar(10)  
) PARTITION BY HASH (code);
```

Создание партиций:

```
CREATE TABLE books_part1 PARTITION OF books FOR VALUES WITH (MODULUS 5,  
REMAINDER 0);  
CREATE TABLE books_part2 PARTITION OF books FOR VALUES WITH (MODULUS 5,  
REMAINDER 1);  
CREATE TABLE books_part3 PARTITION OF books FOR VALUES WITH (MODULUS 5,  
REMAINDER 2);  
CREATE TABLE books_part4 PARTITION OF books FOR VALUES WITH (MODULUS 5,  
REMAINDER 3);  
CREATE TABLE books_part5 PARTITION OF books FOR VALUES WITH (MODULUS 5,  
REMAINDER 4);
```

# Обслуживание партиций

Удаление партиции:

```
DROP TABLE sales_y2024m01;
```

Убрать партицию из главной таблицы, но сохранить возможность обращаться к ней как к самостоятельной таблице:

```
ALTER TABLE sales DETACH PARTITION sales_y2024m01;
```






# Партиционирование по диапазону (GreenPlum)

```
CREATE TABLE book_order
(id INT,
book_id INT,
client_id INT,
book_count SMALLINT,
order_date DATE)
WITH (appendoptimized=true, orientation=row, compresstype=ZLIB,
compresslevel=5)
DISTRIBUTED BY(id)
PARTITION BY RANGE(order_date)
(START(date '2022-01-01') INCLUSIVE
END(date '2023-01-01') EXCLUSIVE
EVERY(INTERVAL '1 month'),
DEFAULT PARTITION other);
```

# Massive parallel processing (MPP)

# Massive parallel processing (MPP)

**MPP – это:**

- массово-параллельная архитектура;
  - класс архитектур параллельных вычислительных систем;
  - особенность архитектуры заключается в том, что память физически разделена;
  - несколько машин, скоординировано выполняющих одну и ту же задачу.
- 
- A decorative graphic at the bottom of the slide consisting of two overlapping wavy lines. The bottom-most line is a dark blue, and the line just above it is a lighter, medium blue. They create a layered, wave-like effect across the width of the slide.

# Примеры MPP-СУБД

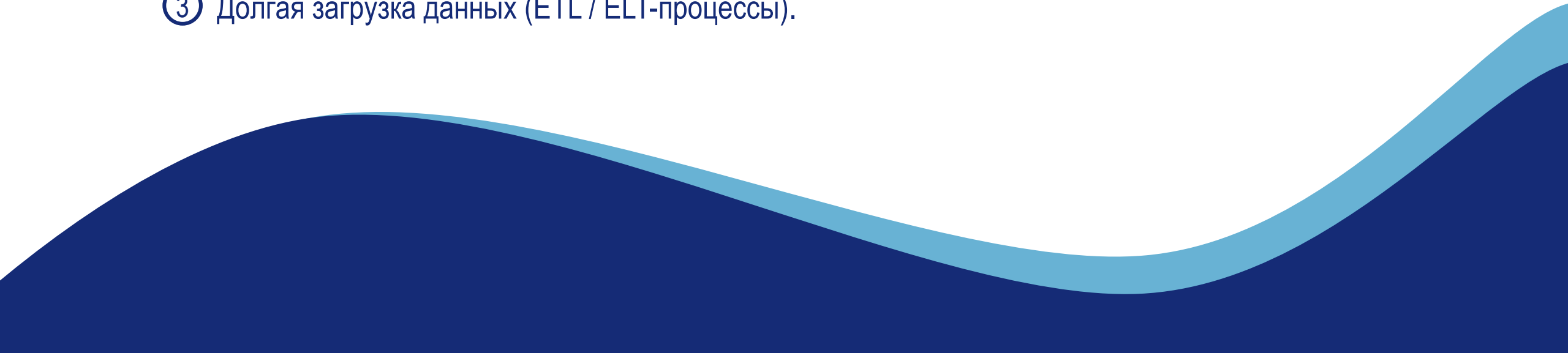


VERTICA



teradata.

# Предпосылки перехода к MPP-СУБД

- ① Большой объем хранилища данных (~ от 1 ТБ) и его быстрый рост;
  - ② Низкая производительность хранилища и отчётности;
  - ③ Долгая загрузка данных (ETL / ELT-процессы).
- 

# MPP-СУБД

## Преимущества:

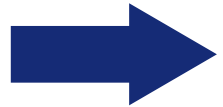
- + Быстрая обработка больших объёмов данных
- + Простая горизонтальная масштабируемость (нет необходимости переходить на более мощную машину, достаточно докупить несколько аналогичных)
- + Отказоустойчивость (зеркалирование, резервирование)

## Недостатки:

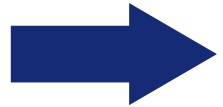
- Высокие требования к инфраструктуре (сеть, цп, память, диски)
- Медленно работает для большого количества простых запросов (для таких задач лучше использовать, к примеру, ClickHouse)



# GreenPlum



несколько экземпляров (инстансов, instance) PostgreSQL, которые работают как единая СУБД



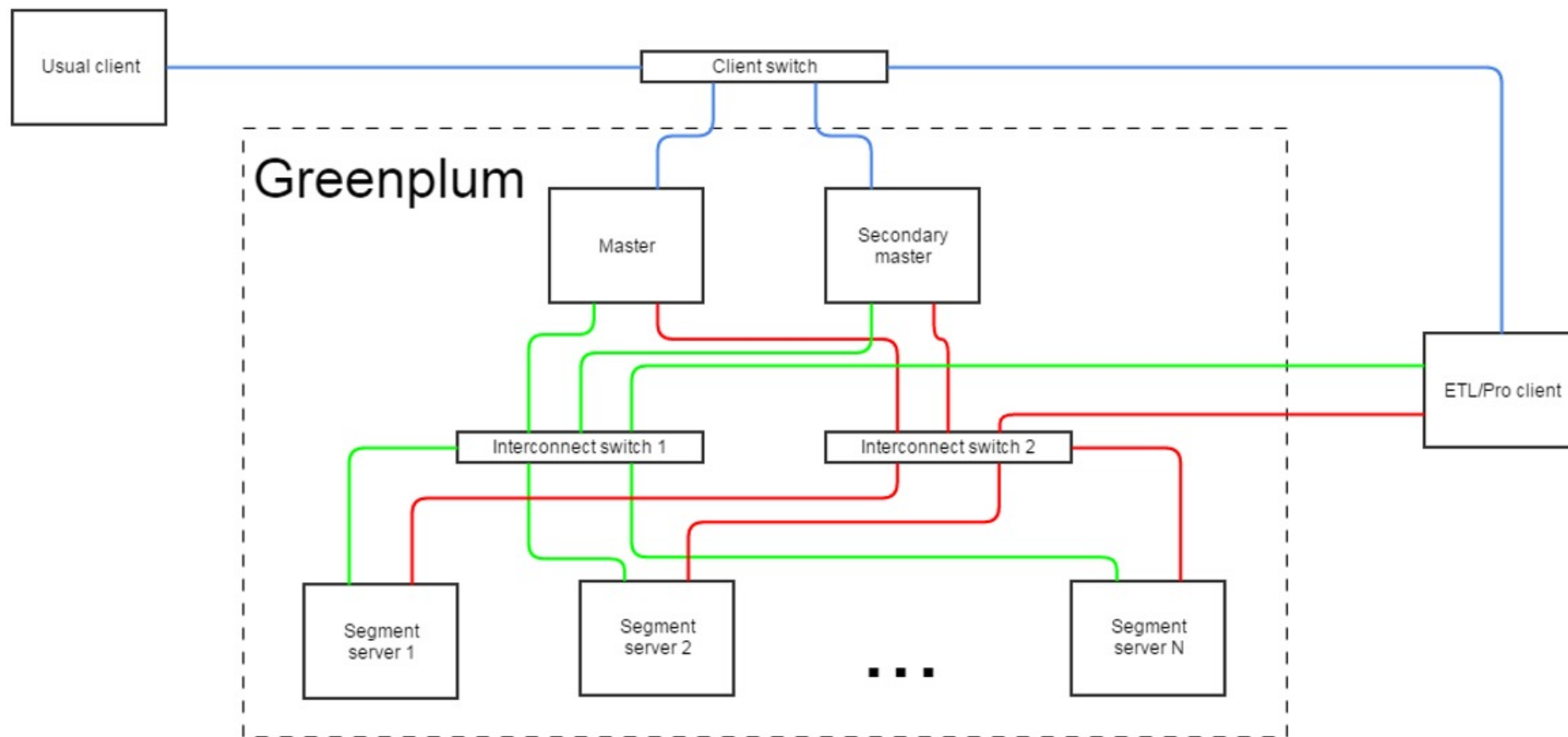
связь между отдельными инстансами PostgreSQL осуществляется на сетевом уровне с помощью быстрых сетей (interconnect)



# Особенности GreenPlum

- Горизонтальное масштабирование;
- Поддерживает и строковое и колоночное хранение данных;
- SQL-запросы выполняются параллельно;
- Полуавтоматическое партиционирование данных;
- Конечные пользователи взаимодействуют с GreenPlum, как с обычной СУБД, несмотря на сложную архитектуру
- Концепция Shared Nothing (без разделения ресурсов):
  - Узлы кластера не разделяют ресурсы между собой;
  - Каждый узел имеет собственные ресурсы: ОС, память, диски.

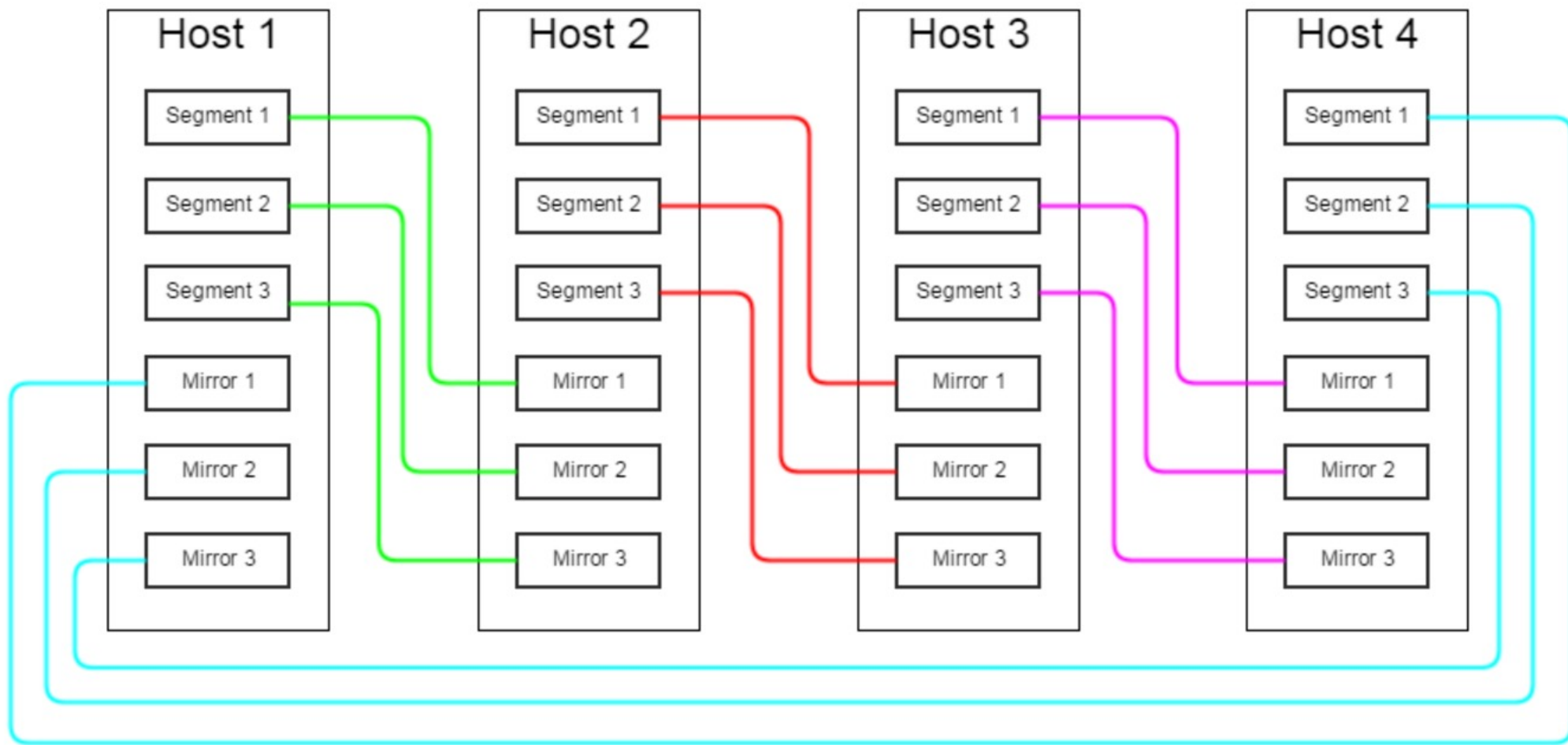
# Архитектура GreenPlum



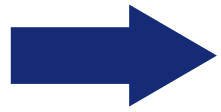
# Архитектура GreenPlum

- **Master instance** — входная точка для пользователей (экземпляр БД, к которому подключаются клиенты). Координатор работы других экземпляров БД.
- **Secondary master instance** — резервный инстанс, используется при отказе мастера;
- **Primary segment instance** — хранит и обрабатывает данные.
- **Mirror segment instance** — инстанс, который автоматически включается в работу при отказе primary segment instance.

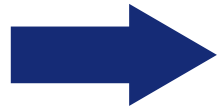
# Распределение Primary и Mirror сегментов по узлам (зеркалирование)



# Хранение данных в GreenPlum




Каждая таблица разделена на  $N+1$  таблиц, где  $N$  – число сегментов кластера (+1 это таблица на мастере, в ней нет данных)



На каждом сегменте хранится  $1/N$  строк таблицы. Данные разбиваются по заданному ключу (например, по дате).

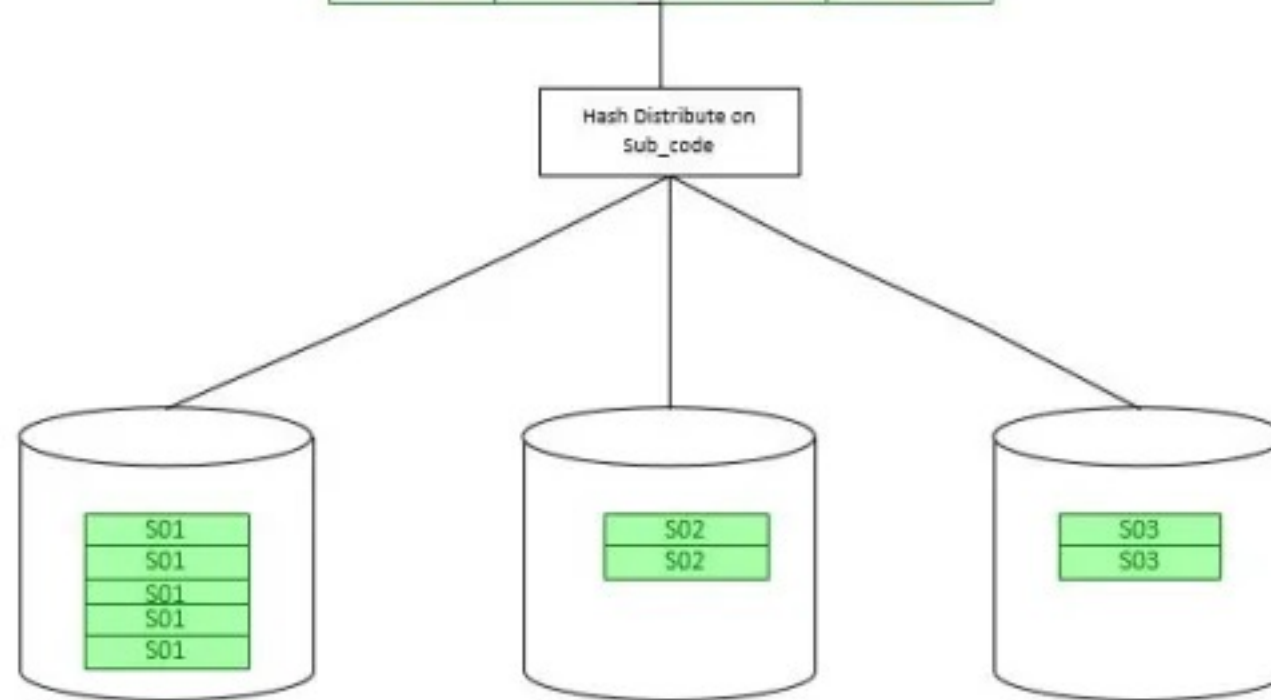
Данный ключ также называется ключом распределения или ключом дистрибьюции (distribution key).

# Распределение данных

- Распределение данных (distribution) является одной из ключевых концепций GreenPlum.
  - Определяет хранение данных каждой таблицы на различных сегментах кластера.
  - Чем более равномерно данные распределены между сегментами, тем выше производительность всего кластера.
  - В идеальном случае данные должны иметь равномерное распределение между сегментами.
- 

# Пример распределения данных

Student_id	Sub_code	Name	Marks
101	S01	Student1	50
102	S02	Student2	60
103	S03	Student3	55
104	S01	Student4	70
105	S01	Student5	59
106	S02	Student6	71
107	S03	Student7	83
108	S01	Student8	91
109	S01	Student9	21



# Способы распределения

Тип	Описание	Пример
DISTRIBUTED BY (<column(-s)>)	Хеш-распределение. Конкретный сегмент выбирается на основе хешей, которые рассчитываются по полям, указанным в скобках. Рекомендуется использовать для таблиц, имеющих первичные ключи (PRIMARY KEY) или столбцы с уникальными значениями (UNIQUE).	CREATE TABLE test_distr_key (a INT PRIMARY KEY, b TEXT) DISTRIBUTED BY (a);
DISTRIBUTED REPLICATED	Распределение данных, при котором копия таблицы сохраняется на каждом сегменте кластера. Рекомендуется для небольших таблиц (например, таблиц-справочников). Позволяет избежать любых перемещений данных (Motion) при JOIN-запросах.	CREATE TABLE test_distr_replicated (a TEXT, b TEXT) DISTRIBUTED REPLICATED;
DISTRIBUTED RANDOMLY	Случайное распределение данных. Поскольку система выбирает сегменты случайным образом, равномерность распределения данных между ними не гарантируется. Рекомендуется для случаев, когда в таблицах нет столбцов с уникальными значениями, а размер таблиц достаточно большой.	CREATE TABLE test_distr_random (a TEXT, b INT) DISTRIBUTED RANDOMLY;



# Типы таблиц

- **HEAP** - используется по умолчанию и рекомендуются для OLTP (Online Transaction Processing)-нагрузок. Heap-таблицы являются оптимальным выбором в случае частого обновления данных после первоначальной загрузки, а также подходят для однострочных операций INSERT, UPDATE и DELETE.
- **Append-optimized** - предпочтительны для OLAP (Online Analytical Processing)-нагрузок. Они также отлично подходят для пакетной загрузки данных. Их рекомендуется использовать при редких обновлениях данных, когда в системе преобладают read-only запросы. В отличие от Heap-таблиц (в которых возможна только строковая ориентация данных — row-oriented), таблицы append-optimized поддерживают две формы ориентации данных:
  - Строковая (row-oriented). Эта модель хранения данных рекомендуется для запросов, в которых одновременно извлекаются все либо большая часть столбцов таблицы.
  - Колоночная (column-oriented). Эта модель подходит для вычислений на базе небольшого набора столбцов таблицы. Ее также рекомендуется использовать при регулярных обновлениях незначительной части столбцов.


# Типы таблиц

Синтаксис:

```
CREATE TABLE test_a0 (a int, b text)  
WITH (appendoptimized=true)  
DISTRIBUTED BY (a);
```

```
CREATE TABLE test_a0 (a int, b text)  
WITH (appendoptimized=false) -- Или ничего не указывать  
DISTRIBUTED BY (a);
```

# Метод хранения. Head или Append-Optimized

- Используйте Head для таблиц и партиций, где операции INSERT, UPDATE, DELETE проводятся итеративно (батчами или одиночными записями).
  - Используйте Append-Optimized для таблиц, которые были загружены разово и обновляется редко и большими батчами.
  - Избегайте для Append-Optimized единичных операций INSERT, UPDATE.
- 

# Сжатие данных

Для оптимизации и уменьшения размера данных можно использовать сжатие или компрессию данных.

Пример определения сжатия данных для таблицы:

```
CREATE TABLE compressed_table (a int, b text)  
WITH (appendoptimized=true,  
orientation=column,  
compresstype=zlib,  
compresslevel=5)  
DISTRIBUTED BY (a);
```

- `compresstype` — тип сжатия данных. Возможные значения: ZLIB, ZSTD и RLE\_TYPE. Значения не чувствительны к регистру. По умолчанию используется значение none, при котором сжатие не применяется.
- `compresslevel` — уровень сжатия данных. ZLIB(1-9). ZSTD(1-19). RLT\_TYPE (1-4).

# Общие рекомендации по распределению

## Что делать?

- Явно задавайте способ и ключ распределения при создании таблицы, в том числе для временных таблиц;
- Используйте RANDOMLY-распределение для небольших таблиц, где нет хороших кандидатов на ключ распределения из одного атрибута.

## Что будет, если нарушить рекомендацию?

- По умолчанию ключом распределения будет назначен первый атрибут таблицы;
- Неравномерное распределение данных по сегментам;
- Долго вычисляется хэш для ключа распределения из 2 и более атрибутов.

# Избегайте использования в качестве ключа распределения

Что делать?

- Атрибут, который будет часто использоваться в WHERE;
- Поле с типом данных date, timestamp;
- Поля, содержащие множество значений NULL.

Что будет, если нарушить рекомендацию?

- Данные будут распределены неравномерно;
- При фильтрации по ключу распределения участвует только часть сегментов;
- Долго вычисляется хэш.

# Старайтесь использовать в качестве ключа распределения

Что делать?

- UUID / Integer;
- Поле, которое часто будет использоваться при объединении таблиц (JOIN).

Что будет, если нарушить рекомендацию?

- Долго вычисляется хэш по другим типам данных (string);
- Выполнения JOIN потребует перераспределения строк между сегментами и неоптимального перемещения данных.

# Особенности модели данных

Архитектура MPP систем отличается от классических СУБД, где данные высоко нормализованы.

- Используйте денормализованную схему данных «звезда» (star) или «снежинка» (snowflake) с большими фактовыми таблицами и небольшими таблицами измерений.
- Используйте одинаковые и «быстрые» типы данных для полей, по которым происходит объединение таблиц.






# Распределение данных.

- Задавайте распределение явным образом (по колонке или randomly), не используйте распределение по умолчанию;
- Используйте распределение по одной колонке;
- Не используйте в качестве ключа распределения поля, которые:
  - используются в WHERE;
  - используются в качестве ключа партиционирования.

# Партиционирование данных.

- Используйте партиционирование только для больших таблиц;
  - Ключ партии должен использоваться в WHERE;
  - Предпочтительнее использовать RANGE, а не LIST;
  - Чаще всего партиция используется по колонке с датой;
  - Задавайте партицию по умолчанию, но старайтесь не использовать её;
  - Избегайте вложенного партиционирования.
- 

# Ориентация. Строки или таблицы.

## Строки (row)

- Сценарии с итеративными транзакциями, где данные часто обновляются или добавляются;
- Широкие SELECT.

## Колоночные (column)

- Проводятся регулярные обновления отдельных полей без изменения других данных строки;
- Узкие SELECT.



Вам необходимо проверить равномерность распределения по сегментам данных таблицы products. Выберите запрос, который поможет это сделать.

- a) **SELECT gp\_segment\_id, count(\*) FROM products GROUP BY gp;**
  - b) **SELECT gp\_segment\_id, count(\*) FROM products GROUP BY gp\_segment\_id;**
  - c) **SELECT products, count(\*) FROM gp\_segment\_id GROUP BY gp;**
  - d) **SELECT products, count(\*) FROM gp\_segment\_id GROUP BY products;**
- 

Необходимо создать таблицу `salse` из трех полей с партиционированием по диапазону значений `дат` и интервалами один месяц. Выберите подходящее разделение на партии.

a) **PARTITION BY LENGTH;**

b) **PARTITION BY RANGE;**

c) **PARTITION BY TABLE;**

d) **PARTITION BY COLUMN;**

# Функции

# Функция

- SQL-функции выполняют произвольный список операторов SQL;
- Тело SQL-функции должно представлять собой список SQL-операторов, разделённых точкой с запятой. Точка с запятой после последнего оператора может отсутствовать.
- Любой набор команд на языке SQL можно скомпоновать вместе и обозначить как функцию.
- Помимо запросов SELECT, эти команды могут включать запросы, изменяющие данные (INSERT, UPDATE и DELETE), а также другие SQL-команды.

```
CREATE FUNCTION one() RETURNS integer AS $$  
SELECT 1 AS result;  
$$ LANGUAGE SQL;
```

# Синтаксис функции (пример)

**CREATE OR REPLACE FUNCTION**

schema\_name.function\_name (p\_table **text**, p\_schema **text**)

**RETURNS void**

**LANGUAGE** plpgsql -- PL/pgSQL – процедурный язык SQL

**AS \$\$**

**declare**

-- Объявление переменных

v\_var\_name\_1 **text**;

v\_var\_name\_2 **int**;

**begin**

-- Тело функции

**end;**

**\$\$**

**EXECUTE ON ANY;**



# Синтаксис функции (пример)

-- способ присвоения значения переменной напрямую


```
v_var_name_1 := 'some_text';
```

-- способ присвоения значения переменной через запрос

```
select column_1  
from schema_name.table_name  
into v_var_name_2;
```

-- Оператор условия

```
if v_var_name_1 not in ('text_1', 'text_2') then ...;  
end if;
```



# Синтаксис функции (пример)

-- Цикл

```
for v_date_range in ...  
loop
```

```
    if var_name_1 not in ('text_1', 'text_2') then  
        v_var_name_2 := 1;  
    end if;
```

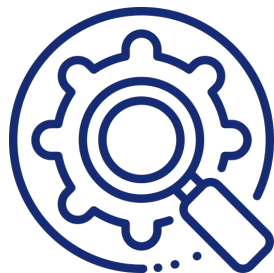
-- Выполнение sql-запросов в теле функции с использованием переменных

```
v_sql := 'select count(*) from ' || p_schema || '.' || p_table;  
execute v_sql;
```

```
end loop;
```



Практика



Напишем функцию и триггер, которые будут сохранять в отдельном справочнике список студентов.

1. Создадим таблицу, которая будет хранить список студентов;
2. Напишем функцию, которая будет:
  1. Проверять, существует ли уже такой студент в списке студентов;
  2. Если нет, то вставлять новое значение в список;
3. Напишем триггер, которые будет вызывать функцию из пункта 2, при осуществлении всех операций вставки и обновления данных (Insert / Update).