

DWH

Лекция №4:

СУБД массивно-параллельной обработки
(MPP)

Особенности работы MPP-систем

Оптимизация запросов в MPP системах



В прошлой
лекции



Концепция Data Vault

Data Vault – набор уникально связанных нормализованных таблиц, содержащих детальные данные, отслеживающих историю изменений и предназначенных для поддержки одной или нескольких функциональных областей бизнеса.

Дизайн Data Vault сосредоточен вокруг функциональных областей бизнеса.

Основные сущности, используемые в Data Vault:

- Хаб (Hub) хранит сущности, соответствующие реальным объектам бизнес-процесса.
- Связь (Link) обеспечивает транзакционную интеграцию между Хабами (связи между сущностями).
- Сателлит (Satellite) предоставляет контекст первичного ключа Хаба (атрибуты, описания)

Якорное моделирование

Якорное моделирование — это технология моделирования гибкой базы данных, подходящая для информации, которая со временем изменяется как по структуре, так и по содержанию.

Якорная модель фокусируется на изменениях.

Основные сущности, используемые в Data Vault:

- **Anchor** (Якорь) — это существительное, объект реального мира.
- **Attribute** (Атрибут) — это таблица для хранения свойства, атрибута объекта
- **Tie** (Связь) — это таблица для хранения связей между объектами.
- **Knot** (Узел) — таблица-состояние, справочник.

Принципы работы РСУБД

• • • •

Массив

Двумерный массив — простейшая структура данных.
Таблица может быть представлена в виде массива:

| | Колонка 0 | Колонка 1 | Колонка 2 | Колонка 3 |
|-------|-----------|-----------|-----------|-----------|
| Ряд 0 | Robert | 55 | manager | USA |
| Ряд 1 | Alex | 23 | developer | GER |
| Ряд 2 | Jennifer | 35 | manager | FRA |
| Ряд 3 | Robert | 45 | CEO | USA |
| Ряд 4 | Charles | 32 | DBA | UK |
| ... | | | | |
| Ряд П | Alice | 34 | developer | ITA |

Сложность поиска $O(N)$

Индексирование

Индекс – особая структура данных, в которых каждая запись состоит из двух значений – отсортированных значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы.

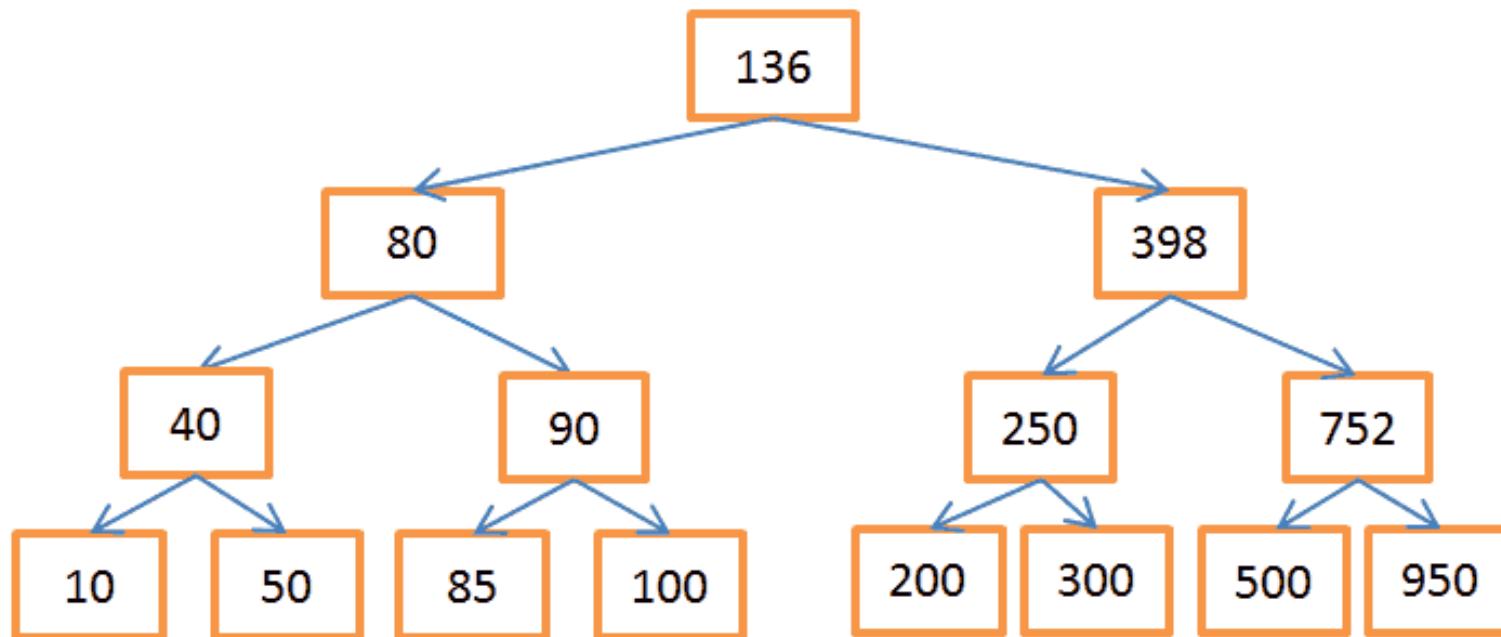
- Быстрый поиск по условию WHERE
- Объединение таблиц посредством JOIN
- Поиск min-max значений из выборки
- Сортировка и группировка по таблице
- Извлечение данных не из таблицы, а из индексного файла

| index | org_id | org_name |
|-------|--------|--|
| 2 | 5 | Выставочный центр СО РАН |
| 3 | 3 | ГНЦ Вектор |
| 4 | 16 | ГПНТБ |
| 5 | 14 | Геофизическая служба СО РАН |
| 12 | 12 | Издательство СО РАН |
| 13 | 2 | Институт автоматики и электрометрии СО РАН |
| 14 | 17 | Институт археологии и этнографии СО РАН |
| 16 | 4 | Институт вычислительных технологий СО РАН |
| 17 | 13 | Институт геологии и минералогии СО РАН |
| 19 | 19 | Институт горного дела СО РАН |

Бинарное дерево поиска

Бинарное дерево поиска — это дерево, в котором ключ в каждом узле должен быть:

- Больше, чем любой из ключей в ветке слева;
- Меньше, чем любой из ключей в ветке справа

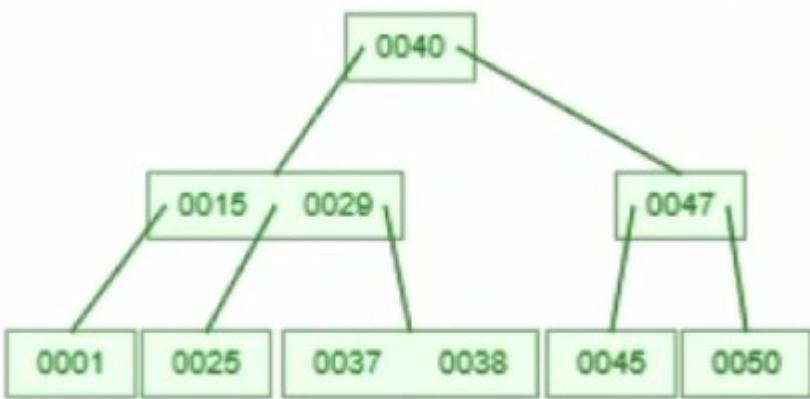


Сложность поиска $O(\log(N))$

В-дерево и индекс БД

При построении В-дерева применяется фактор t , который называется минимальной степенью. Корень содержит от 1 до $2t-1$ ключей. Любой другой узел содержит от $t-1$ до $2t-1$ ключей.

- У корня должно быть хотя бы 2 потомка.
- Все листья на одном уровне (более сильное требование, чем простая сбалансированность).
- Перестройка дерева происходит снизу вверх

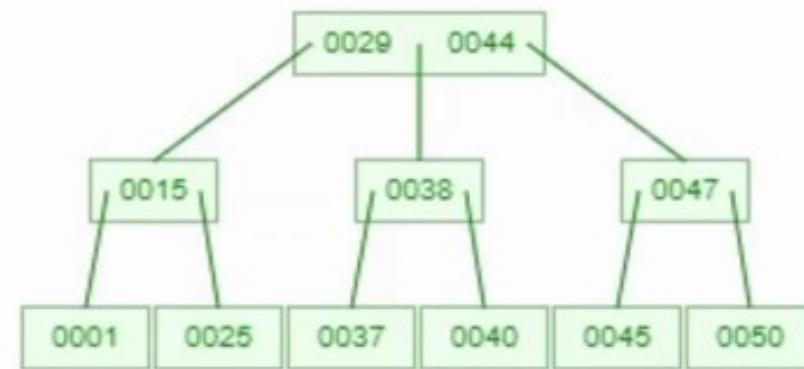


Операция поиска выполняется за время $O(t \log t n)$, где t – минимальная степень. Важно здесь, что дисковых операций мы совершаём всего лишь $O(\log t n)$

B-Trees

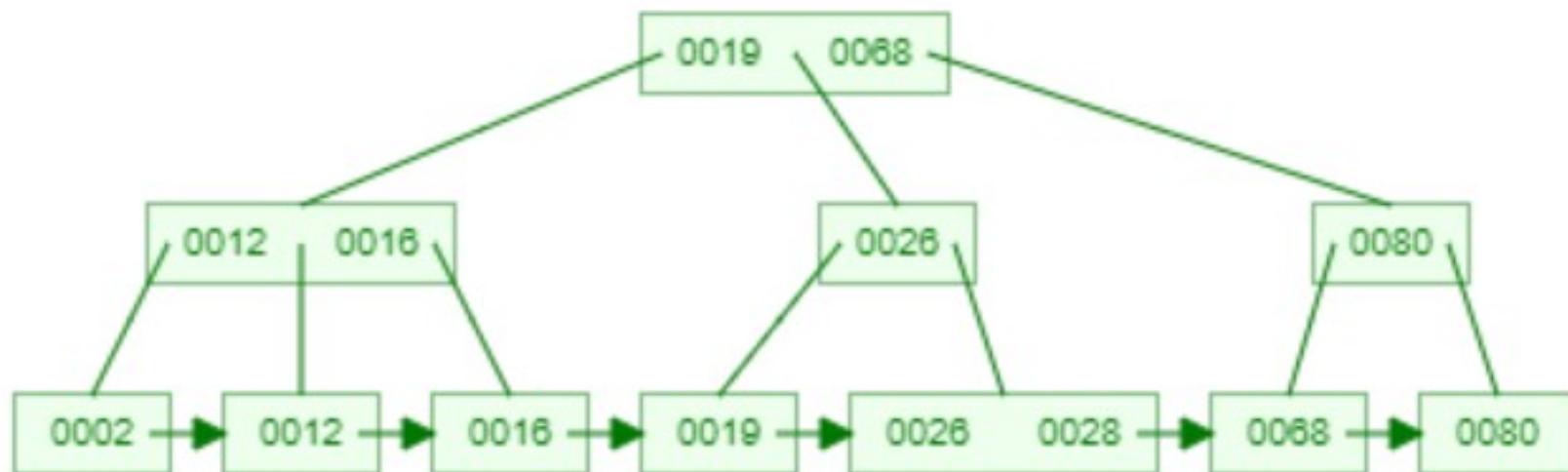
 Insert Delete 45 Find Print Clear

- Max. Degree = 3
- Max. Degree = 4
- Max. Degree = 5
- Max. Degree = 6
- Max. Degree = 7

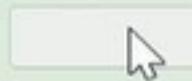


B⁺-дерево

B⁺-дерево. Информацию (record pointer) хранят только листья, а все остальные узлы предназначены для оптимизации поиска по дереву. Листья имеют ссылку на правого брата



B⁺ Trees



Insert

Delete

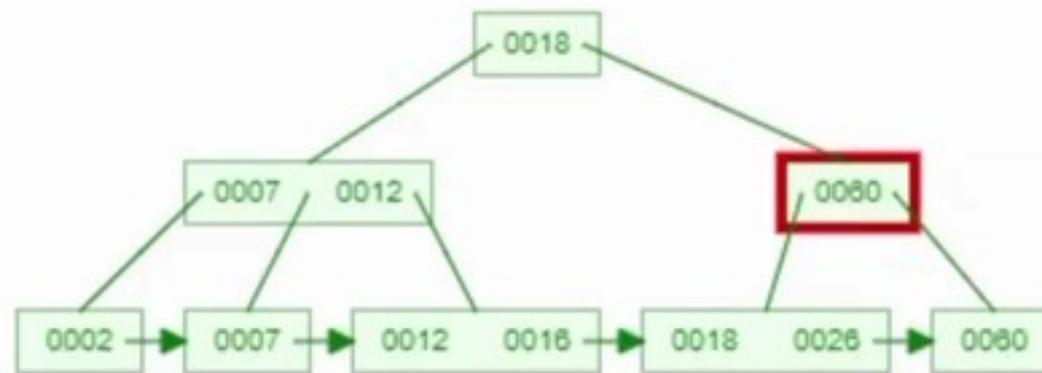
Find

Print

Clear

- Max. Degree = 3
- Max. Degree = 4
- Max. Degree = 5
- Max. Degree = 6
- Max. Degree = 7

Inserting 0019

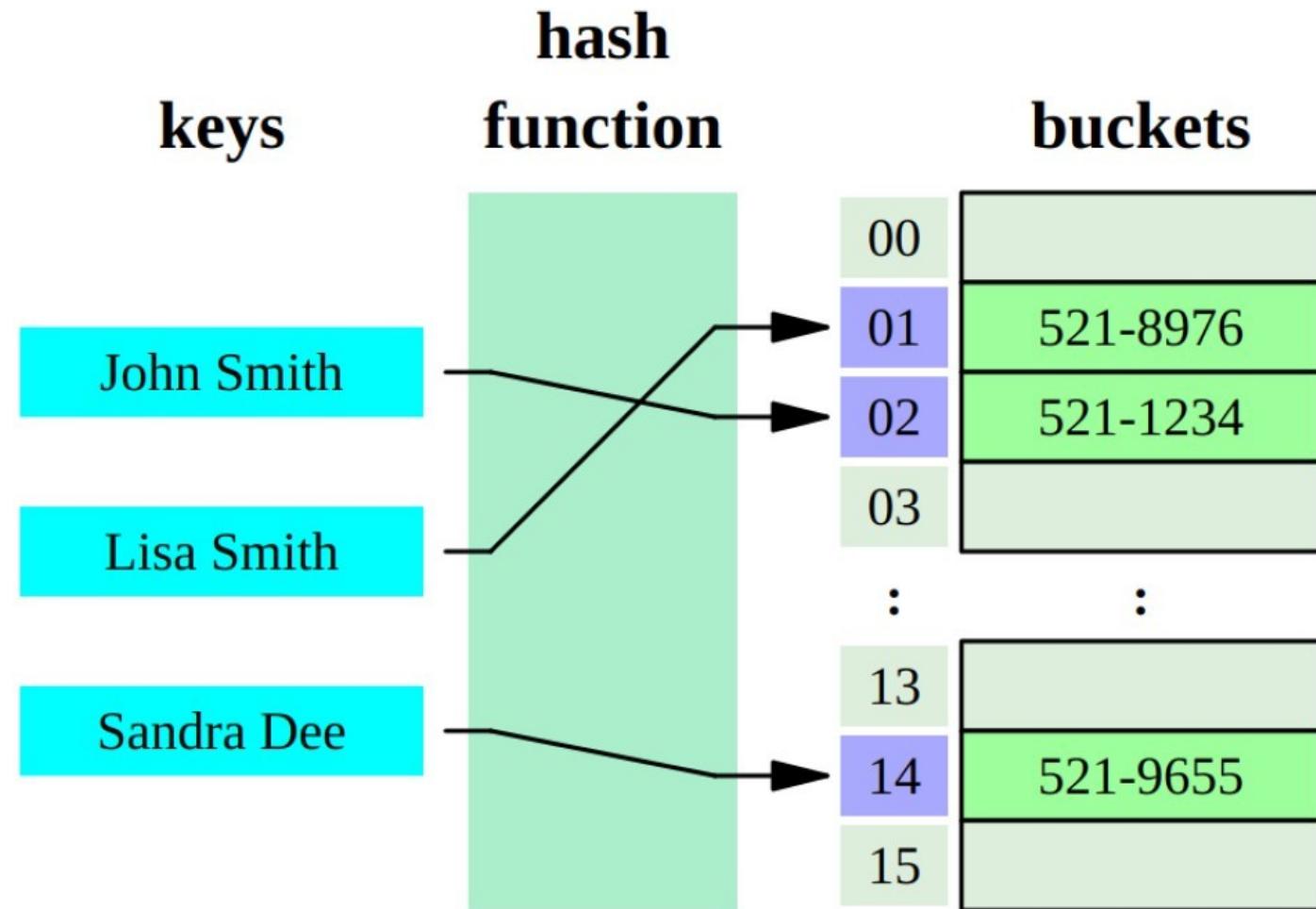


Хеш-таблица

Физическая реализация интерфейса ассоциативного массива (ключ-значение).

Для поиска используется хеш-функция, которая позволяет преобразовать ключ в набор адресов, по которым можно найти связанное значение.

В идеале на каждое значение ключа хеш-функция возвращает уникальный адрес значения. Такая функция называется **совершенной**. К сожалению, так бывает редко, поэтому приходится иметь дело с **коллизиями**.



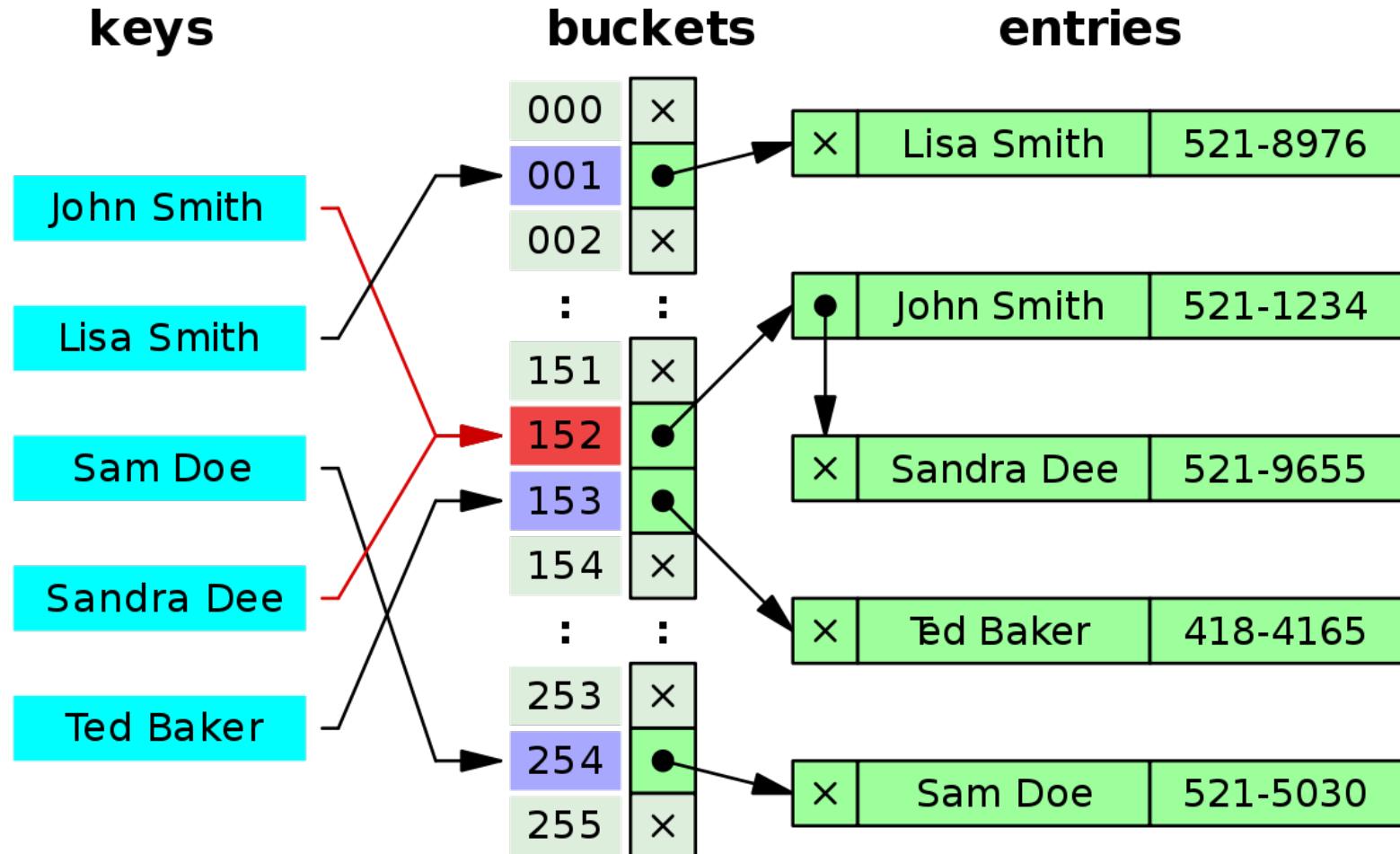
Сложность поиска $O(1)$

Хеш-таблица — коллизии

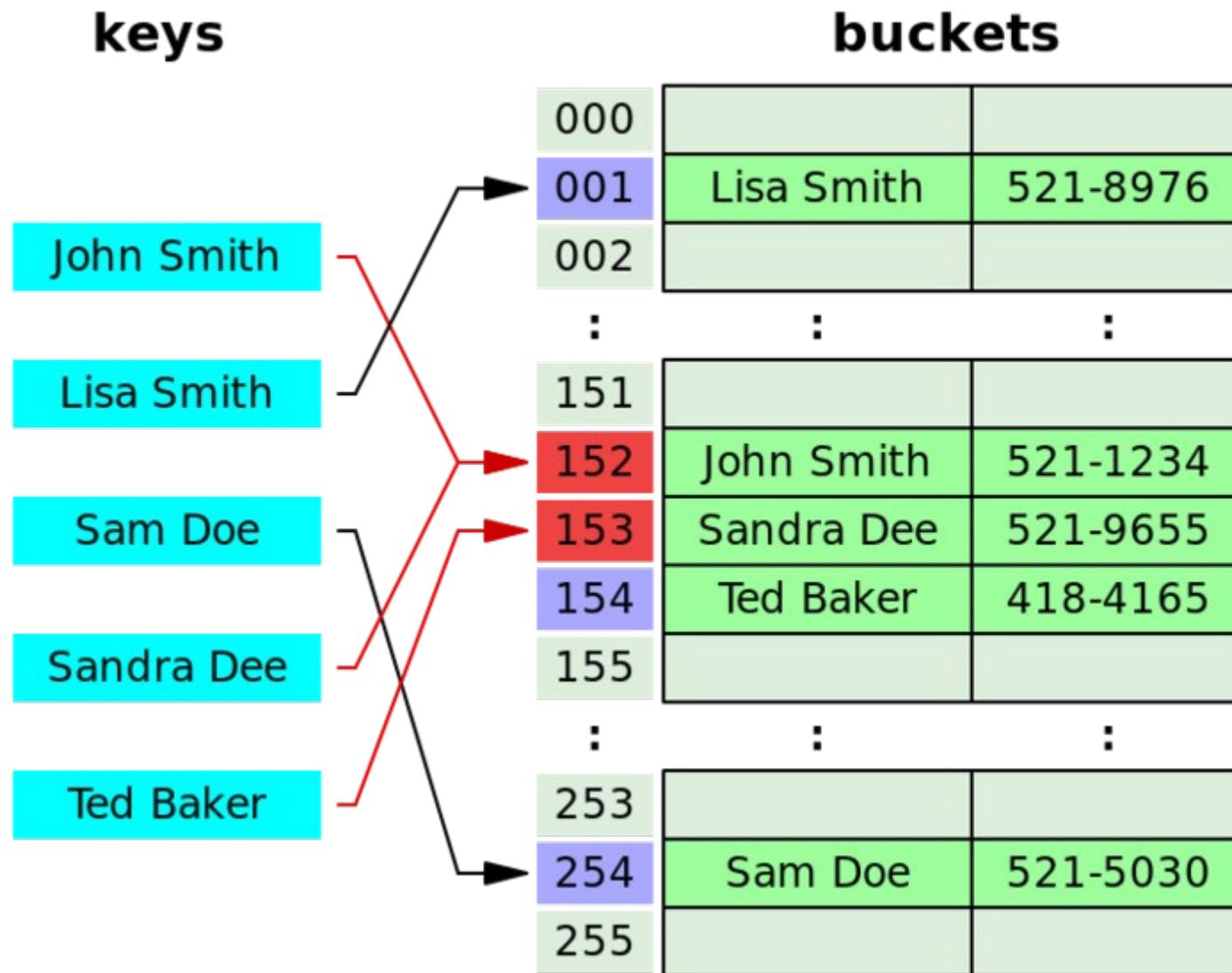
Коллизия хеш-функции возникает, когда два различных входных блока данных x и y для хеш-функции H возвращают одинаковое хеш-значение, т. е. $H(x) = H(y)$. Два основных способа разрешения коллизий в хеш-таблице:

- **Связные списки** — значения с одинаковым хеш-кодом объединяются в список. Элементы таблицы: (ключ, список) Элементы списка: (ключ, значение)
- **Открытая адресация** — в случае коллизии очередной ключ связывается с первым ещё свободным адресом в таблице. То есть после вычисления уже занятого хеш-кода происходит смещение указателя по некоторому алгоритму пробирования (поиск нового адреса зависит от реализации)

Метод списков



Открытая адресация



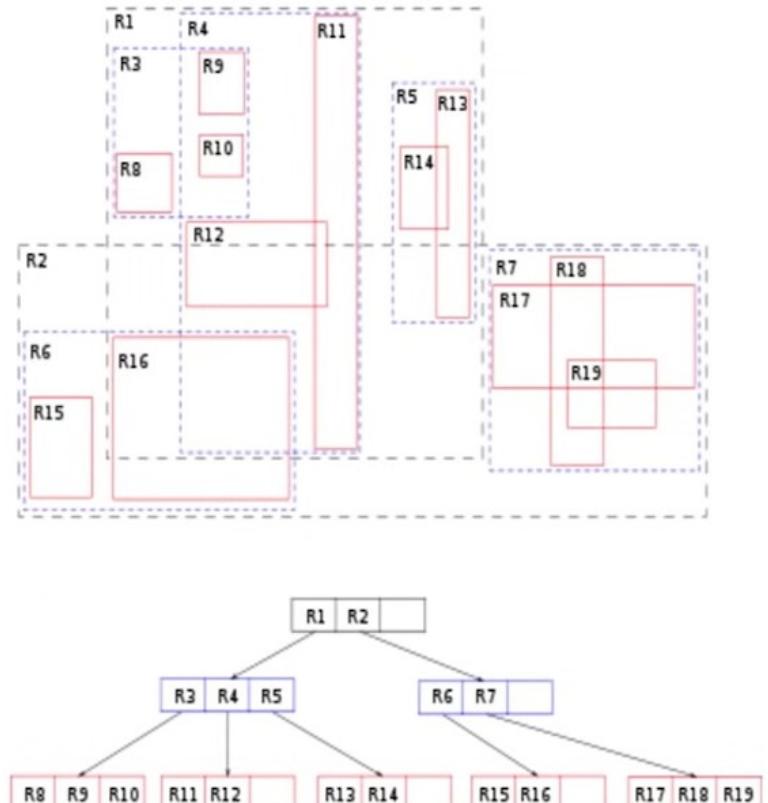
Какие типы индексов еще существуют?

По алгоритмам:

- GIST-индекс – для индексирования пространственных данных;
- GIN-индекс – инвертированный индекс для вложенных структур;
- битовый индекс – компактно хранит информацию о соответствии кортежа какому-либо условию.

По покрытию:

- кластерный индекс – тип индекса, который сортирует строки данных в таблице по их ключевым значениям;
- покрывающий индекс – индекс по всем полям конкретного запроса;
- частичный индекс – сохраняет только определенные строки таблицы;
- функциональный индекс – сохраняет значение функции от значения колонки.



Способы обращений к таблице СУБД

- **Full (Index) Scan.** СУБД последовательно считывает файл таблицы или индекса. Для дисковой подсистемы индекс читать дешевле, чем таблицу.
- **Index Unique Scan.** Используется в тех случаях, когда вам нужно получить из уникального индекса одно конкретное значение. Возвращает не более одной строки.
- **Index Range Scan.** Срабатывает, например, когда вы используете предикаты вида WHERE AGE > 20 AND AGE < 40. Очевидно, для этого нужно иметь индекс по полю AGE.
- **Access by ROWID.** Применяется в случаях, когда нам однозначно известен внутренний идентификатор интересующей нас строки таблицы (ROWID)
- **Index fast full scan**
- **Index skip scan**

Способы обращений к таблице СУБД

- EXPLAIN SELECT * FROM film;

```
QUERY PLAN
▶ Seq Scan on film  (cost=0.00..64.00 rows=1000 width=384)
```

- EXPLAIN SELECT * FROM film WHERE film_id = 100;

```
QUERY PLAN
▶ Index Scan using film_pkey on film  (cost=0.28..8.29 rows=1 width=384)
  Index Cond: (film_id = 100)
```

Физические реализации JOIN'ов

Вспомним, какие типы джойнов семантически определяет язык SQL:

- Inner — внутренний;
- Outer (Left / Right / Full) — внешние;
- Cross — декартово произведение.

SQL о том, **что** делает написанный джойн.

Но нас интересует, **как** этот джойн физически реализуется в СУБД.

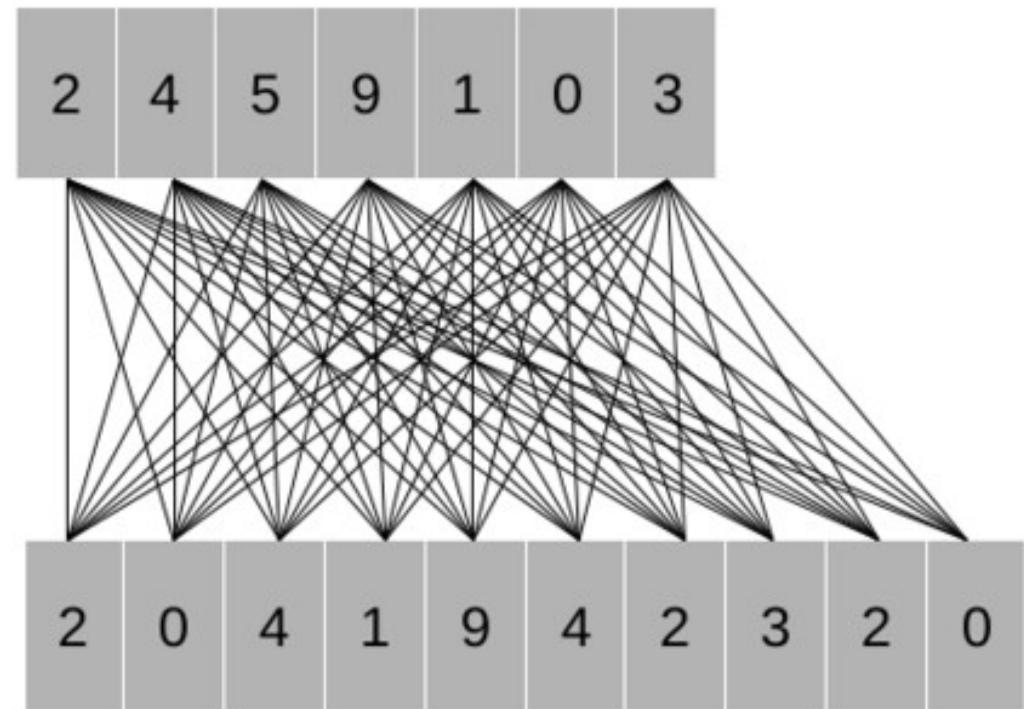
Три основные стратегии джойна таблиц в СУБД:

- Nested Loops Join — соединение вложенными циклами;
- Hash Join — соединение через хеш-таблицу;
- Merge Join — соединение слиянием.

Nested Loops Join

Если вся внутренняя таблица не влезает в память, можно подключить к работе диск:

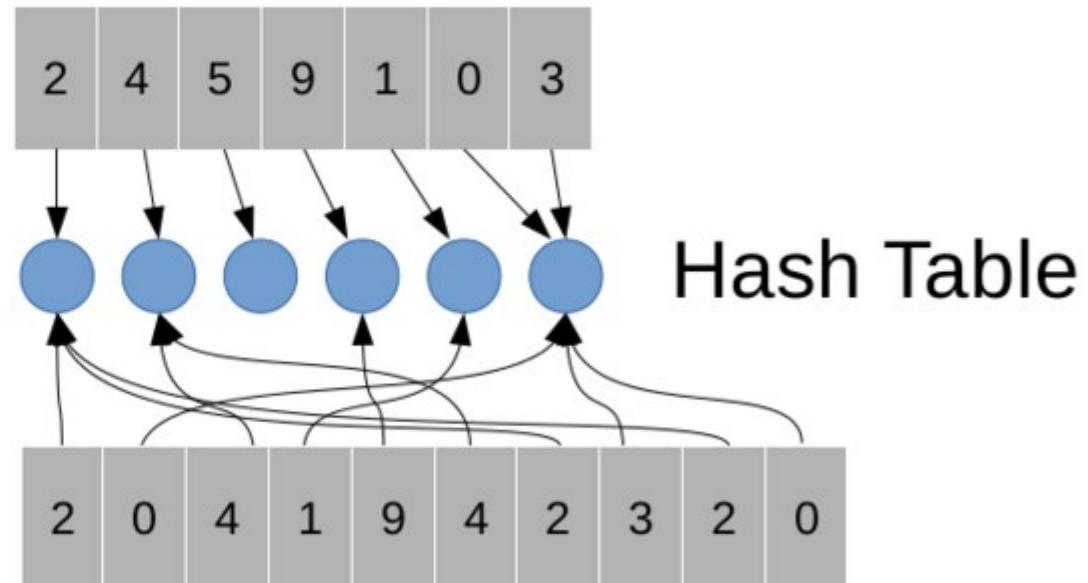
1. Вместо чтения обеих таблиц целиком оничитываются группами строк (batch), при этом в памяти держится по одной группе из каждой таблицы.
2. Строки из этих групп сравниваются между собой, а найденные совпадения сохраняются отдельно.
3. Затем в память подгружаются новые группы и цикл повторяется



Hash Match Join

1. В памяти создаётся хэш-таблица по всем элементам внутренней таблицы.
2. Один за другим считаются все элементы из внешней таблицы.
3. Для каждого элемента вычисляется хэш (используя ту же хеш-функцию), чтобы найти соответствующий блок во внутренней таблице.
4. Элементы из блока сравниваются с элементами из внешней таблицы.

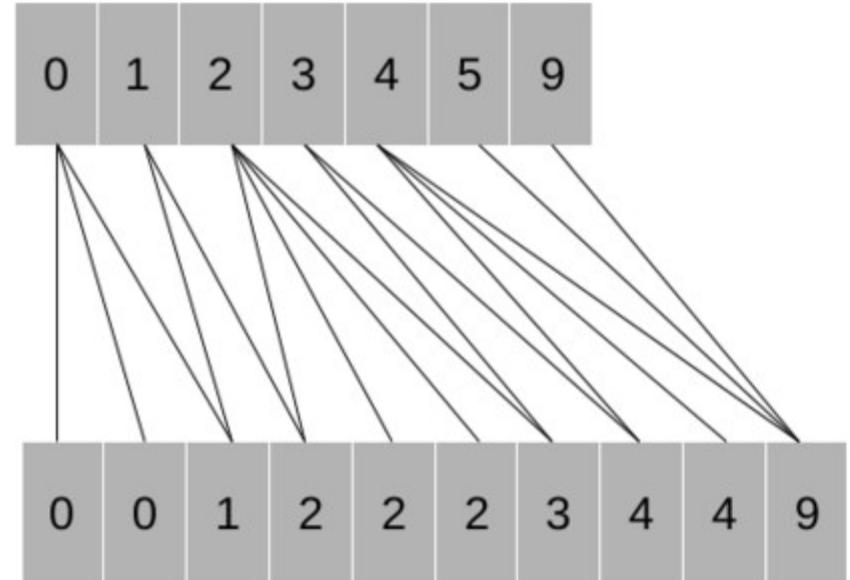
Hash Join



Merge Join

1. Сравниваются два текущих элемента обеих таблиц.
2. Если они равны, то соединённая запись добавляется к результирующей выборке. Далее указатели в обеих таблицах перемещаются к следующим элементам.
3. Если они не равны, то для таблицы с меньшим значением указатель смещается к следующему значению.
4. Шаги 1-3 повторяются, пока не закончатся элементы одной из таблиц

Merge Join



Физические реализации JOIN'ов

- EXPLAIN SELECT

```
f.film_id, title, name, category_name
FROM film AS f
    INNER JOIN film_category fc ON fc.film_id = f.film_id
    INNER JOIN category c ON c.category_id = fc.category_id
ORDER BY title;
```

| QUERY PLAN | |
|------------|---|
| ▶ | Sort (cost=169.51..172.01 rows=1000 width=87) |
| | Sort Key: f.title |
| -> | Hash Join (cost=41.93..119.68 rows=1000 width=87) |
| | Hash Cond: (f.film_id = fc.film_id) |
| -> | Seq Scan on film f (cost=0.00..64.00 rows=1000 width=19) |
| -> | Hash (cost=29.43..29.43 rows=1000 width=70) |
| -> | Hash Join (cost=1.36..29.43 rows=1000 width=70) |
| | Hash Cond: (fc.category_id = c.category_id) |
| -> | Seq Scan on film_category fc (cost=0.00..16.00 rows=1000 width=4) |
| -> | Hash (cost=1.16..1.16 rows=16 width=72) |
| -> | Seq Scan on category c (cost=0.00..1.16 rows=16 width=72) |

Физические реализации JOIN'ов

- EXPLAIN SELECT

```
f.film_id, title, name, fc.category_id
FROM films AS f
INNER JOIN film_category fc ON fc.film_id = f.film_id
ORDER BY F.film_id;
```

QUERY PLAN

```
Merge Join
  Merge Cond: (f.film_id = fc.film_id)
    -> Index Scan using film_pkey on film f
    -> Index Scan using film_category_pkey on film_category fc
(4 rows)
```

Выбор алгоритма

Объём доступной памяти. Её может не хватить для хранения всей хеш-таблицы.

Размер соединяемых таблиц. Если одна таблица большая, а вторая маленькая, иногда быстрее использовать nested loops join, поскольку создание хеш-таблицы может оказаться дороже. Для двух больших таблиц вложенные циклы работают очень плохо, поскольку эта операция очень интенсивно использует процессор, а сложность алгоритма растёт экспоненциально от числа строк.

Сортировка входных данных. Для отсортированных данных используется merge join. Такой вход можно получить при джойне по индексированным ключам (B-tree \times B-tree) или в результате каких-то подзапросов

Выбор алгоритма

- **Распределение данных (skew factor).** Если данные неравномерно распределены по условию джойна (например, джойн по фамилиям, но часто встречаются однофамильцы), то использовать hash join не лучший вариант. Иначе хэш-функция будет постоянно заниматься разбором коллизий, очень сильно теряя в производительности.
- **Семантический тип join'a.** Некоторые стратегии join'a не работают в конкретных случаях:
 - условие джойна может быть по равенству (Table1.ColA = Table2.ColB), по неравенству или по более сложному предикату;
 - внутренний, внешний, декартово произведение или self-join.

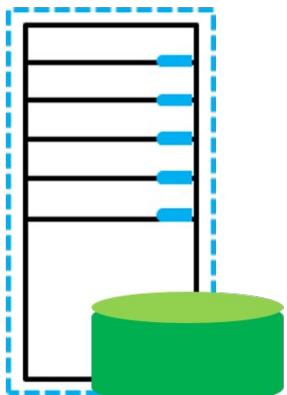
Принципы параллельной обработки

• • • •

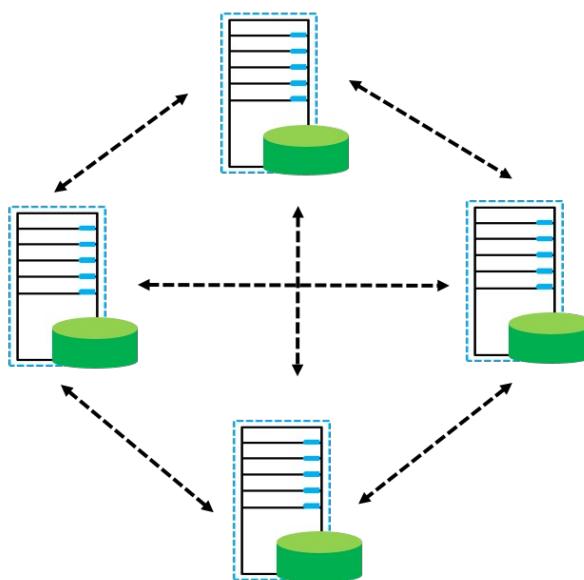
SMP vs. MPP

SMP vs. MPP

Symmetric Multiprocessing



Massively Parallel Processing



Архитектура распределённых систем

Распределённая система управления базой данных работает с несколькими вычислительными узлами единой компьютерной сети, что позволяет ей (если это возможно) распараллеливать выполнение некоторых операций с целью повышения общей производительности обработки.

Важно понимать, что речь идёт о единой БД с единой моделью данных и единым интерфейсом доступа к этим данным, а не о произвольной распределённой файловой системе, разбросанной по нескольким узлам сети

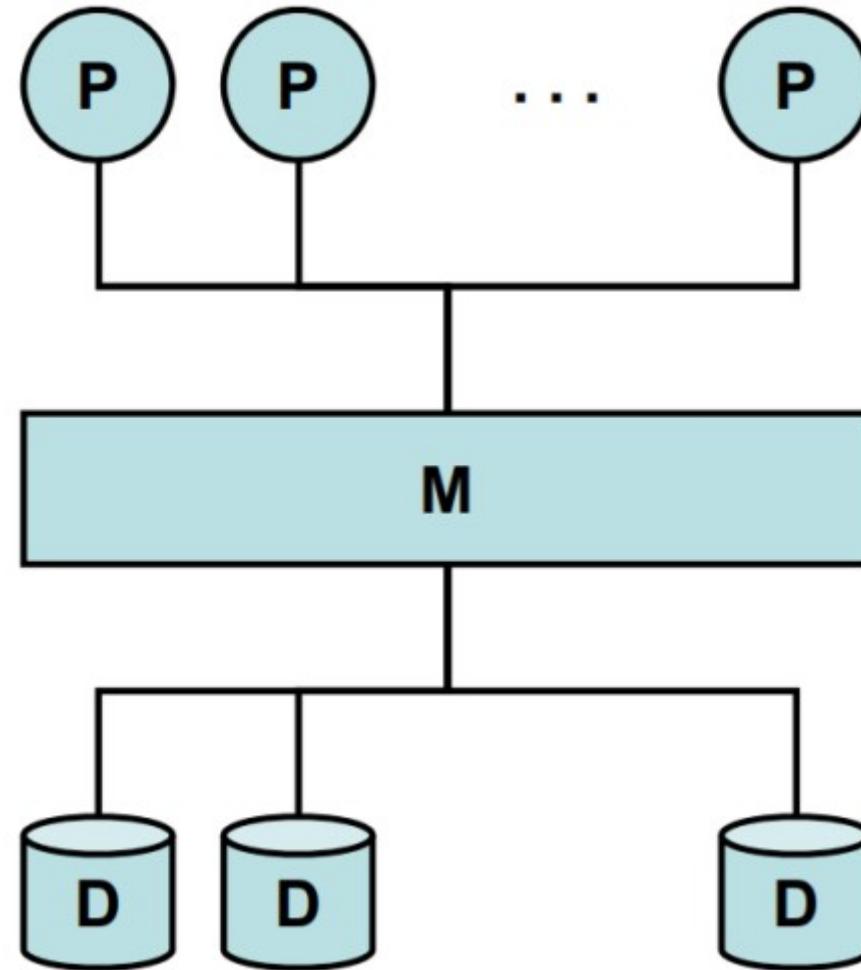
Архитектура распределённых систем

Одна из наиболее популярных на практике классификаций распределённых систем выделяет три типа архитектуры:

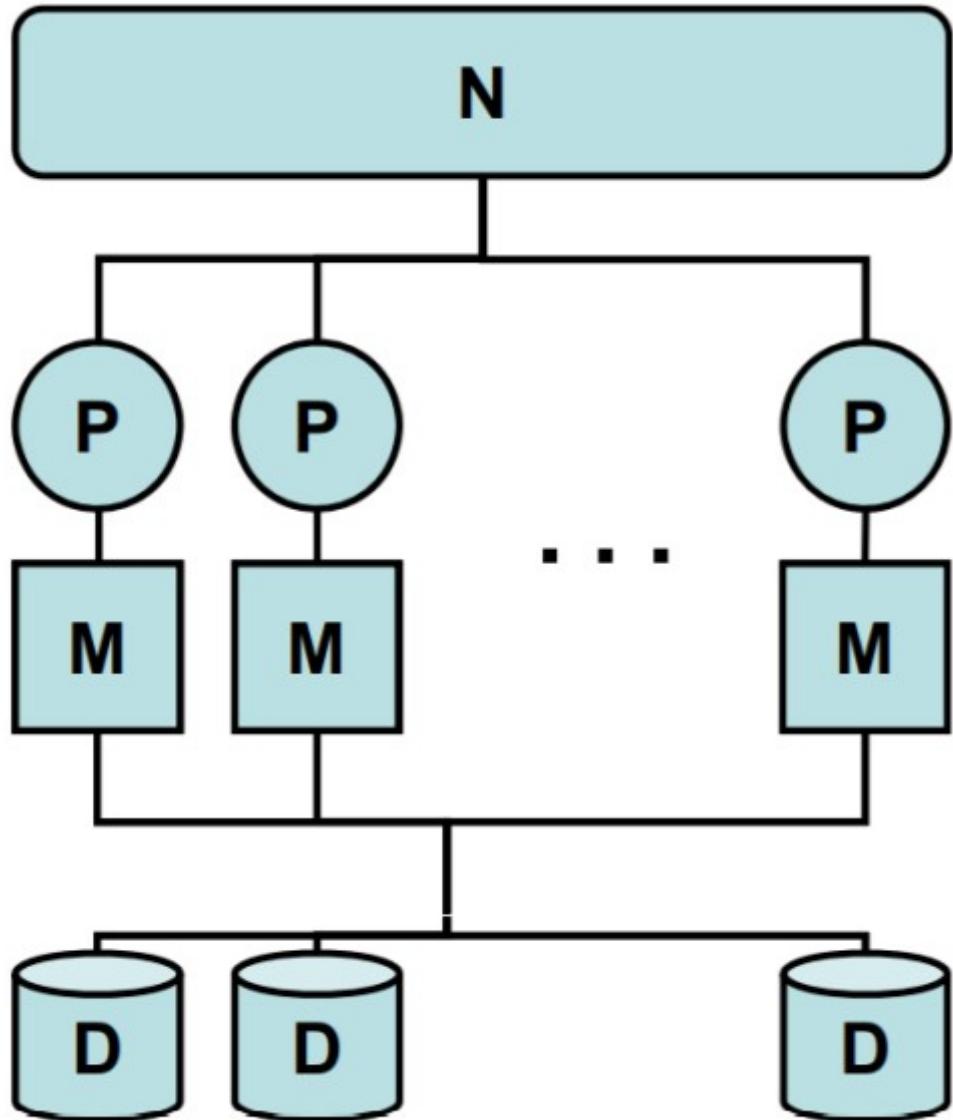
- SE ([Shared Everything](#) aka Shared Memory) — архитектура с совмешёнными памятью и дисками;
- SD ([Shared Disks](#)) — архитектура с совмешёнными дисками, но независимыми пулами памяти;
- SN ([Shared Nothing](#)) — архитектура без совместного использования ресурсов.

SE (Shared-Everything)

- Процессоры работают с единым адресным пространством общей оперативной памяти.
- Дисковое пространство едино и доступно всем процессорам с одинаковым временем доступа.
- Межпроцессорные коммуникации осуществляются через общую оперативную память.
- Каждый процессор в SE-системе имеет собственную кэш-память



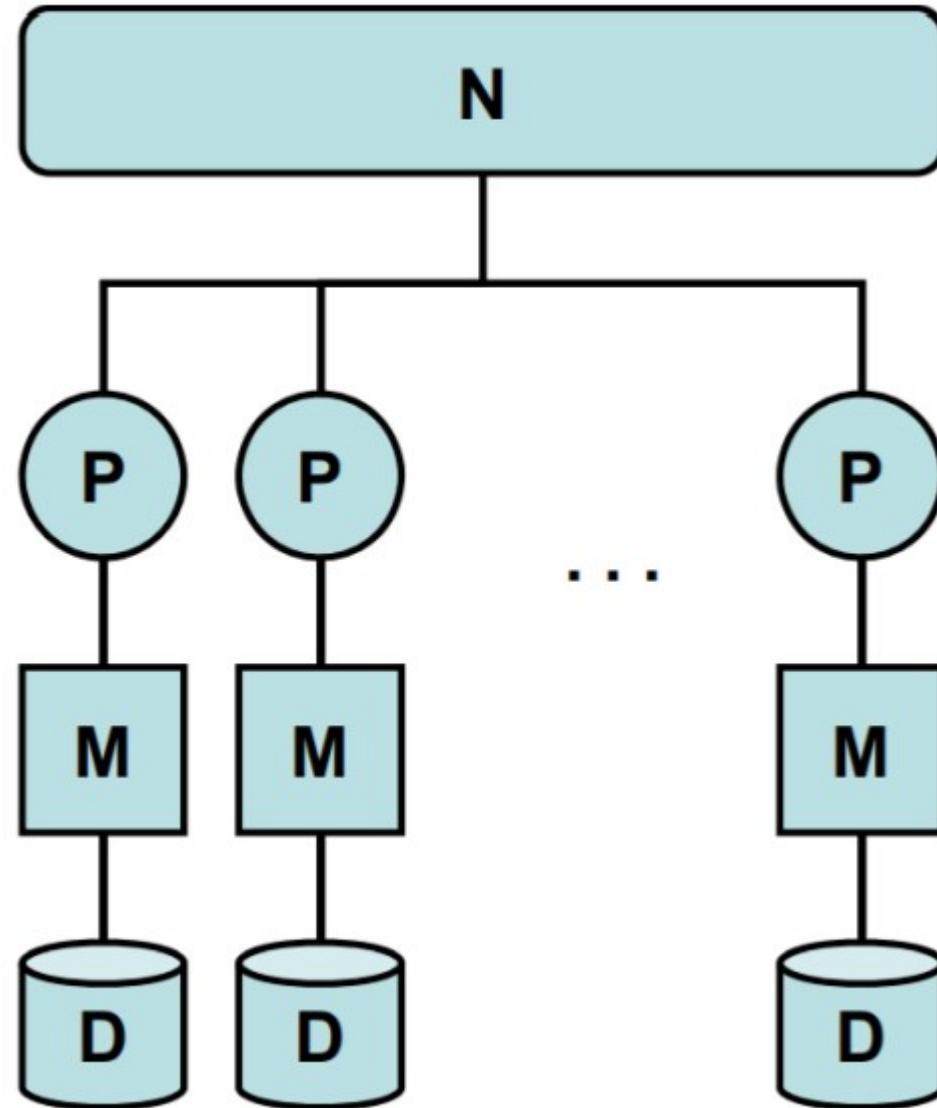
SD (Shared-Disks)



- Каждый процессор работает с приватной оперативной памятью.
- Дисковое пространство едино и доступно всем процессорам с одинаковым временем доступа.
- Межпроцессорные коммуникации осуществляются через высокоскоростную соединительную сеть.

SN (Shared-Nothing)

- Каждый процессор работает с приватными оперативной памятью и дисковым пространством.
- Межпроцессорные коммуникации осуществляются через высокоскоростную соединительную сеть



Фрагментация данных

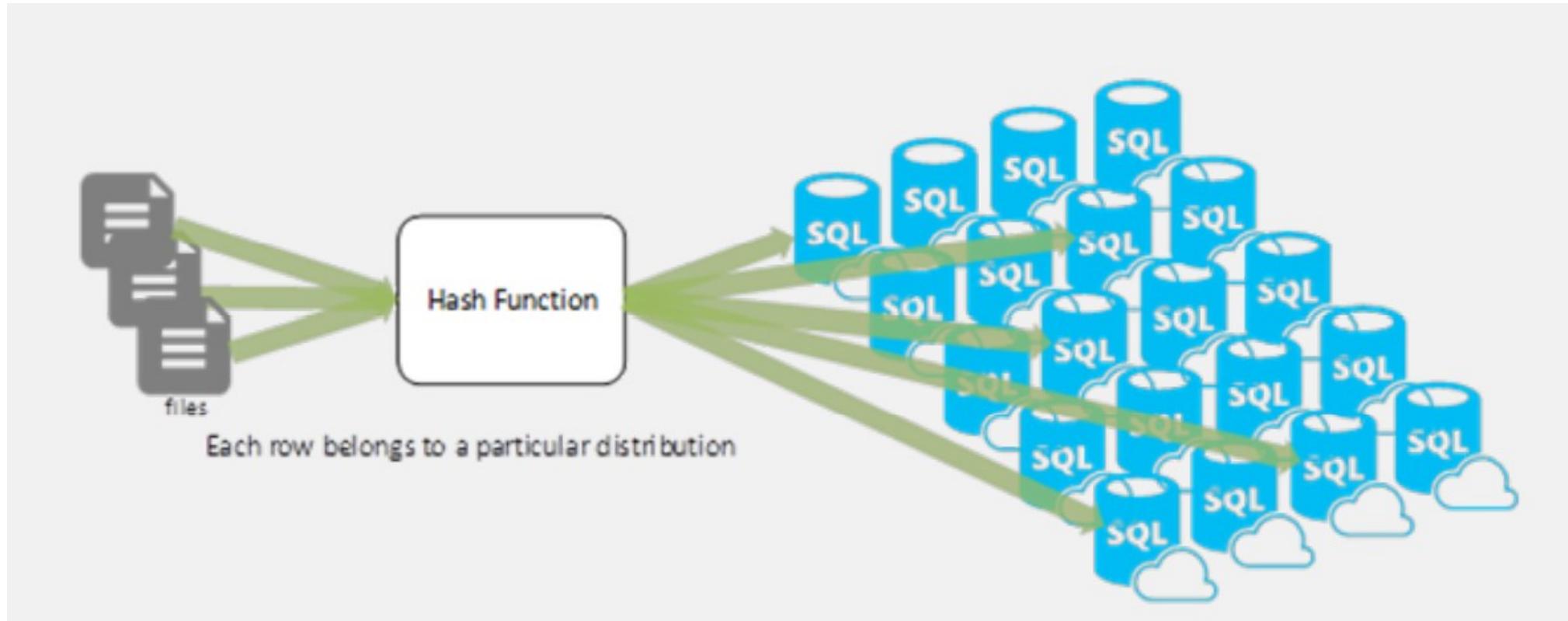
Data Distribution требуется для фрагментации данных между дисками в Shared Nothing архитектуре.

Основные способы фрагментирования данных:

- С помощью функции распределения (обычно, хеш-функция);
- Round-Robin.

Функция фрагментации

Для каждой строки функция фрагментации определяет диск, на который будет записана эта строка.

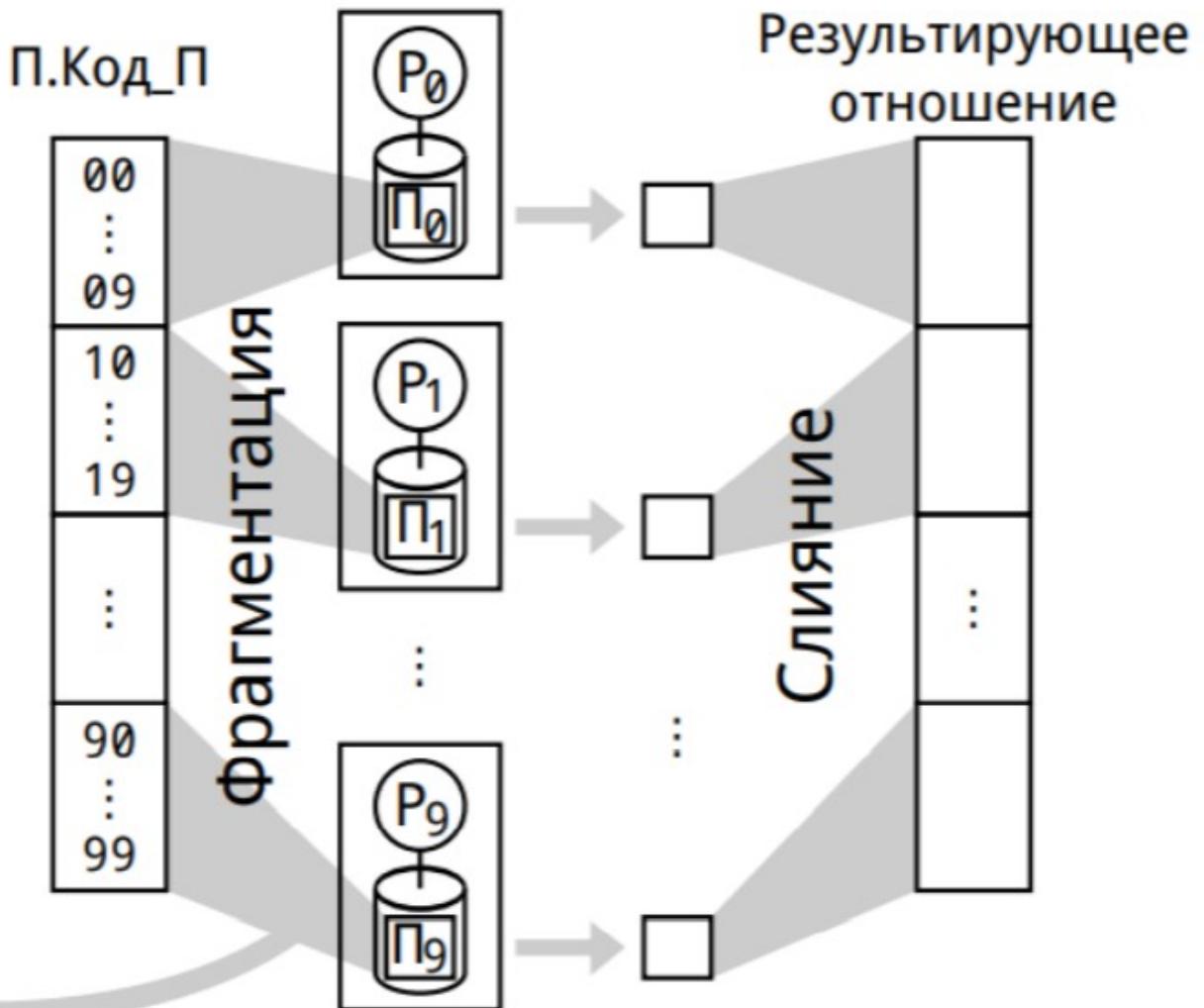


Фрагментация данных

$$\Pi_i = \{t | t \in \Pi, \phi(t) = i\}$$
$$i = 0, \dots, 9$$

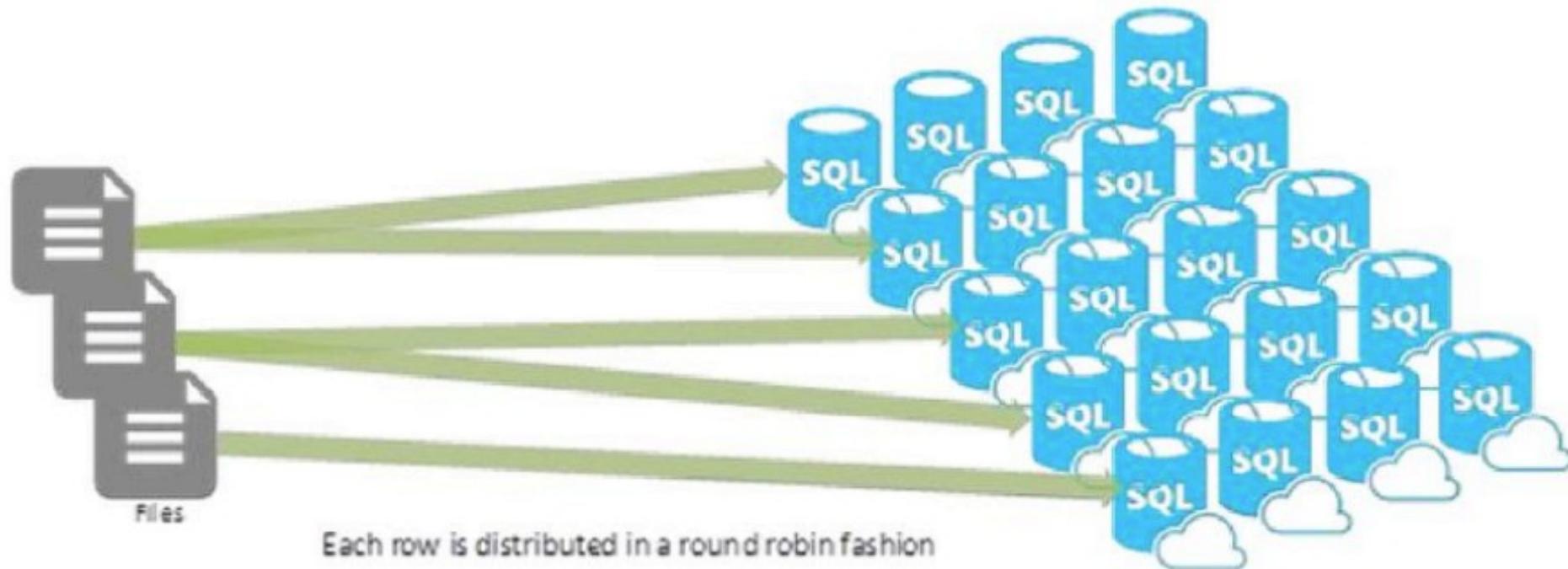
Функция фрагментации

$$\phi(t) = (t.\text{Код_П} \text{ div } 10) \text{ mod } 10$$



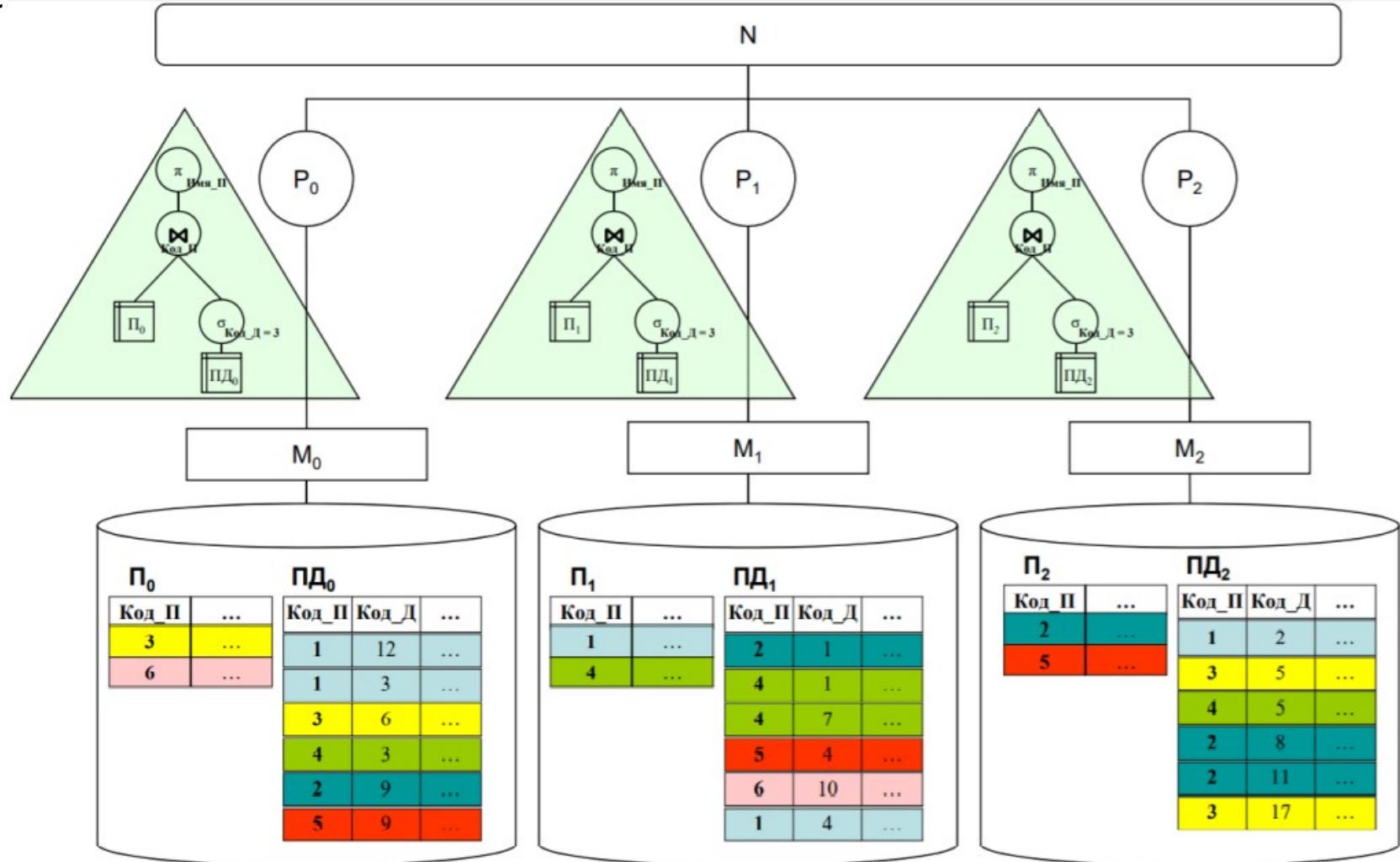
Фрагментация данных

Round-robin алгоритм распределяет строки по закольцованной очереди



Тиражирование запросов

Распределённая СУБД получает от пользователя один логический запрос, но физически каждый узел может выполнять свою часть независимо



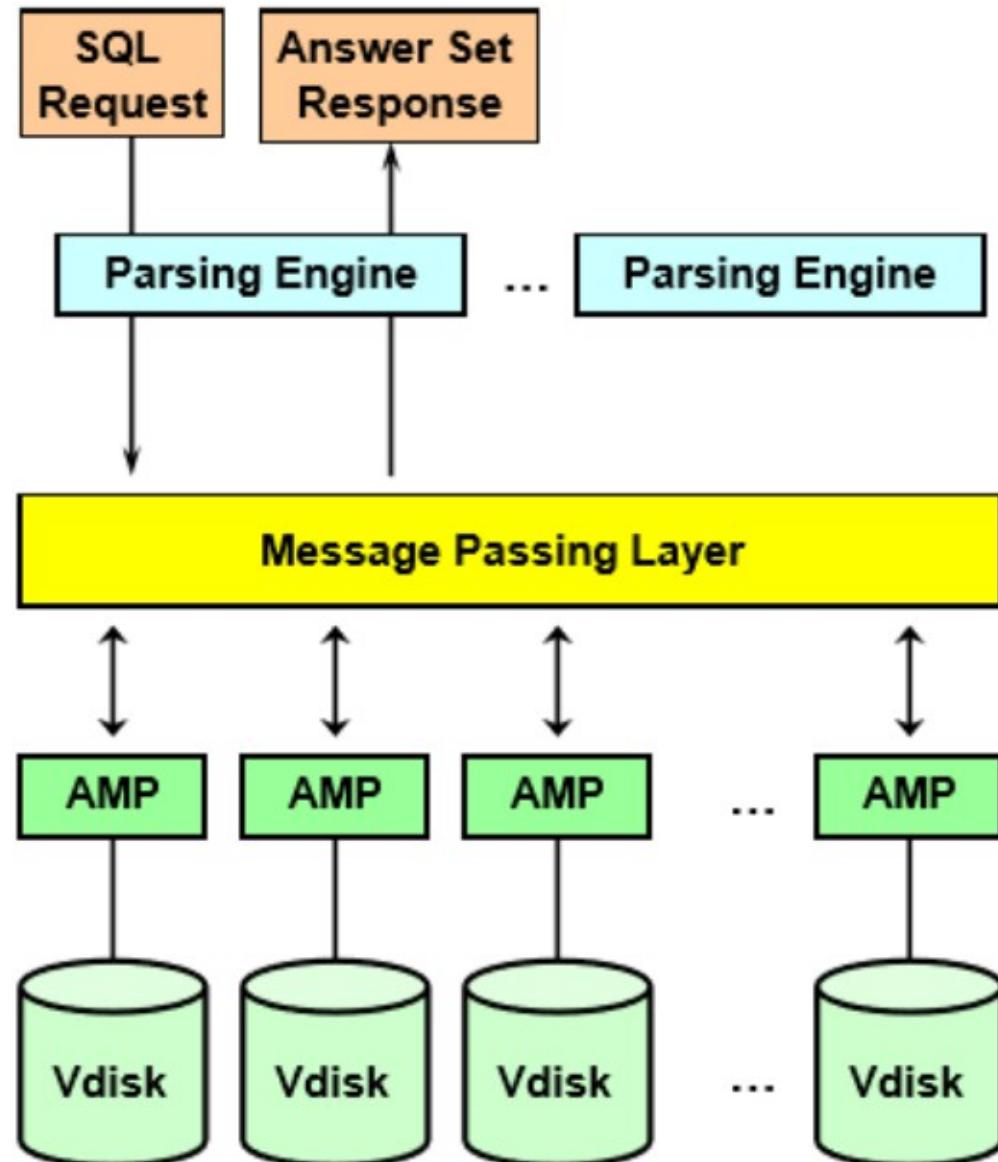


Teradata

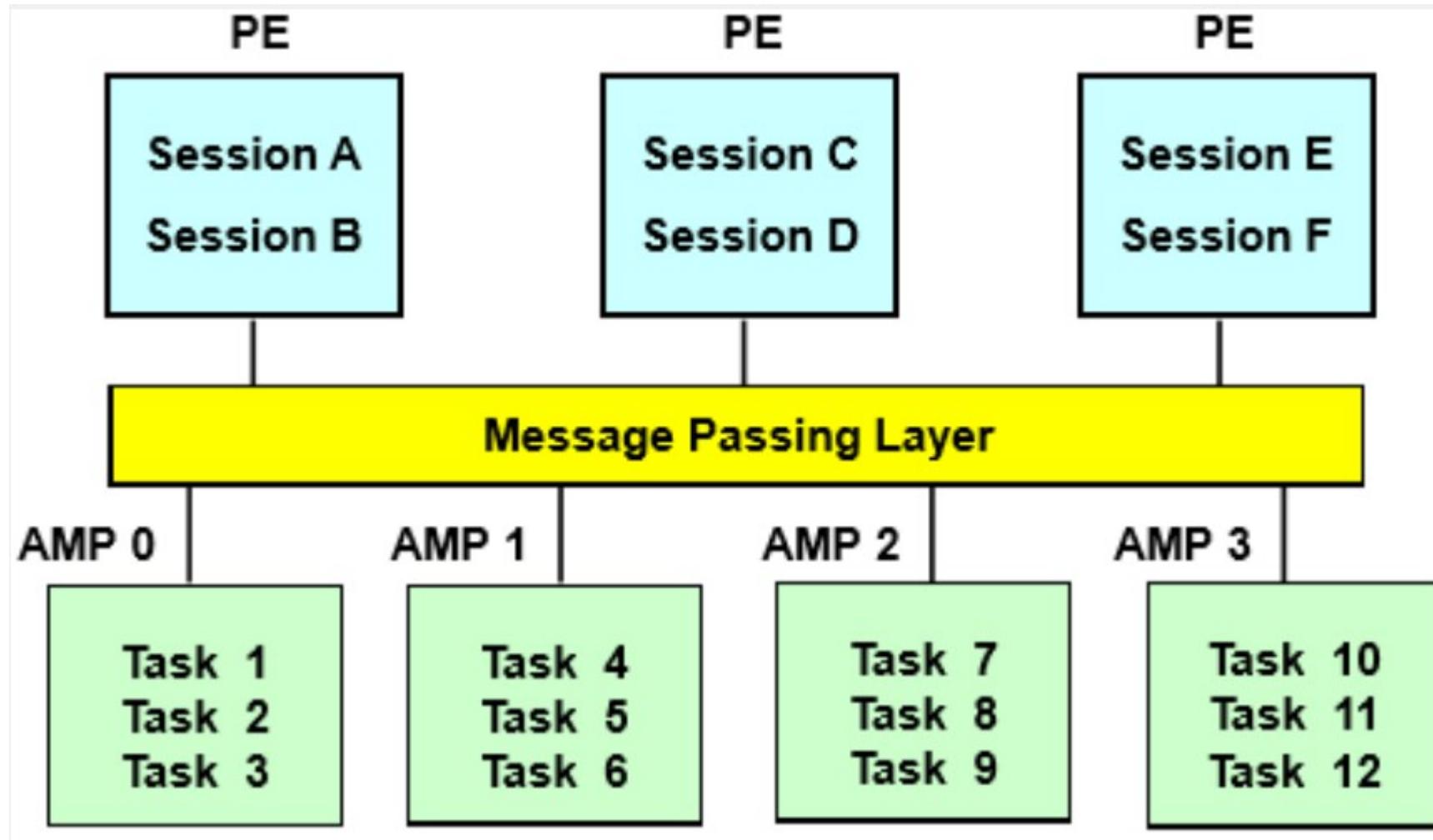


Логическая архитектура Teradata

- **Parsing Engines** Управление сессиями пользователей. Чтение и оптимизация запросов. Выдача результатов запросов пользователю.
- **Message Passing Layer** Посредник между AMP и PE. Слой коммуникации между узлами системы.
- **Access Module Processors** Управляет выделенным пространством памяти и дисков. Выполняет запрошенные от MPL операции.
- **Virtual Disks** Непосредственно выделенное дисковое пространство



Параллелизм в Teradata



Доступ к данным в Teradata

Существует три основных способа доступа к данным:

- Использование первичного индекса (один AMP)
- Использование вторичного индекса (два или все AMP)
- Полное сканирование таблицы (все AMP)

Первичный индекс

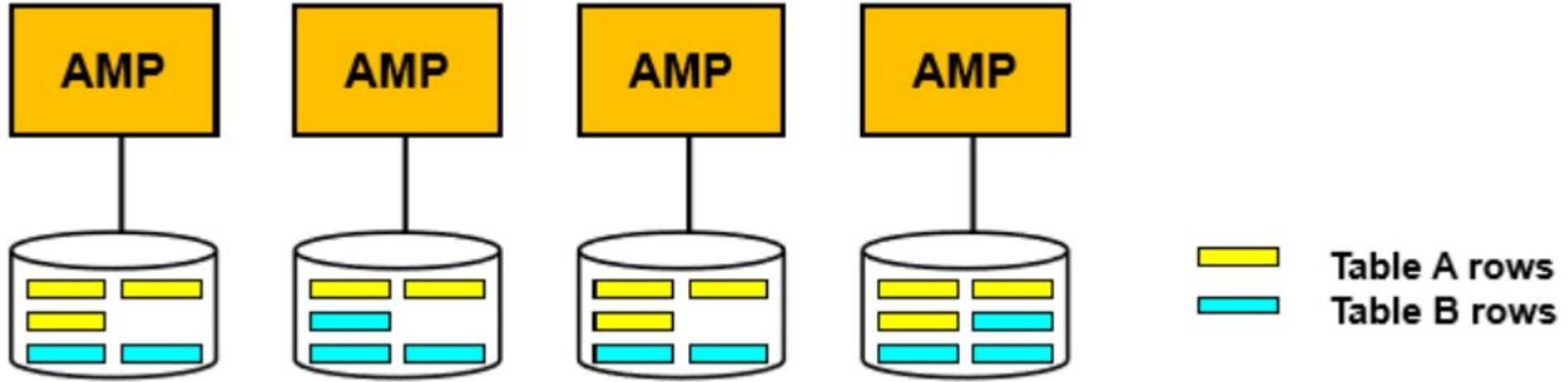
Первичный индекс (Primary index) не надо путать с первичным ключом:

- Первичный ключ – это понятие из теории реляционных баз данных.
- Первичный индекс в Teradata задаёт способ распределения и обработки данных .
- У таблиц атрибуты первичного ключа и первичного индекса могут совпадать.
- В хорошо спроектированной OLAP-системе первичный ключ и первичный индекс в большинстве случаев не будут совпадать

Сравнение ключа и индекса

| Первичный ключ | Первичный индекс |
|--|--|
| Логическая концепция на этапе моделирования | Физическая модель хранения данных |
| Teradata не нуждается в определении PK | Таблица должна обязательно иметь один PI |
| Без ограничений на количество атрибутов | Не более 64 атрибутов |
| Зафиксирован на этапе моделирования | Зафиксирован на этапе создания |
| Должен быть уникальным | Может быть неуникальным |
| Идентифицирует единственную строку | Может идентифицировать несколько строк |
| Не может быть Null | Может быть Null |
| Не влияет на скорость доступа к данным | Напрямую влияет на скорость доступа к данным |
| Выбирается для логической непротиворечивости | Выбирается для улучшения производительности |

Распределение данных по узлам



Строки каждой таблицы распределены по узлам

- Для таблиц с первичным индексом распределение по узлам задаётся хеш-функцией от колонок первого ключа.
- Для таблиц без первичного индекса распределение строк происходит по случайному алгоритму и всегда равномерно.

DDL первичного индекса

```
1. CREATE TABLE sample_1  
2. (col_a INTEGER, col_b CHAR(10), col_c DATE)  
3. UNIQUE PRIMARY INDEX (col_b);
```

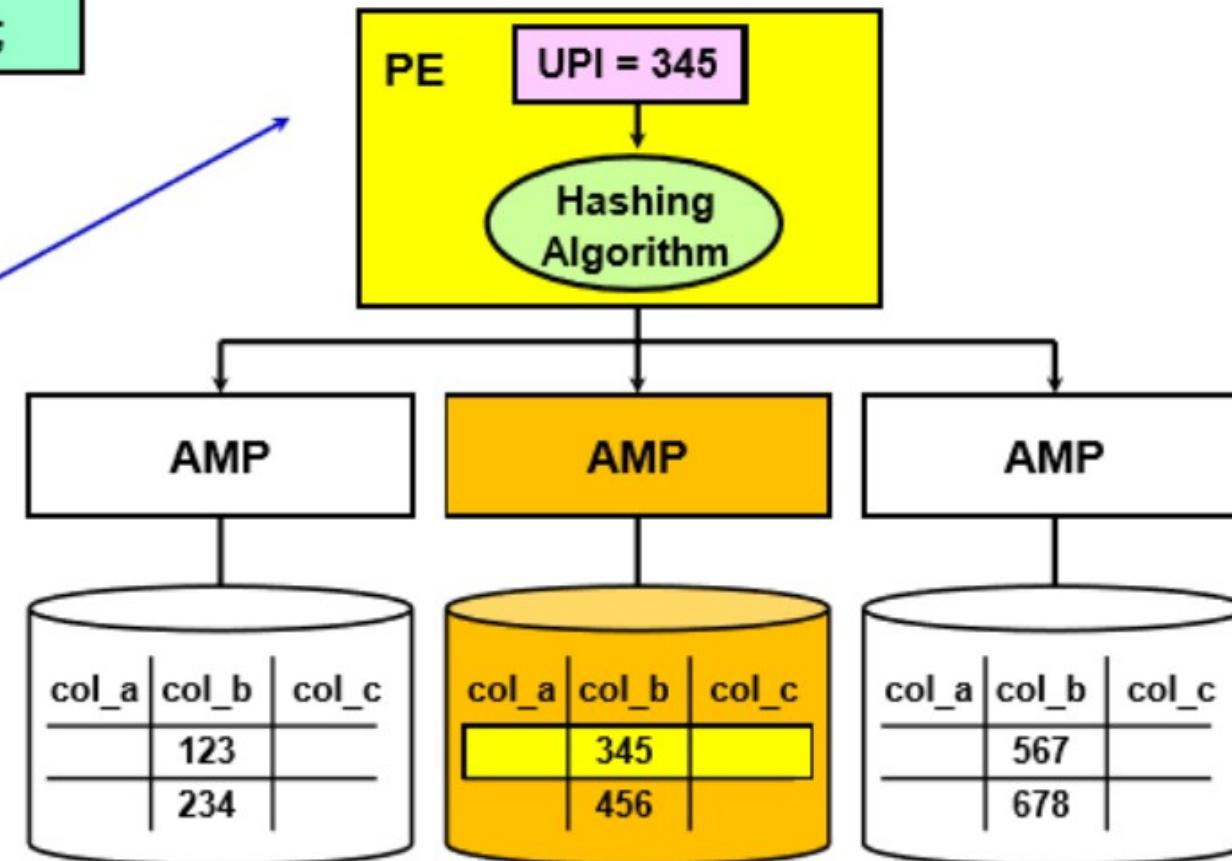
```
1. CREATE TABLE sample_2  
2. (col_m INTEGER, col_n CHAR(10), col_o DATE)  
3. PRIMARY INDEX (col_m);
```

```
1. CREATE TABLE sample_3  
2. (col_x INTEGER, col_y CHAR(10), col_z DATE)  
3. NO PRIMARY INDEX;
```

Доступ к данным с помощью IIDI

```
CREATE TABLE sample_1
  (col_a  INTEGER
   ,col_b  INTEGER
   ,col_c  CHAR(4))
UNIQUE PRIMARY INDEX (col_b);
```

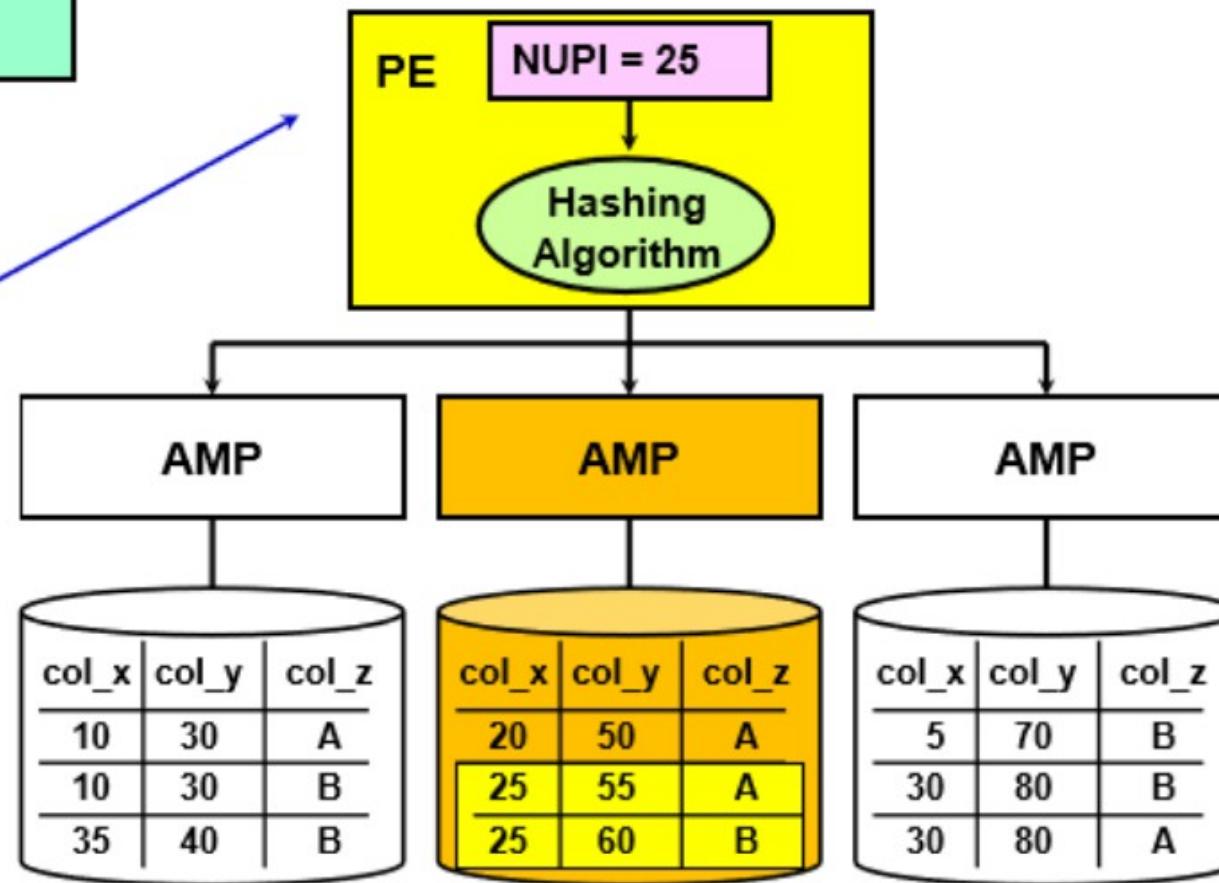
```
SELECT col_a
      ,col_b
      ,col_c
  FROM sample_1
 WHERE col_b = 345;
```



Доступ к данным с помощью NUPI

```
CREATE TABLE sample_2
  (col_x  INTEGER
   ,col_y  INTEGER
   ,col_z  CHAR(4))
PRIMARY INDEX (col_x);
```

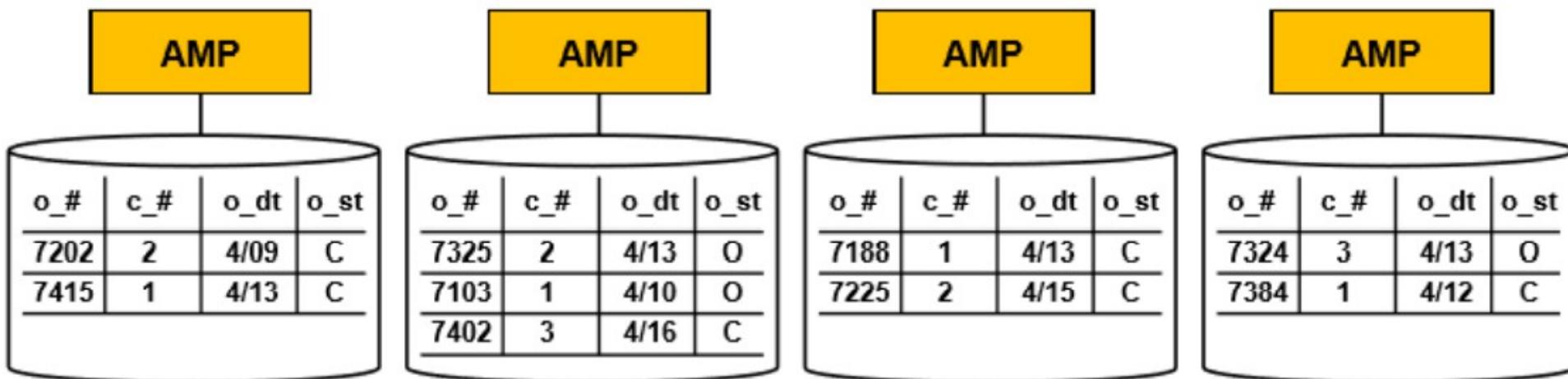
```
SELECT col_x
      ,col_y
      ,col_z
  FROM sample_2
 WHERE col_x = 25;
```



Выбор первичного

Orders

| Order Number | Customer Number | Order Date | Order Status |
|--------------|-----------------|------------|--------------|
| PK | | | |
| UPI | | | |
| 7325 | 2 | 4/13 | O |
| 7324 | 3 | 4/13 | O |
| 7415 | 1 | 4/13 | C |
| 7103 | 1 | 4/10 | O |
| 7225 | 2 | 4/15 | C |
| 7384 | 1 | 4/12 | C |
| 7402 | 3 | 4/16 | C |
| 7188 | 1 | 4/13 | C |
| 7202 | 2 | 4/09 | C |



Выбор первичного

Orders

| Order Number | Customer Number | Order Date | Order Status |
|--------------|-----------------|------------|--------------|
| PK | | | |
| | NUP1 | | |
| 7325 | 2 | 4/13 | O |
| 7324 | 3 | 4/13 | O |
| 7415 | 1 | 4/13 | C |
| 7103 | 1 | 4/10 | O |
| 7225 | 2 | 4/15 | C |
| 7384 | 1 | 4/12 | C |
| 7402 | 3 | 4/16 | C |
| 7188 | 1 | 4/13 | C |
| 7202 | 2 | 4/09 | C |

AMP

| o_# | c_# | o_dt | o_st |
|------|-----|------|------|
| 7325 | 2 | 4/13 | O |
| 7202 | 2 | 4/09 | C |
| 7225 | 2 | 4/15 | C |

AMP

| o_# | c_# | o_dt | o_st |
|------|-----|------|------|
| 7384 | 1 | 4/12 | C |
| 7103 | 1 | 4/10 | O |
| 7415 | 1 | 4/13 | C |
| 7188 | 1 | 4/13 | C |

AMP

| o_# | c_# | o_dt | o_st |
|------|-----|------|------|
| 7402 | 3 | 4/16 | C |
| 7324 | 3 | 4/13 | O |

AMP

Выбор первичного

Orders

| Order Number | Customer Number | Order Date | Order Status |
|--------------|-----------------|------------|--------------|
| PK | | | |
| | | | NUPI |
| 7325 | 2 | 4/13 | O |
| 7324 | 3 | 4/13 | O |
| 7415 | 1 | 4/13 | C |
| 7103 | 1 | 4/10 | O |
| 7225 | 2 | 4/15 | C |
| 7384 | 1 | 4/12 | C |
| 7402 | 3 | 4/16 | C |
| 7188 | 1 | 4/13 | C |
| 7202 | 2 | 4/09 | C |

AMP

AMP

AMP

AMP

| o_# | c_# | o_dt | o_st |
|------|-----|------|------|
| 7402 | 3 | 4/16 | C |
| 7202 | 2 | 4/09 | C |
| 7225 | 2 | 4/15 | C |
| 7415 | 1 | 4/13 | C |
| 7188 | 1 | 4/13 | C |
| 7384 | 1 | 4/12 | C |

| o_# | c_# | o_dt | o_st |
|------|-----|------|------|
| 7103 | 1 | 4/10 | O |
| 7324 | 3 | 4/13 | O |
| 7325 | 2 | 4/13 | O |

Выбор первичного индекса

Критерии выбора первичного индекса:

- Доступ к данным столбцов первичного индекса должен осуществляться чаще остальных
- Распределение значений должно быть равномерным и без выбросов (skew factor)
- Для больших таблиц количество уникальных значений должно превышать количество узлов как минимум в 10 раз
- Изменения данных не должно происходить слишком часто

Вторичный индекс

Вторичный индекс является альтернативным путем доступа к данным

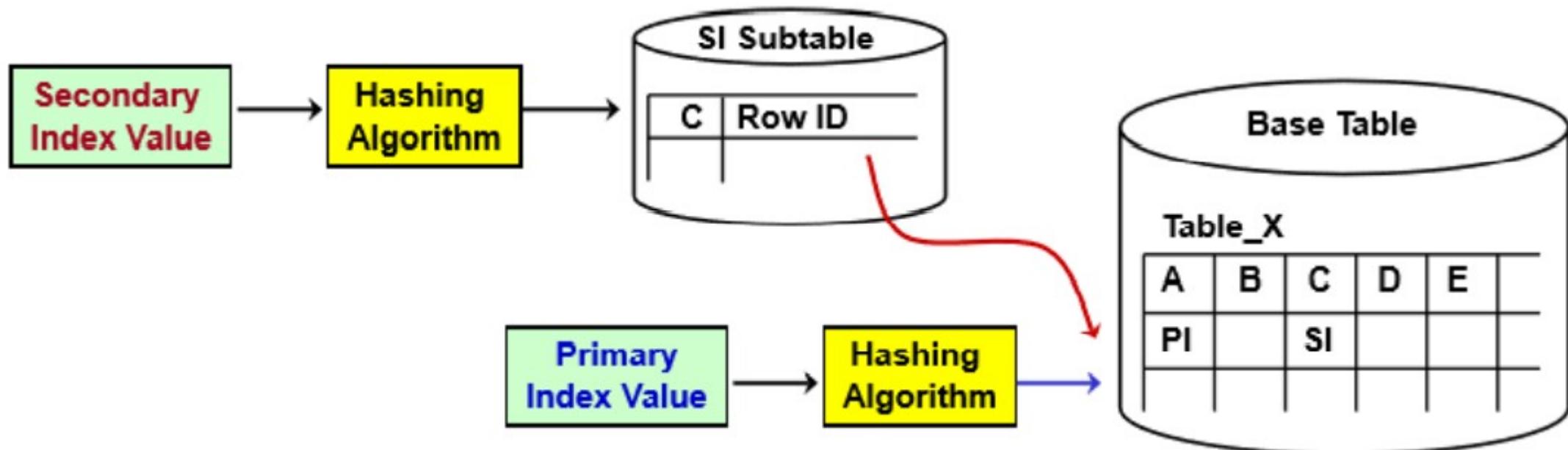
- Не влияет на дистрибуцию данных по узлам
- Увеличивает объём хранимых на диске данных и повышает расходы на поддержку системы
- Могут добавляться и удаляться в любое время
- Используются для повышения производительности

```
1. CREATE UNIQUE INDEX (Employee_Number)  
2. ON Employee;
```

```
1. CREATE INDEX (Last_Name)  
2. ON Employee;
```

Вторичный индекс

Вторичный индекс хранится в виде дополнительной таблицы, распределённой по ключевым атрибутам индекса.



Доступ к данным с помощью USI

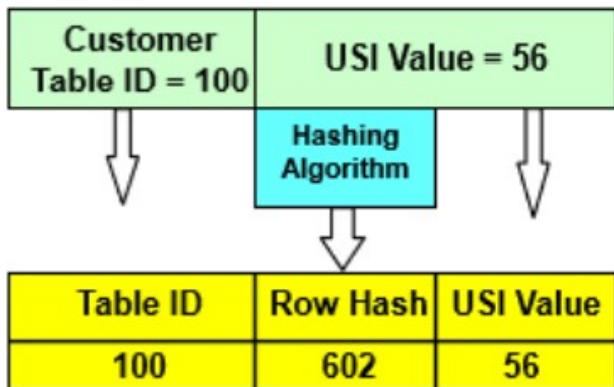
Create USI

```
CREATE UNIQUE INDEX  
(Cust) ON Customer;
```

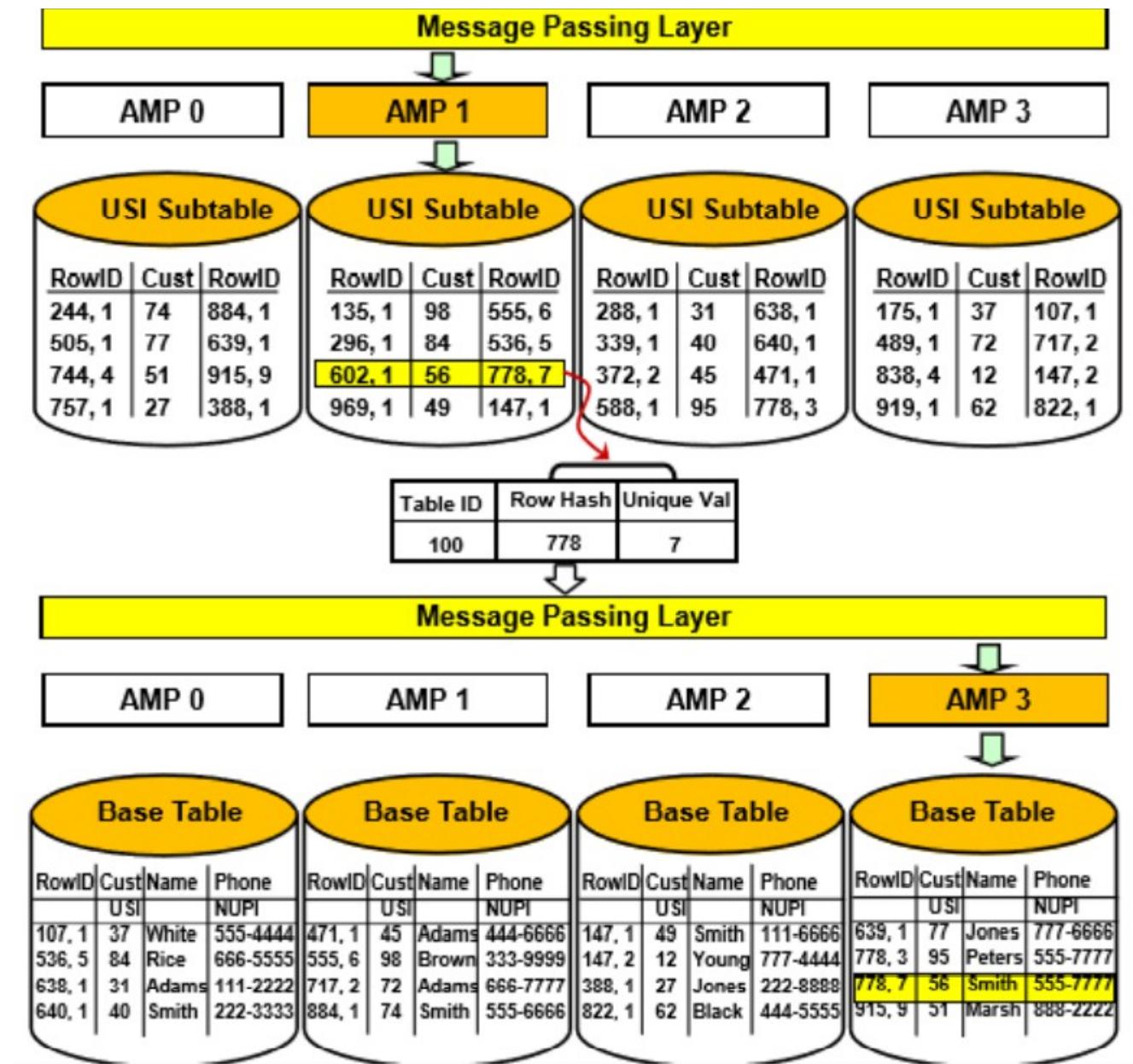
Access via USI

```
SELECT *  
FROM Customer  
WHERE Cust = 56;
```

PE



to MPL



Доступ к данным с помощью NUSI

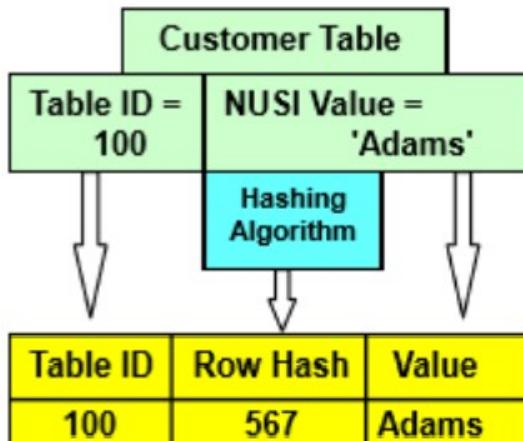
Create NUSI

```
CREATE INDEX (Name)  
ON Customer;
```

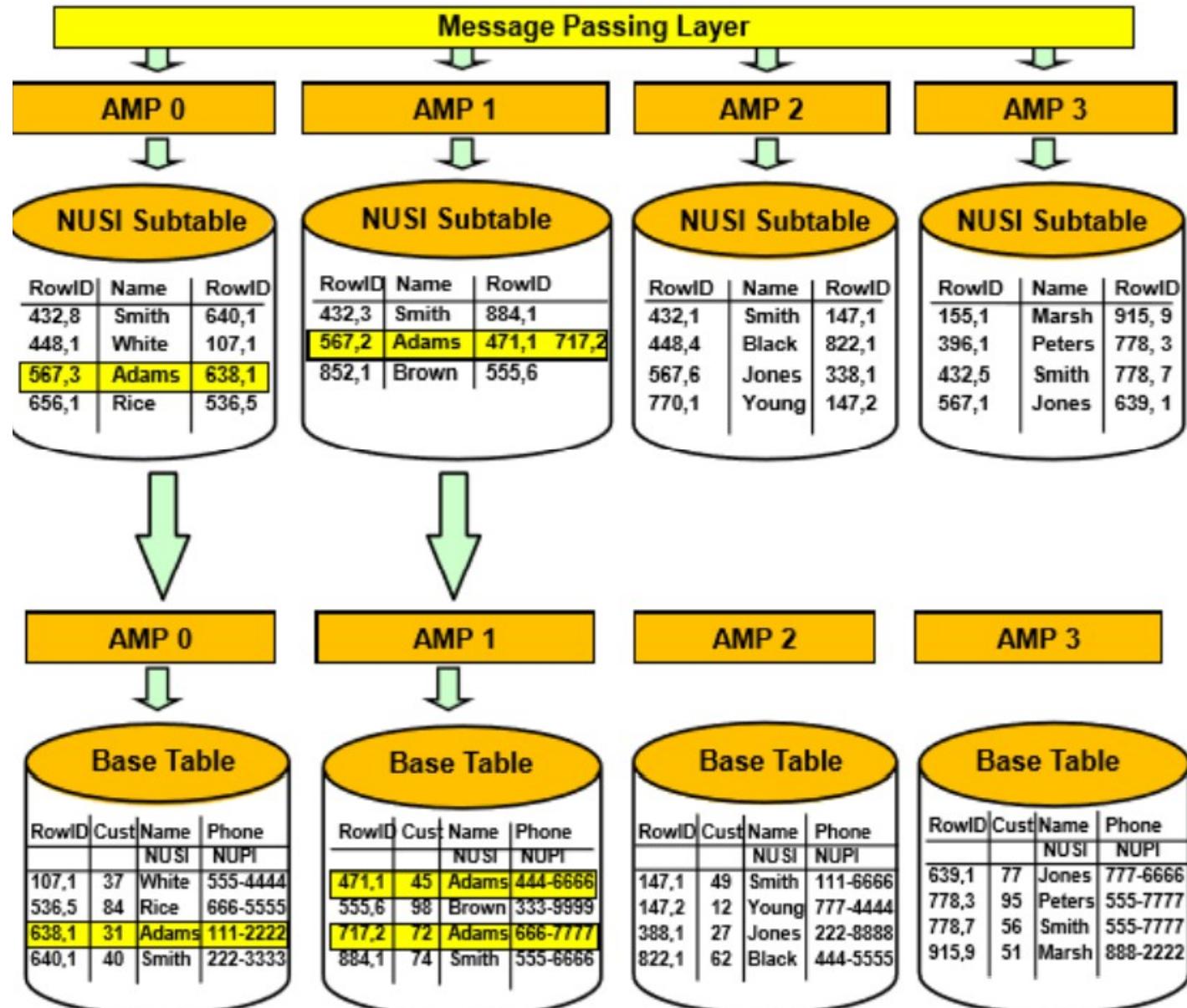
Access via NUSI

```
SELECT *  
FROM Customer  
WHERE Name = 'Adams';
```

PE



to MPL



Использование вторичного индекса

- Использование вторичного индекса увеличивает накладные расходы
- Основной критерий использования вторичного индекса – частое использование полей в запросе
- Если условие высокоселективное, то выгоднее использовать полное сканирование (пороговое значение - 10%)
- Лучше избегать использование вторичного индекса по часто обновляемым данным
- Неиспользуемые индексы необходимо удалять

Полное сканирование

Полное сканирование таблицы означает чтение каждой строки

Происходит в том случае, если:

- Поля индекса не используются в запросе
- Поля индекса используются в сложных условиях (не равенство)

```
1. SELECT * FROM Customer  
2. WHERE Cust_Phone LIKE '858-485-_____';  
3.  
4. SELECT * FROM Customer  
5. WHERE Cust_Name = 'Koehler';  
6.  
7. SELECT * FROM Customer  
8. WHERE Cust_ID > 1000
```

Партицирование

Партицирование - физическое разделение данных по какому-то полю

- Строки распределяются по AMP в зависимости от первичного индекса
- На каждом AMP строки делятся по партициям

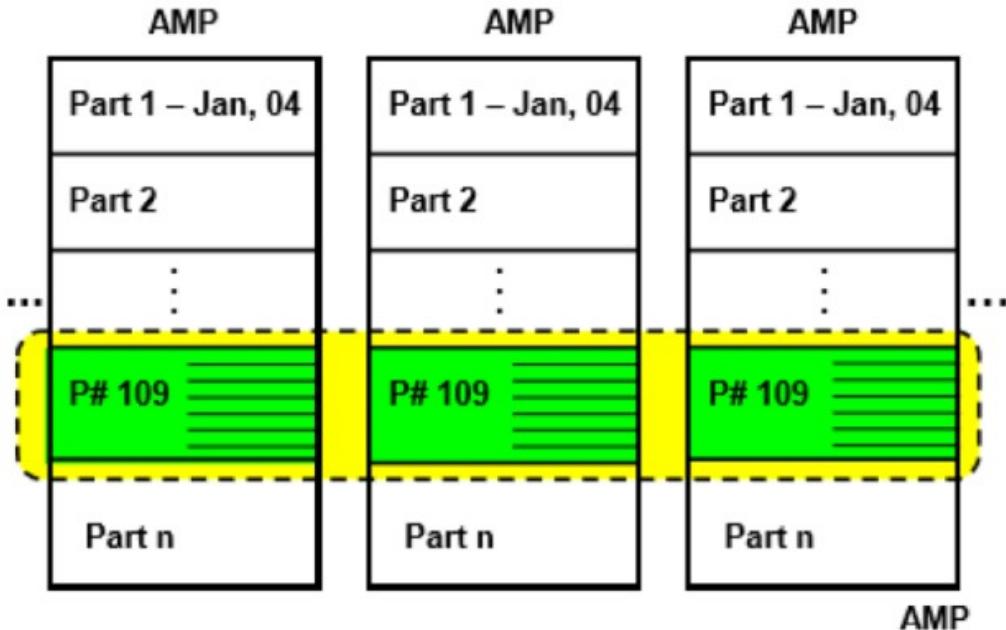
```
1. CREATE TABLE ...
2. [UNIQUE] PRIMARY INDEX (col1, col2, ...)
3. PARTITION BY <partitioning-expression>
```

<partitioning-expression>:

- Range
- Case

Столбцыパーティционирования могут не совпадать со столбцами первичного индекса

Использованиеパーティционирования



QUERY – PPI

```
SELECT *
FROM Claim_PPI
WHERE claim_date BETWEEN
DATE '2013-01-01' AND
DATE '2013-01-31';
```

PLAN – PPI

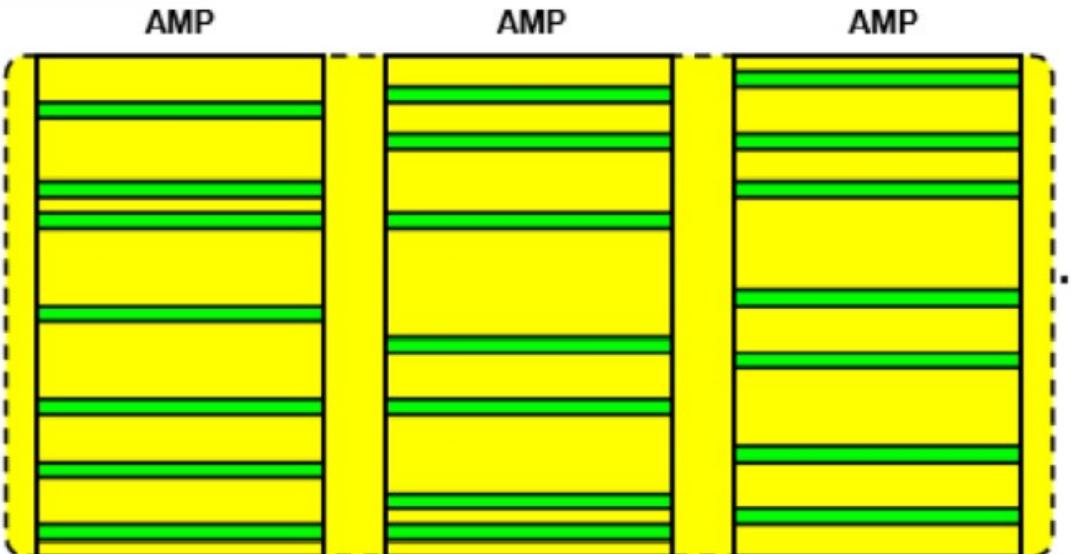
ALL-AMPs – Single Partition Scan
EXPLAIN estimated cost – 0.44 sec.

QUERY – NPPI

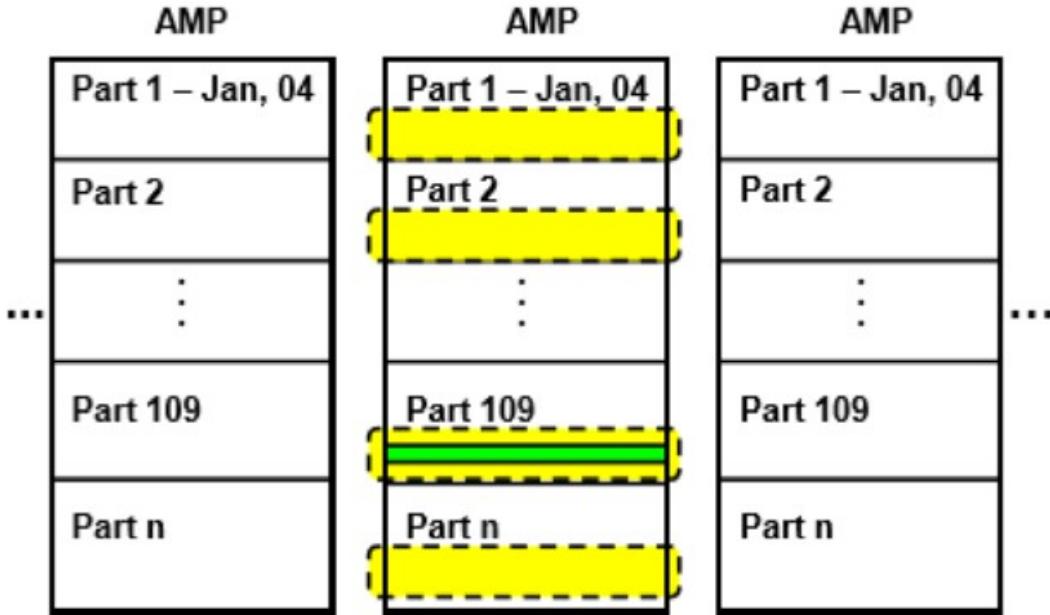
```
SELECT *
FROM Claim_NPPI
WHERE claim_date BETWEEN
DATE '2013-01-01' AND
DATE '2013-01-31';
```

PLAN – NPPI

ALL-AMPs – Full Table Scan
EXPLAIN estimated cost – 49.10 sec.



Использованиеパーティционирования



QUERY – PPI

```
SELECT *
FROM   Claim_PPI
WHERE  claim_id = 260221;
```

PLAN – PPI

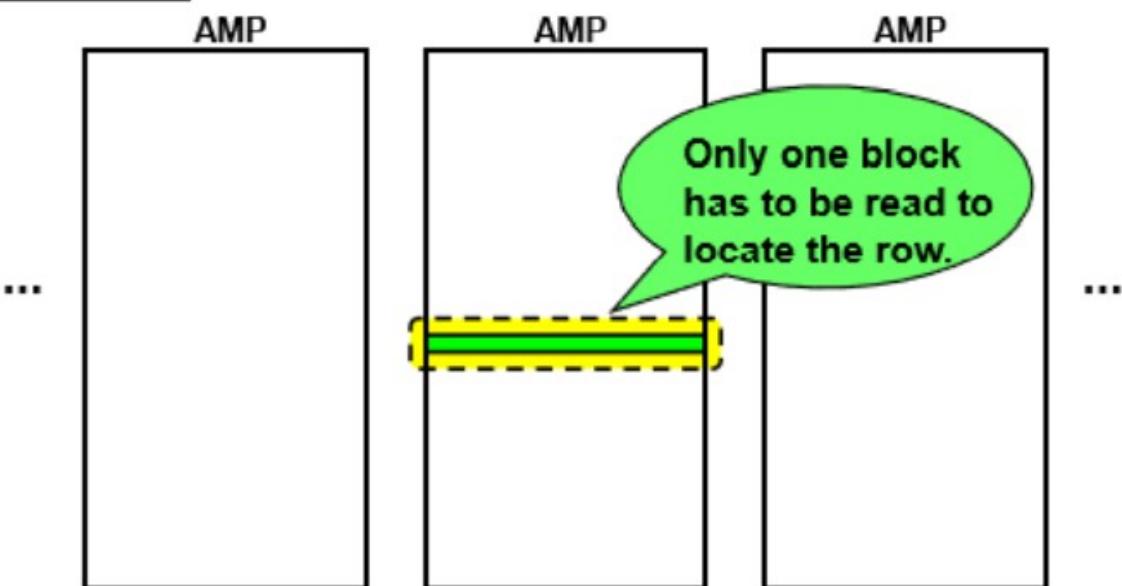
One AMP – All Partitions are probed
EXPLAIN estimated cost – 0.09 sec.

QUERY – NPPI

```
SELECT *
FROM   Claim_NPPI
WHERE  claim_id = 260221;
```

PLAN – NPPI

One AMP – UPI Access
EXPLAIN estimated cost – 0.00 sec.



Использованиеパーティционирования

Преимуществаパーティционирования:

- При использовании в запросе таблицы, по которой произведеноパーティционирование, читается только необходимый блок информации

Недостаткиパーティционирования:

- накладные расходы на хранение и вставку записей;
- запросы без использования колонокパーティционирования работают хуже;
- не работает Merge Join без использования колонокパーティционирования.

Многоуровневое партицирование



Виды join'ов

Основные стратегии join'ов:

- Merge Join
- Product Join
- Hash Join

Основной критерий для выбора стратегии –
первичный индекс соединяемых таблиц:

- Обе таблицы соединяются по полям, составляющим
первичный индекс
- Одна из таблиц соединяется по полям,
составляющим первичный индекс
- Ни одна из таблиц не использует первичный индекс
в соединении

Перераспределение

Join columns are from the same domain.

No Redistribution needed.

```
SELECT ...  
FROM Table1 T1  
INNER JOIN Table2 T2  
ON T1.A = T2.A;
```

| T1 | A | B | C |
|-----|-----|-----|---|
| PI | | | |
| 100 | 214 | 433 | |

| T2 | A | B | C |
|-----|-----|-----|---|
| PI | | | |
| 100 | 725 | 002 | |

Join columns are from the same domain.

Redistribution needed.

```
SELECT ...  
FROM Table3 T3  
INNER JOIN Table4 T4  
ON T3.A = T4.B;
```

| T3 | A | B | C |
|-----|-----|-----|---|
| PI | | | |
| 255 | 345 | 225 | |

| T4 | A | B | C |
|-----|-----|-----|---|
| PI | | | |
| 867 | 255 | 566 | |

Redistribute T4 rows in spool on column B.

| SPOOL | A | B | C |
|-------|-----|-----|---|
| | | PI | |
| 867 | 255 | 566 | |

Merge Join

- Необходимо, чтобы строки были одинаково распределены
- Блоки таблицы читаются единожды
- В общем случае быстрее иных видов join

Алгоритм:

1. Находим меньшую таблицу
2. Выбираем, что с ней делать:
 - Перераспределяем
 - Дублируем
 - Сортируем
3. Производим join

Merge Join

Employee

| Enum | Name | Dept |
|------|--------|------|
| PK | | FK |
| UPI | | |
| 1 | BROWN | 200 |
| 2 | SMITH | 310 |
| 3 | JONES | 310 |
| 4 | CLAY | 400 |
| 5 | PETERS | 150 |
| 6 | FOSTER | 200 |
| 7 | GRAY | 310 |
| 8 | BAKER | 310 |
| 9 | TYLER | 450 |
| 10 | CARR | 450 |

Employee_Phone

| Enum | Area_Code | Phone |
|------|-----------|---------|
| PK | | |
| FK | | |
| NUPI | | |
| 1 | 213 | 3241576 |
| 1 | 213 | 4950703 |
| 3 | 408 | 3628822 |
| 4 | 415 | 6347180 |
| 5 | 312 | 7463513 |
| 6 | 203 | 8337461 |
| 8 | 301 | 2641616 |
| 8 | 301 | 6675885 |
| 8 | 301 | 2641616 |

Primary Indexes match: no duplication or sorting needed

Example: SELECT E.Enum, E.Name, ...
 FROM Employee E
 INNER JOIN Employee_Phone P
 ON E.Enum = P.Enum ;

Employee rows hash distributed on Enum (UPI)

6 FOSTER 200
8 BAKER 310

4 CLAY 400
3 JONES 310
9 TYLER 450

1 BROWN 200
7 GRAY 310

5 PETERS 150
2 SMITH 310
10 CARR 450

Employee_Phone rows hash distributed on Enum (NUPI)

6 203 8337461
8 301 2641616
8 301 6675885

4 415 6347180
3 408 3628822

1 213 3241576
1 213 4950703

5 312 7463513

Merge Join

Employee

| Enum | Name | Dept |
|------|--------|------|
| PK | | FK |
| UPI | | |
| 1 | BROWN | 200 |
| 2 | SMITH | 310 |
| 3 | JONES | 310 |
| 4 | CLAY | 400 |
| 5 | PETERS | 150 |
| 6 | FOSTER | 200 |
| 7 | GRAY | 310 |
| 8 | BAKER | 310 |
| 9 | TYLER | 450 |
| 10 | CARR | 450 |

Example:

```
SELECT E.Enum, E.Name, D.Dept, D.Name  
FROM Employee E  
INNER JOIN Department D  
ON E.Dept = D.Dept ;
```

Employee rows hash distributed on Employee.Enum (UPI)

| | | | |
|--------------|-------------|-------------|--------------|
| 6 FOSTER 200 | 4 CLAY 400 | 1 BROWN 200 | 5 PETERS 150 |
| 8 BAKER 310 | 3 JONES 310 | 7 GRAY 310 | 2 SMITH 310 |

Department

| Dept | Name |
|------|-----------|
| PK | |
| UPI | |
| 150 | PAYROLL |
| 200 | FINANCE |
| 310 | MFG. |
| 400 | EDUCATION |
| 450 | ADMIN |

Spool file after redistribution on Employee.Dept row hash

| | | | |
|--------------|-------------|--------------|-------------|
| 5 PETERS 150 | 7 GRAY 310 | 1 BROWN 200 | 4 CLAY 400 |
| | 3 JONES 310 | 6 FOSTER 200 | 9 TYLER 450 |
| | 8 BAKER 310 | 10 CARR 450 | |
| | 2 SMITH 310 | | |

Department rows hash distributed on Department.Dept (UPI)

| | | | |
|-------------|----------|-------------|---------------|
| 150 PAYROLL | 310 MFG. | 200 FINANCE | 400 EDUCATION |
| | | 450 ADMIN | |

Дублирование таблицы по узлам

Table ➔

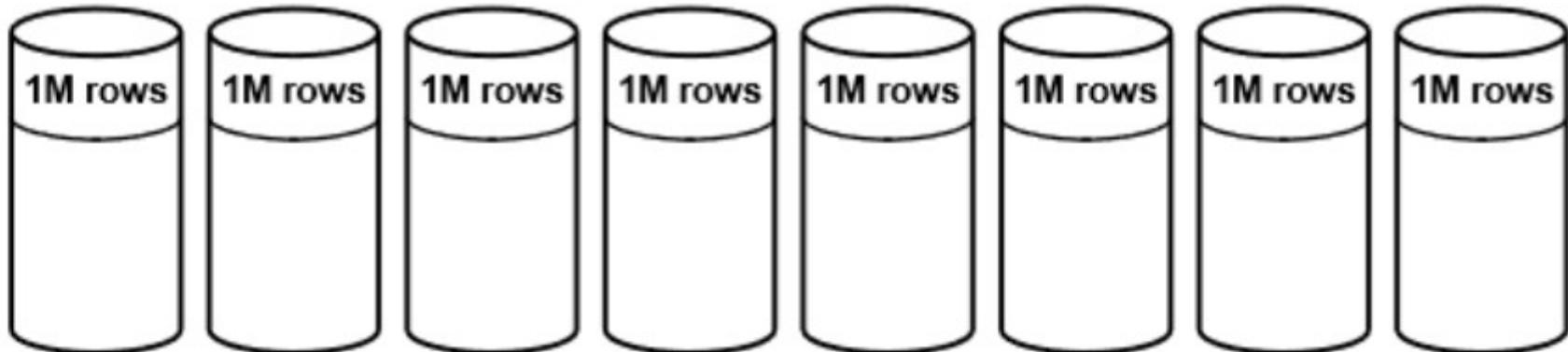
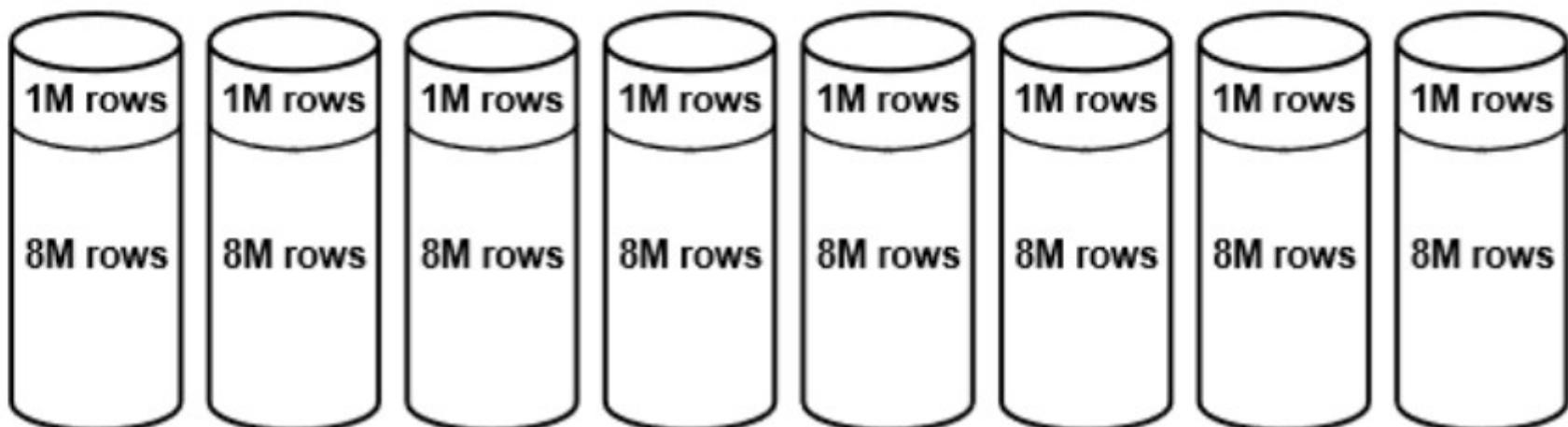


Table ➔

SPOOL
(Table is duplicated
on each AMP)



Merge Join

DUPLICATE and SORT the Smaller Table on all AMPs.
LOCALLY BUILD a copy of the Larger Table and SORT.

Employee

| Enum | Name | Dept |
|------|--------|------|
| PK | | FK |
| UPI | | |
| 1 | BROWN | 200 |
| 2 | SMITH | 310 |
| 3 | JONES | 310 |
| 4 | CLAY | 400 |
| 5 | PETERS | 150 |
| 6 | FOSTER | 200 |
| 7 | GRAY | 310 |
| 8 | BAKER | 310 |
| 9 | TYLER | 450 |
| 10 | CARR | 450 |

Department

| Dept | Name |
|------|-----------|
| PK | |
| UPI | |
| 150 | PAYOUT |
| 200 | FINANCE |
| 310 | MFG. |
| 400 | EDUCATION |
| 450 | ADMIN |

Example:

```
SELECT  
FROM  
INNER JOIN  
ON
```

E.Enum, ...
Employee E
Department D
E.Dept = D.Dept ;

Department rows hash distributed on Department.Dept (UPI)

| | | | |
|-------------|----------|--------------------------|---------------|
| 150 PAYROLL | 310 MFG. | 200 FINANCE 450 ADMIN | 400 EDUCATION |
|-------------|----------|--------------------------|---------------|

Spool file after duplicating and sorting on Department.Dept row hash.

| | | | |
|---------------|---------------|---------------|---------------|
| 150 PAYROLL | 150 PAYROLL | 150 PAYROLL | 150 PAYROLL |
| 200 FINANCE | 200 FINANCE | 200 FINANCE | 200 FINANCE |
| 310 MFG. | 310 MFG. | 310 MFG. | 310 MFG. |
| 400 EDUCATION | 400 EDUCATION | 400 EDUCATION | 400 EDUCATION |
| 450 ADMIN | 450 ADMIN | 450 ADMIN | 450 ADMIN |

Employee rows hash distributed on Employee.Enum (UPI)

| | | | |
|--------------|-------------|-------------|--------------|
| 6 FOSTER 200 | 4 CLAY 400 | 1 BROWN 200 | 5 PETERS 150 |
| 8 BAKER 310 | 3 JONES 310 | 7 GRAY 310 | 2 SMITH 310 |

Spool file after locally building and sorting on Employee.Dept row hash

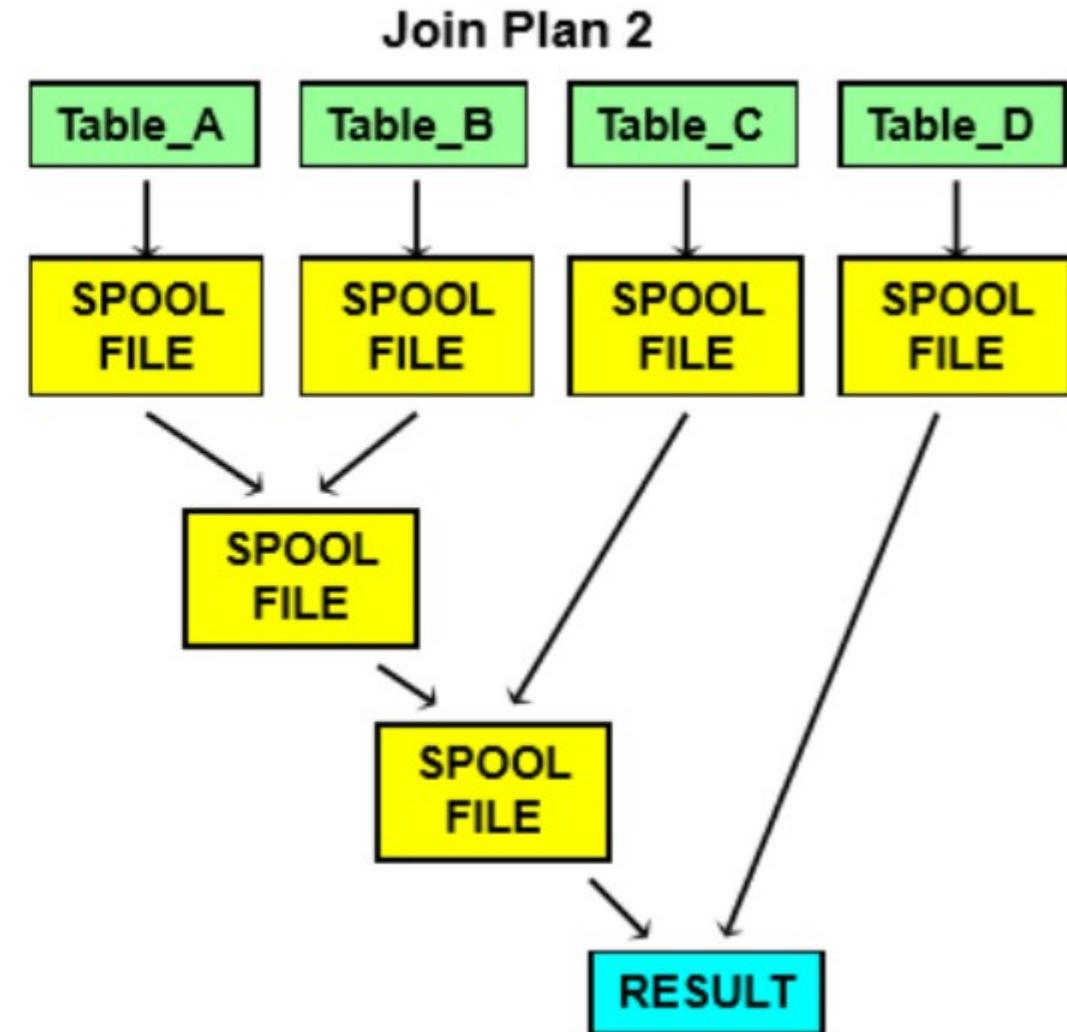
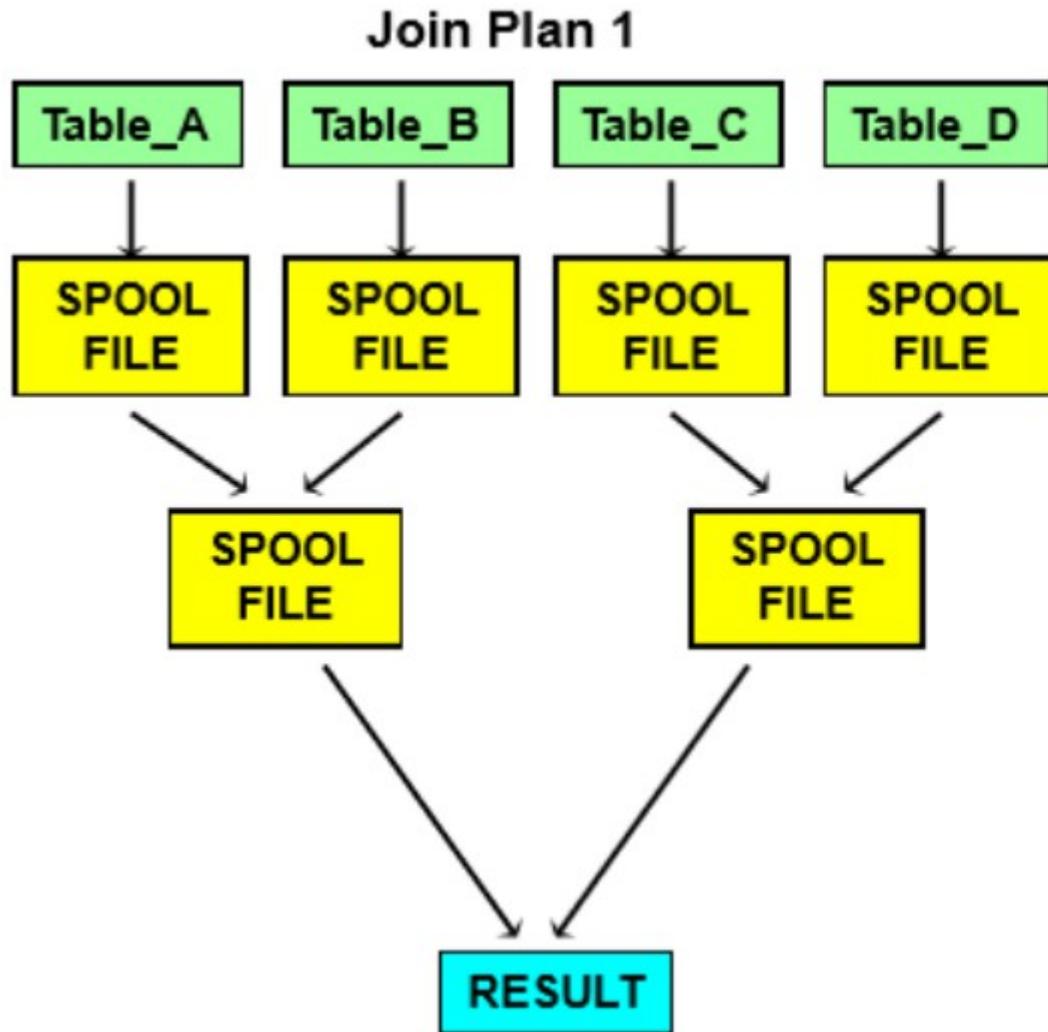
| | | | |
|--------------|-------------|-------------|--------------|
| 6 FOSTER 200 | 3 JONES 310 | 1 BROWN 200 | 5 PETERS 150 |
| 8 BAKER 310 | 4 CLAY 400 | 7 GRAY 310 | 2 SMITH 310 |

Product Join

Product join работает по принципу классического алгоритма Nested Loops, но со следующими особенностями:

- Работает с предикатами любой сложности;
- Предполагает дублирование таблицы по всем узлам.

Сложный join



Vertica HP



Vertica

- Разработал Майкл Стоунбрейкер.
- Компания Vertica основана в 2005 году.
- Куплена корпорацией Hewlett Packard в 2011 году.

VERTICA

Vertica – особенности

От обычной СУБД ее отличают несколько ключевых признаков:

- Колоночное хранение
- Сжатие данных
- Проекции — вместо классических индексов

В отличие от традиционных баз, хранящих данные в формате строк, Vertica хранит данные в виде колонок в сжатом виде. Такое хранение позволяет иметь большую степень сжатия, освобождая много дискового пространства.

RLE – Run-length encoding

Кодирование длин серий (англ. run-length encoding, RLE)

Алгоритм заменяет подряд идущие одинаковые символы на пару: (символ, число его повторений).

Такое преобразование относят к классу алгоритмов сжатия без потерь, поскольку оно полностью обратимо.

Пример: XXAAAAAY → 2X5A1Y.

RLE

- Основным плюсом группы алгоритмов RLE является простота и скорость работы (в том числе и скорость декодирования)
- Главным минусом является неэффективность на неповторяющихся наборах символов.

Часть проблем можно решить модификацией алгоритма. Например, кодировать только серии из 2+ символов.

Vertica — MPP

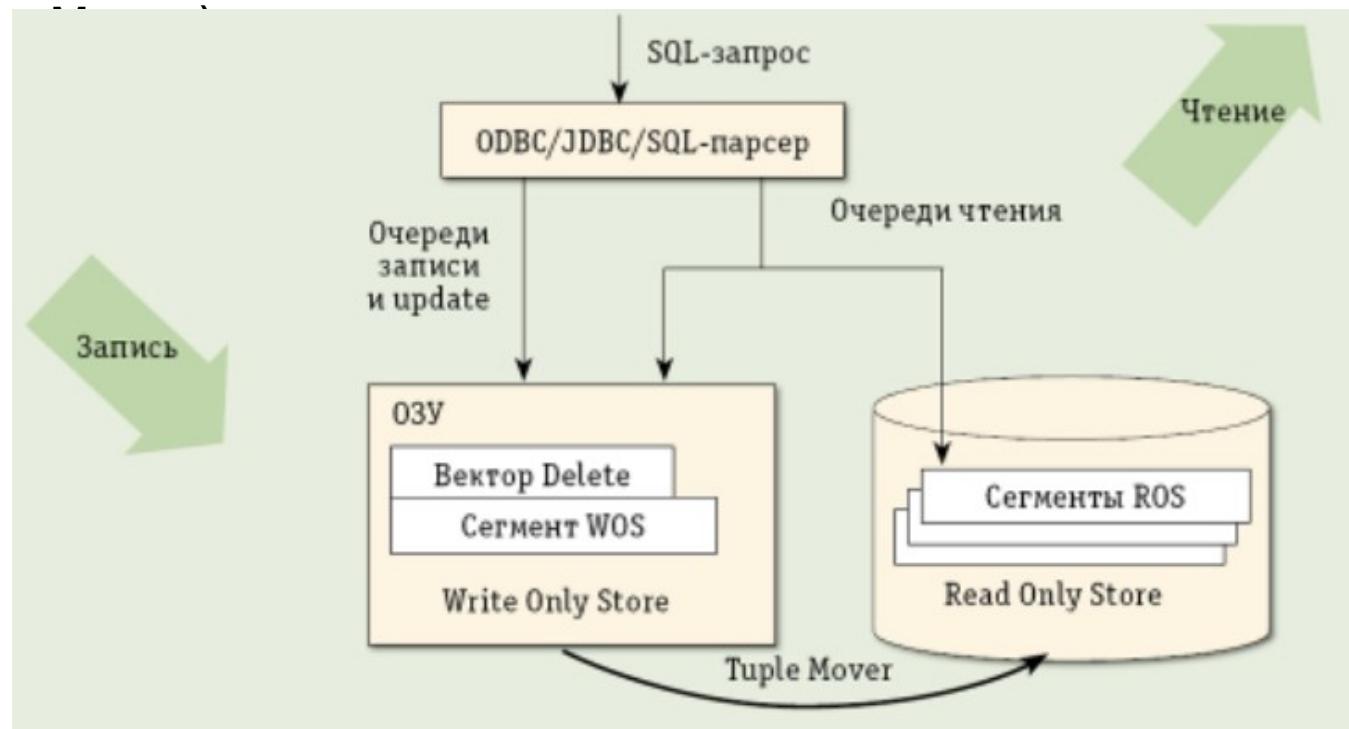
Vertica спроектирована для работы на MPP-кластере (massive parallel processing).

Vertica позволяет легко масштабировать свой кластер с помощью добавления стандартных серверов общего применения, таким образом до определенной степени сокращая аппаратные расходы.

Vertica MPP RDBMS

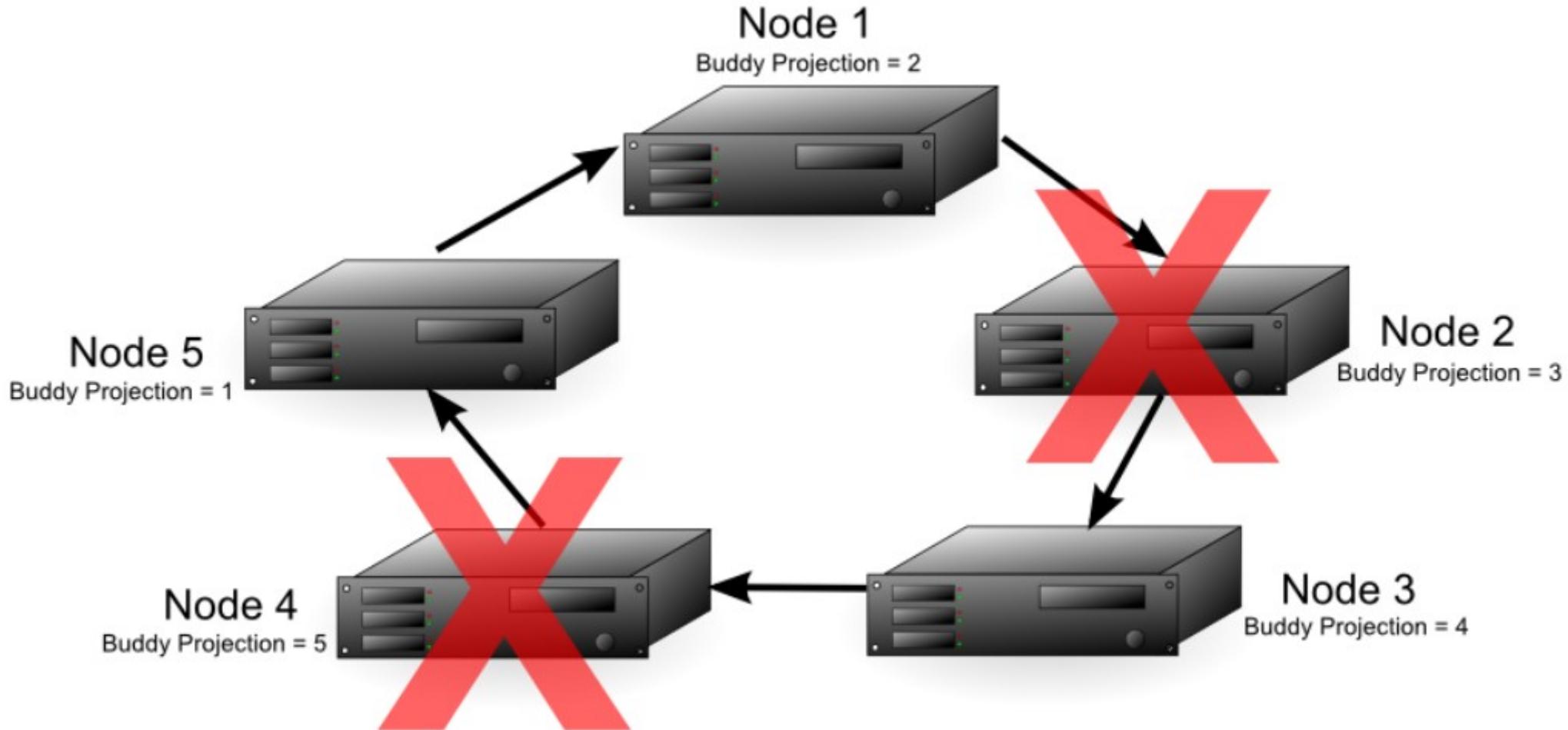
В Vertica существует двухкомпонентное хранилище данных, состоящие из:

- WOS (write optimized store) – хранилище в оперативной памяти
- ROS (read optimized store) – хранилище на диске
- Асинхронный процесс, осуществляющий перемещение данных между хранилищами (Tuple Mover)



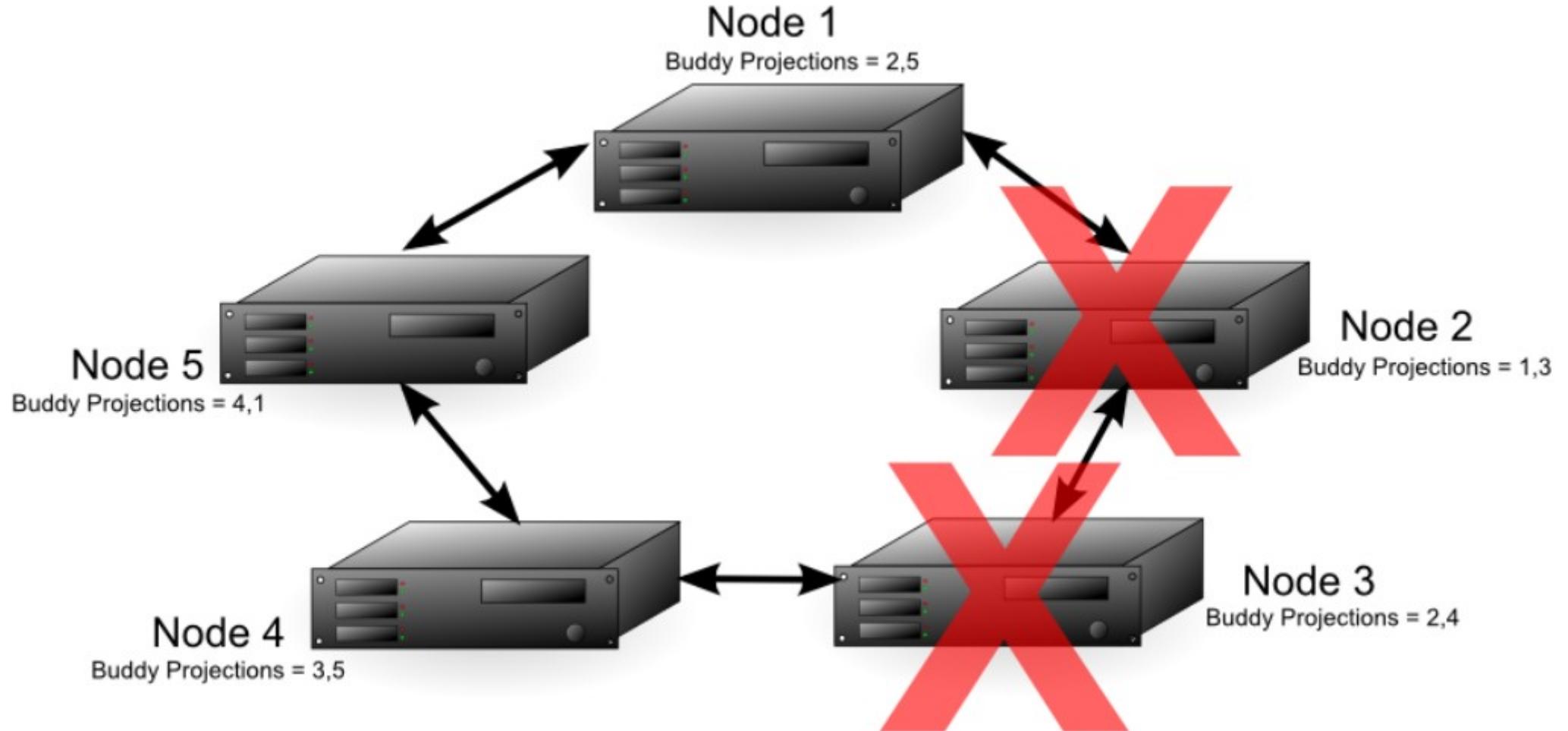
K-safety

К-безопасность устанавливает отказоустойчивость на кластере баз данных Vertica. K-safety = 1



K-safety

K-safety = 2



Vertica – Проекции

Проекции – реализуют оптимизированное хранение данных.

В Vertica нет понятия индексов, а таблица — это логическая структура хранения, а не физическая. Данные хранятся в виде проекций.

Проекция — это некий аналог материализованного представления, которое является единицей физического хранения данных в Vertica.

Информация в проекциях может дублироваться для обеспечения быстрого доступа к данным.

Проекции в *Vertica*

Проекции чем-то похожи на индексы, они хранят данные по всем или не всем столбцам таблицы.

Может быть более одной проекции на таблицу, проекции могут хранить отсегментированные и отсортированные данные по разным правилам.

Данные во всех проекциях автоматически обновляются при обновлении записей таблицы.

Виды проекций в Vertica

Виды проекций:

- суперпроекции (Superprojection),
- запрос-ориентированные проекции (Query-Specific Projections),
- агрегированные проекции (Aggregate Projections).

Clickhouse



ClickHouse

ClickHouse — это колоночная аналитическая СУБД с открытым кодом, позволяющая выполнять аналитические запросы в режиме реального времени на структурированных больших данных, разрабатываемая компанией Яндекс.

Разработан для решения задач веб-аналитики



ClickHouse - секционирование

В ClickHouse имеются секционированные таблицы, состоящие из указанного набора узлов. Здесь нет «центральной власти» или сервера метаданных. Все узлы, между которыми разделена та или иная таблица, содержат полные, идентичные копии метаданных, включая адреса всех остальных узлов, на которых хранятся секции этой таблицы.

Шардинг кластера ClickHouse

Шардинг в ClickHouse позволяет записывать и хранить фрагменты данных в распределенном кластере и обрабатывать данные параллельно на всех узлах кластера, увеличивая пропускную способность и уменьшая задержку.

Для шардирования используется движок `Distributed`.

Он не хранит данные самостоятельно, а делегирует запросы `SELECT` к таблицам сегментов, позволяет обрабатывать запросы распределённо, на нескольких серверах. Чтение автоматически распараллеливается.

Запись данных в сегменты может выполняться в двух режимах:

1. через распределенную таблицу и дополнительный ключ сегментирования;
2. непосредственно в таблицы сегментов, из которых затем данные будут считываться через распределенную таблицу.

Репликация кластера ClickHouse

ClickHouse поддерживает репликацию данных, обеспечивая целостность данных на репликах. Для репликации данных используются специальные движки таблиц семейства MergeTree:

- ReplicatedMergeTree;
- ReplicatedSummingMergeTree;
- ReplicatedAggregatingMergeTree;
- ReplicatedCollapsingMergeTree; . и т.д

```
CREATE TABLE table_name(  
    EventDate DateTime,  
    CounterID UInt32,  
    UserID UInt32,  
    ver UInt16  
) ENGINE =  
ReplicatedReplacingMergeTree('/clickhouse/tables/{layer}-  
{shard}/table_name', '{replica}', ver)  
PARTITION BY toYYYYMM(EventDate)  
ORDER BY (CounterID, EventDate, intHash32(UserID))  
SAMPLE BY intHash32(UserID);
```

Clickhouse – движки таблиц

Семейство MergeTree

Общим свойством этих движков является быстрая вставка данных с последующей фоновой обработкой данных

Семейство Log

Простые движки для быстрой записи больших массивов данных в небольшие таблицы

- TinyLog
- StripeLog
- Log

Движки таблиц для интеграции

Движки для связи с другими системами хранения и обработки данных.

- Kafka
- MySQL
- JDBC
- S3
- ...

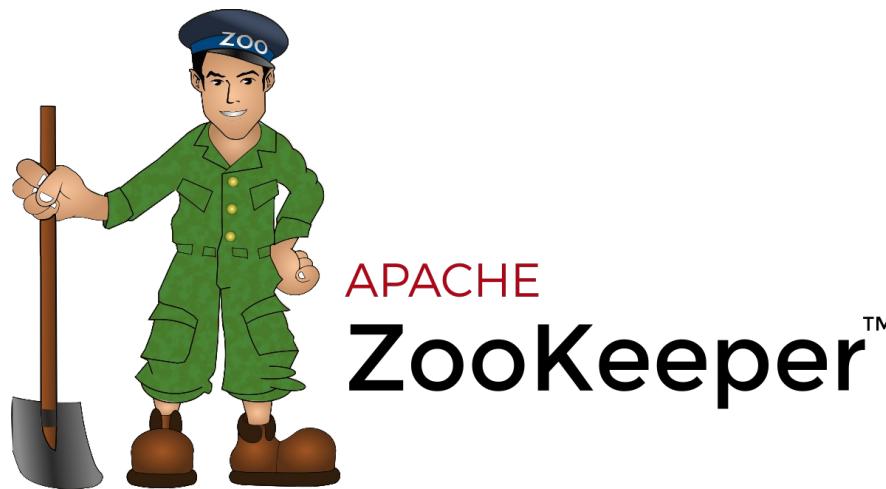
Специальные движки

- | | |
|---|---|
| <ul style="list-style-type: none">• ODBC• JDBC• MySQL• ... | <ul style="list-style-type: none">• URL• MaterializedView• Memory |
|---|---|

Репликация кластера ClickHouse

За репликацию отвечает Zookeeper

Шардинг и репликация данных полностью независимы. Шардинг — естественная часть ClickHouse, в то время как репликация сильно зависит от Zookeeper, который используется для уведомления реплик об изменениях состояния данных.



ClickHouse – особенности

- Столбцовая СУБД
- Сжатие данных
- Линейная масштабируемость.
- СУБД оптимизирована для работы на HDD-дисках. Можно обрабатывать данные, которые не помещаются в оперативную память.
- Параллельная обработка запроса на многих процессорных ядрах
- Частичная поддержка SQL (Зависимые подзапросы и оконные функции не поддерживаются.)
- Наличие индексов(данные сортируются по первичному ключу)
- Поддержка приближённых вычислений

ClickHouse — достоинства

- скорость;
- масштабируемость;
- расширяемость;
- высокая доступность и отказоустойчивость;
- простота развертывания и удобство эксплуатации.

ClickHouse — недостатки

- Отсутствие полноценных транзакций.
- Ограниченная поддержка операций JOIN.
- Нет возможности изменять или удалять ранее записанные данные с низкими задержками и высокой частотой запросов.
- Разреженный индекс делает ClickHouse плохо пригодным для точечных чтений одиночных строк по своим ключам.
- Зависимость от оперативной памяти.
- Отсутствие полноценного оптимизатора запросов.
- Низкая производительность небольших вставок.

ClickHouse — ограничения SQL

Использует собственный диалект SQL, не соответствующий стандарту, но содержащий расширения (массивы, функции для работы с URL) В частности:

- update/delete — асинхронные через alter table;
- плохо джойнит большое количество таблиц;
- отсутствие оконных функций и зависимых подзапросов.

ClickHouse — выводы

ClickHouse обладает высокой производительностью, простой и удобный в использовании.

Предназначен для OLAP сценария. Хорошо подходит для аналитики и отчётов в реальном времени.

Не подходит для OLTP. Имеет некоторые ограничения.

Если подразумевается выполнять много JOIN'ов не подходит.

Спасибо
за
внимание!

