[⭐⭐] Соединения

Files & media

CROSS JOIN является, по сути, Декартовым произведением всех элементов выбранных таблиц, другими словами — это комбинация всех элементов одной таблицы со всеми элементами других таблиц. Такой **JOIN** ещё могут называть **кортежем** выбранных таблиц.

JOIN испольуется для того, чтобы соединить данные из нормализованных таблиц и **денормализовать их для дальнейшего анализа и обработки**. Ключевое слово **JOIN** используется для объединения **столбцов** двух и более **таблиц**, основываясь на взаимосвязанных элементах между ними (обычно с помощью **SELECT**, **DELETE** и **UPDATE)**.

▼ MAX/MIN записей

Тип соединения	Минимальное число строк	Максимальное число строк	Сложность алгоритма
NATURAL JOIN	0	min(a, b)	O(a * b) (сравниваются только столбцы с одинаковыми именами и типами)
SELF JOIN	0	a * (a-1)	O(a * a) (соединение таблицы с самой собой, сложность зависит от числа строк в таблице)
ANTI JOIN	0	a	O(a + b) (возвращает строки только из левой таблицы, которые не имеют совпадений в правой)
SEMI JOIN	0	a	O(a + b) (возвращает только строки из левой таблицы, для которых есть совпадения в правой)

Тип соединения	Минимальное число строк	Максимальное число строк	Сложность алгоритма
INNER JOIN	0	min(a, b)	O(a * b) (в худшем случае сравниваются все строки)
LEFT JOIN (LEFT OUTER JOIN)	a	a * b	O(a * b) (аналогично INNER JOIN, но всегда возвращает все строки из левой таблицы)
RIGHT JOIN (RIGHT OUTER JOIN)	b	a * b	O(a * b) (аналогично LEFT JOIN, но с правой таблицей)
FULL JOIN (FULL OUTER JOIN)	max(a, b)	a + b	O(a * b) (соединение с учётом всех строк из обеих таблиц)
CROSS JOIN	a * b	a * b	O(a * b) (всегда возвращает декартово произведение, соединяет каждую строку первой таблицы с каждой строкой второй таблицы)
Broadcast JOIN (распределённый)	0	a * b	O(a + b) (одна из таблиц передаётся всем узлам, эффективен для маленьких таблиц)
Repartition JOIN (распределённый)	0	a * b	O(a log a + b log b) (перераспределение данных по ключу соединения, после чего выполняется локальный JOIN)
ZigZag JOIN (распределённый)	0	a * b	O(a * log b) (оптимизация с использованием двух Bloom фильтров, снижает количество ненужных строк)
Semi-JOIN (распределённый)	0	a	O(a + b) (возвращает только строки из первой таблицы, минимизируя объём передаваемых данных)
Bloom JOIN (распределённый)	0	a * b	O(a + b) (использование Bloom фильтров для исключения строк без совпадений)

Тип соединения	Минимальное число строк	Максимальное число строк	Сложность алгоритма
PERF JOIN (распределённый)	0	a * b	O(a + b) (использует битмапы для фильтрации строк, улучшает производительность при дублирующихся значениях)
Track JOIN (распределённый)	0	a * b	O(a + b) (минимизирует сетевой трафик, требует полного сканирования таблиц для настройки передачи данных)

▼ Виды JOIN

Есть классические и те, которые используются в распределенных системах.

💗🎀Классические

▼ INNER JOIN

Данный вид **JOIN** является опцией по умолчанию (если вы не укажите ничего перед ключевым словом). Он возвращает значения, которые присутствуют как в первой, так и в последующих таблицах.

> **Обрати внимание!**
>
> Строки с **NULL** в столбце, который участвует в соединении, не будут включены в результат, так как **NULL** не считается равным чему-либо, даже другому **NULL**.

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.Cu
stomerID;
```

▼ LEFT (OUTER) JOIN

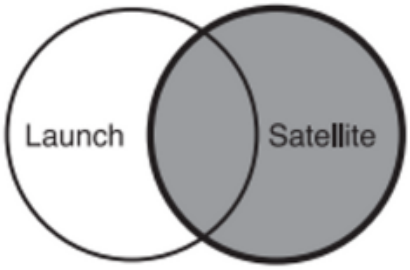
Данный вид **JOIN** возвращает все значения из первой таблицы, а также все совпадающие значения из последующих таблиц. В случае отсутствия совпадений (с правой таблицей) выдаёт **NULL** в колонках со значениями из этой таблицы.

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.Custom
erID;
```

▼ RIGHT (OUTER) JOIN

Данный вид JOIN возвращает все значения из последней таблицы, а также все совпадающие значения из предыдущих таблиц. В случае отсутствия совпадений (с левой таблицей) выдаёт **NULL** в колонках со значениями из этой таблицы.

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
RIGHT JOIN Customers ON Orders.CustomerID = Customers.Cu
stomerID;
```

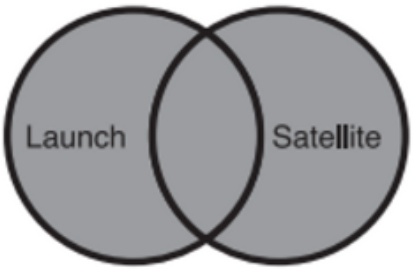


▼ FULL (OUTER) JOIN

> **Обрати внимание!**
>
> Не путать с **CROSS JOIN**.

Данный вид **JOIN** возвращает все значения, совпадающие либо с значениями из первой таблицы, либо с значениями одной из последующих таблиц. В случае отсутствия совпадений выдаёт **NULL** по колонкам.

```
SELECT Customers.CustomerName, Orders.OrderID
FROM CustomersFULL
OUTER JOIN Orders ON Customers.CustomerID = Orders.Custo
merID;
```



▼ SEMI JOIN

Semi Join возвращает только те строки из левой таблицы, для которых существует совпадение в правой таблице. В отличие от обычного `JOIN`, который возвращает данные из обеих таблиц, `Semi Join` возвращает только данные из левой таблицы.

В PostgreSQL можно смоделировать **Semi Join** с помощью `EXISTS`:

```
SELECT *
FROM table1 t1
WHERE EXISTS (
    SELECT 1
    FROM table2 t2
    WHERE t1.column = t2.column
);
```

Этот запрос выбирает все строки из `table1`, для которых существуют совпадающие строки в `table2`.

> **Обрати внимание!**
>
> - **INNER JOIN**: возвращает пересечение данных, но **с данными обеих таблиц**.
> - **SEMI JOIN**: возвращает только данные из левой таблицы, которые **имеют совпадения** в правой таблице, **без возврата данных из правой таблицы**.
>
> ---
>
> - **NULL** значения в столбцах для соединения исключают строки с **NULL**. (из-за сравнения)

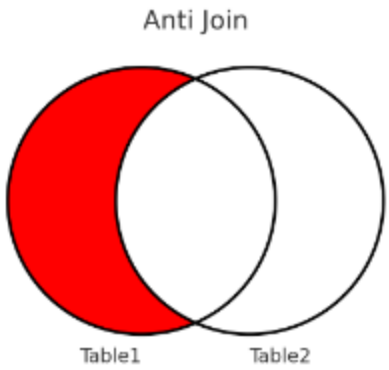
▼ ANTI JOI

Anti Join — это противоположность **Semi Join**. Он возвращает строки из левой таблицы, для которых **не существует** совпадений в правой таблице.

В PostgreSQL это можно сделать с помощью `NOT EXISTS`:

```
SELECT *
FROM table1 t1
WHERE NOT EXISTS (
    SELECT 1
    FROM table2 t2
    WHERE t1.column = t2.column
);
```

Этот запрос выбирает все строки из `table1`, которые **не имеют** совпадений в `table2`.

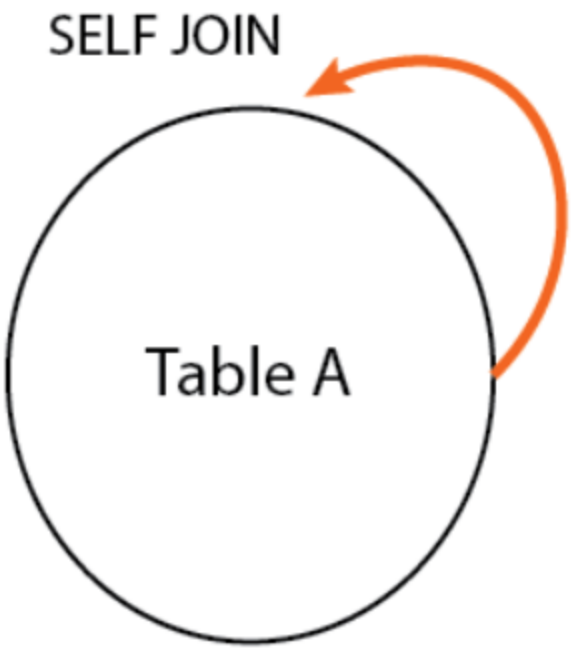


▼ SELF JOIN

Данный вид **JOIN** работает в области соединения общих элементов в пределах одной таблицы. По этой причине во время описания параметров соединения активно используются операторы присваивания новых имён (**Alias, As**).

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName A
S CustomerName2, A.City
```

```
FROM Customers AS A, Customers AS B
WHERE A.CustomerID <> B.CustomerID AND A.City = B.City;
```

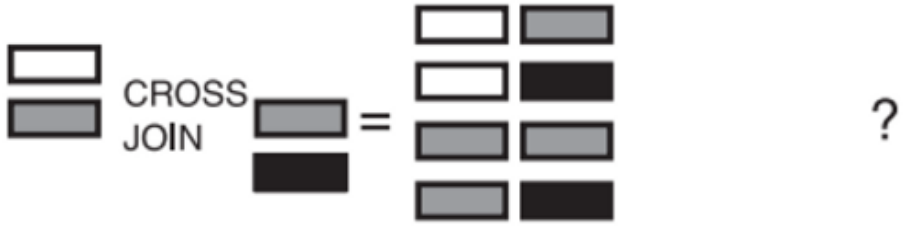


▼ CROSS JOIN

> **Обрати внимание!**
>
> - Данный вид **JOIN** практически не используется вследствие нагрузки, вызванной подобным количеством чтений.
> - **В CROSS JOIN NULL** значения включаются как есть.

CROSS JOIN является, по сути, Декартовым произведением всех элементов выбранных таблиц, другими словами — это комбинация всех элементов одной таблицы со всеми элементами других таблиц. Такой **JOIN** ещё могут называть **кортежем** выбранных таблиц.

```
SELECT column_name(s)
FROM table1
CROSS JOIN table2;
```



▼ NATURAL JOIN

NATURAL JOIN автоматически соединяет две таблицы по всем столбцам с одинаковыми именами и типами данных. Он не требует явного указания столбцов для соединения. Это удобно, но может быть потенциально опасным, если столбцы с одинаковыми именами не предназначены для соединения.

Особенности:

- Работает только по столбцам с одинаковыми именами.
- Используется для упрощения синтаксиса, но важно внимательно следить за именами столбцов в таблицах, чтобы избежать неожиданных соединений.

Пример:

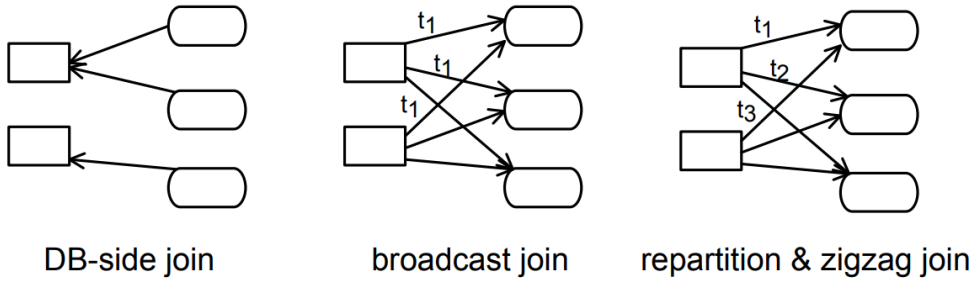
```
SELECT *
FROM Customers
NATURAL JOIN Orders;
```

Этот запрос соединит таблицы `Customers` и `Orders` по всем общим столбцам, например, по `CustomerID`, если он есть в обеих таблицах.

Минусы:

- Меньший контроль над процессом соединения.
- Риск неожиданного соединения по ненужным столбцам, если их имена совпадают.

💗Распределенные системы



▼ Broadcast JOIN

Маленькая таблица передаётся (или "распространяется") на все узлы кластера, чтобы соединение с другой, большей таблицей могло быть выполнено локально на каждом узле.

Преимущество: Уменьшает сетевой трафик, так как данные передаются только один раз для маленькой таблицы.

Недостаток: Применимо только для небольших таблиц.

▼ Repartition JOIN

Оба набора данных перераспределяются (репартицируются) по ключу соединения. Каждая строка с одинаковым значением ключа попадает на один узел, что позволяет выполнить соединение локально.

Преимущество: Подходит для больших таблиц, так как обе таблицы перераспределяются по узлам.

Недостаток: Требует передачи большого объёма данных по сети для перераспределения.

▼ ZigZag JOIN

Использует два Bloom фильтра для фильтрации ненужных строк с обеих сторон соединения. Один из фильтров применяется к одной таблице, второй к другой, что позволяет уменьшить объём данных, участвующих в соединении.

Преимущество: Эффективно при работе с большими таблицами, особенно если одна из них значительно меньше.

Недостаток: Требует два сканирования одной из таблиц, что увеличивает время обработки.

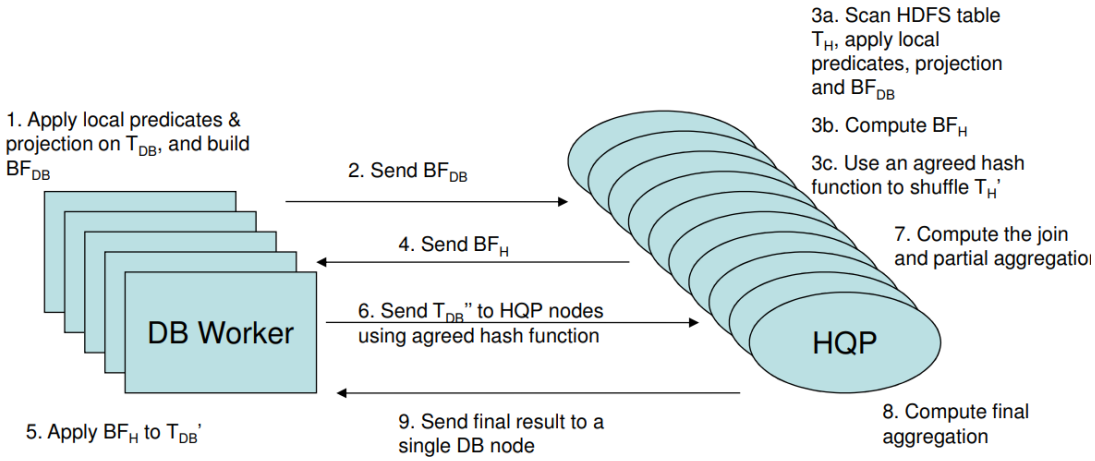


Figure 4: Data flow of zigzag join

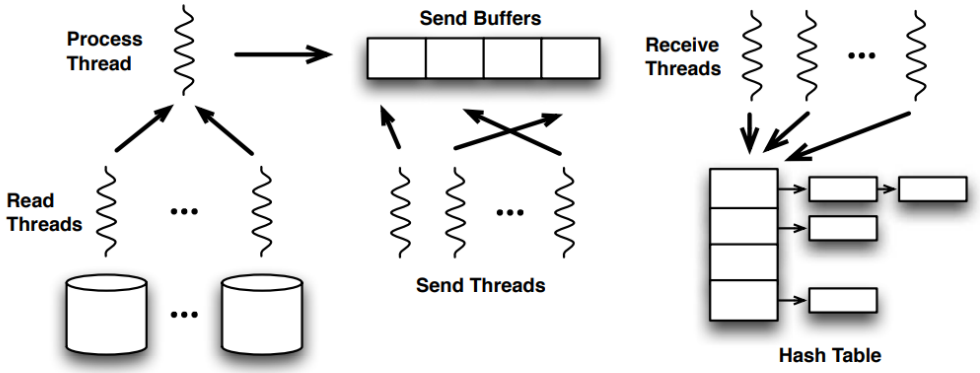


Figure 7: Interleaving of scanning, processing and shuffling of HDFS data in zigzag join

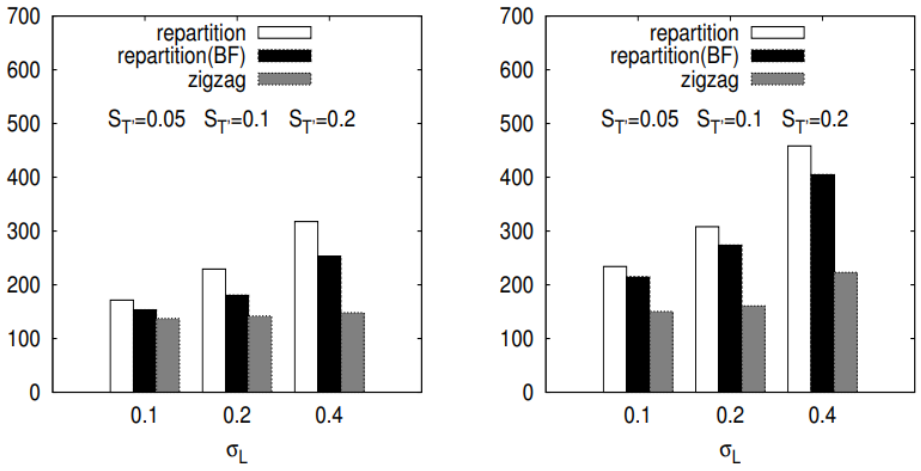


Figure 8: Zigzag join vs repartition joins: execution time (sec)

▼ Semi-JOIN

Используется для уменьшения объёма данных, передаваемых между узлами. Возвращает только строки из одной таблицы, у которых есть совпадения в другой таблице.

Преимущество: Уменьшает сетевой трафик, так как передаются только необходимые строки.

Недостаток: Возвращает только строки из одной таблицы, а не объединённые данные.

▼ Bloom JOIN

Использует **Bloom фильтры** для исключения строк, которые не имеют совпадений в другой таблице, ещё до основного соединения.

Преимущество: Значительно уменьшает количество данных, передаваемых по сети, исключая ненужные строки на этапе фильтрации.

Недостаток: Эффективность зависит от правильной настройки Bloom фильтров, может быть сложен в настройке.

▼ PERF JOIN

Оптимизированное соединение, использующее битмапы вместо Bloom фильтров для фильтрации данных.

Преимущество: Быстрое выполнение за счёт использования битмапов для фильтрации строк.

Недостаток: Может быть менее эффективен при большом количестве уникальных значений в ключе соединения.

▼ Track JOIN

Оптимизирует соединения, минимизируя сетевой трафик за счёт передачи данных по ключам соединения, а не по всей таблице.

Преимущество: Снижает нагрузку на сеть и увеличивает производительность при работе с большими таблицами.

Недостаток: Требует полного сканирования таблиц перед соединением для настройки передачи данных.

▼ Bloom фильтр

Bloom фильтр — это вероятностная структура данных, которая используется для проверки, принадлежит ли элемент множеству. Он позволяет с высокой степенью вероятности определить, **включён ли элемент** в множество, однако может дать **ложно-положительные результаты** (т.е. указать, что элемент существует в множестве, хотя это не так), но **никогда не даст ложно-отрицательных результатов** (если элемент отсутствует, это гарантированно верно).

Основные характеристики Bloom фильтра:

1. **Эффективность**:
   - **Bloom фильтр** работает очень эффективно по памяти и времени, так как требует небольшое количество памяти и времени для выполнения операций вставки и поиска.
2. **Ложно-положительные результаты**:

- Хотя фильтр гарантированно указывает на отсутствие элемента, при нахождении элемента он может ошибочно указать на его наличие (даже если его там нет).

3. **Отсутствие удаления**:
   - Операции удаления элементов в стандартных Bloom фильтрах не поддерживаются, поскольку удаление одного элемента может случайно повлиять на другие элементы в фильтре.

Как работает Bloom фильтр:

1. **Инициализация**:
   - Bloom фильтр представляет собой битовый массив длины **m**, инициализированный нулями.
2. **Хеш-функции**:
   - Для каждого элемента, который нужно добавить в фильтр, используется **k различных хеш-функций**, каждая из которых вычисляет индекс в битовом массиве. Эти индексы устанавливаются в значение 1.
3. **Проверка принадлежности**:
   - Чтобы проверить, есть ли элемент в множестве, те же **k хеш-функций** применяются к элементу, чтобы получить индексы в битовом массиве. Если все индексы содержат значение 1, элемент, скорее всего, принадлежит множеству. Если хотя бы один бит равен 0, элемент точно не принадлежит множеству.

Пример работы Bloom фильтра:

Представим, что у нас есть множество имен, и мы хотим быстро проверять, принадлежит ли новое имя этому множеству. Мы добавляем имена с помощью нескольких хеш-функций, каждая из которых устанавливает несколько битов в битовом массиве.

1. Добавляем имя "Alice" в фильтр: Хеш-функции возвращают индексы 3, 7 и 15, и биты в этих позициях становятся равны 1.
2. Добавляем имя "Bob": Хеш-функции возвращают индексы 1, 4 и 9, и эти биты становятся 1.

Когда мы хотим проверить наличие имени "Charlie":

- Хеш-функции возвращают индексы 2, 5 и 10. Поскольку хотя бы один из этих битов (например, 2-й) равен 0, мы точно знаем, что "Charlie" не находится в фильтре.

Если мы хотим проверить "Alice", то все соответствующие биты (3, 7 и 15) установлены в 1, и мы считаем, что "Alice" есть в фильтре. Однако возможен случай, когда эти биты установлены и для другого элемента, что создаёт **ложно-положительный результат**.

Использование Bloom фильтров в распределённых системах (например, для JOIN-ов):

В распределённых системах, таких как Hadoop или Spark, **Bloom фильтры** часто используются для оптимизации соединений (JOIN) и других операций с большими объёмами данных:

1. **Уменьшение трафика**: Перед выполнением соединения между двумя таблицами, Bloom фильтр может быть применён к одной из таблиц, чтобы исключить строки из другой таблицы, которые не имеют совпадений. Это позволяет сократить объём передаваемых данных между узлами кластера.
2. **Оптимизация распределённых JOIN-ов**: При использовании Bloom фильтров строки, которые не могут участвовать в соединении, исключаются до основного процесса соединения, что снижает нагрузку на вычислительные ресурсы.

✅Преимущества:

- **Эффективность по памяти**: Использует очень мало памяти по сравнению с другими структурами данных.
- **Быстрота**: Очень быстро работает, так как использует простые операции с хеш-функциями и битовыми массивами.
- **Полезен для больших данных**: В распределённых системах уменьшает объём передаваемых и обрабатываемых данных.

❌Недостатки:

- **Ложно-положительные результаты**: Может указать на наличие элемента, которого нет в фильтре.
- **Нельзя удалить элементы**: Обычные Bloom фильтры не поддерживают операцию удаления элементов.

Заключение:

Bloom фильтр — это отличное решение для задач, связанных с проверкой принадлежности элемента множеству при ограниченных ресурсах памяти, и он широко используется в распределённых системах для оптимизации операций соединения, фильтрации и работы с большими данными.

▼ Алгоритмы соединения

Алгоритмы соединения — способ того, **по какому принципу** мы будем соединять между собой **две** (если таблиц больше, то соединяются две в промежуточную, потом добавляется еще одна и так до бесконечности) **таблицы**. В зависимости от того, какие у нас данные, отсортированы они или нет и какой тип соединения, оптимизатор подбирает алгорим.

Одним из способов улучшения производительности базы данных является **оптимизация выражений с JOIN.** Оптимизатор запросов автоматически выбирает лучший вариант соединения, но можно указать **явные хинты для влияния на выбор алгоритма**:

```
<join_hint> ::=
     { LOOP | HASH | MERGE | REMOTE }
```

▼ Nested Loop

Обрати внимание

Данный аргумент несовместим с параметрами соединения **RIGHT** и **FULL**.

🔤 **Nested Loop** — Соединение с использованием вложенного цикла. Внешняя таблица проходит построчно, для каждой строки ищутся совпадения во внутренней таблице.

Сложность: O(n * m), где **n** — количество строк в внешней таблице, **m** — во внутренней. Если есть индекс, то **O(Log(N))**

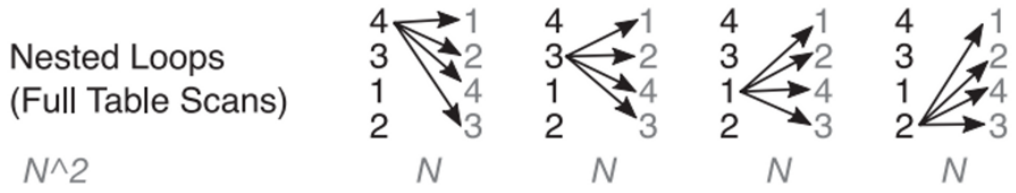
✅Когда используем:

- Маленькие таблицы с индексированными столбцами (менее 10 строк).
- Подходит для **небольших запросов**, где внутренняя таблица имеет **индексы** по ключам соединения.

Почему для небольших?

- Когда таблицы маленькие, они могут полностью поместиться в оперативной памяти. Это снижает затраты на операции ввода-вывода (I/O) и делает алгоритм Nested Loop Join еще более быстрым, так как данные всегда доступны для прямого доступа, без необходимости частого обращения к диску.
- Для небольших данных нет смысла тратить время на создание хэш-таблиц или сортировку. Эффективнее будет просто прогнать все со всем.

В таком случае **indexed nested loop join** будет наиболее быстрой операцией **JOIN** из-за наименьшего количества чтений и сравнений **(I/O)**.



▼ Sort Merge Join

🔤 **Sort Merge Join —** Алгоритм, который сначала сортирует обе таблицы по ключам соединения, а затем последовательно **объединяет** отсортированные наборы данных.

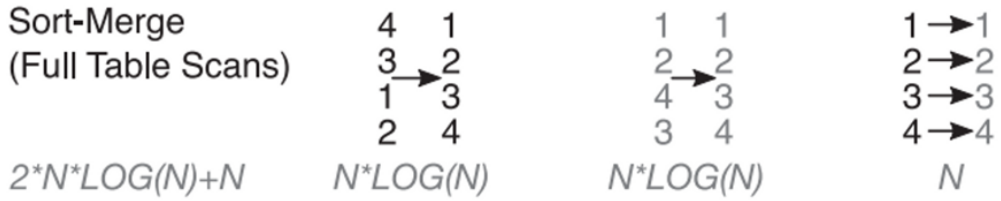
Сложность: O(n + m), где **n** и **m** — количество строк в таблицах.

✅Когда используем:

- Если обе стороны **JOIN** невелики, но **сортированы** по столбцу объединения (например, путем сканирования *отсортированных индексов*), то **MERGE JOIN** является **самой быстрой** операцией **JOIN**.
- **MERGE JOIN** сам по себе очень быстр, но может оказаться дорогостоящим вариантом, если требуются операции сортировки.

❌Когда не используем

- **Нечеткие условия соединения (неравенства)**: Sort Merge Join не может работать с условиями соединения, которые включают неравенства (`>`, `<`, `<>` и т.д.), так как для этих условий нельзя эффективно выполнить операцию слияния.



▼ Hash (перевод + конспект)

Hash Join — Алгоритм, который строит хэш-таблицу по одной из таблиц (обычно меньшей) и использует её для быстрого поиска соответствующих записей во второй таблице. Применим для соединений по равенству.

Сложность алгоритма

Алгоритмическая сложность **Hash Join** зависит от нескольких факторов, включая количество записей в обеих таблицах и доступную память. Обычно его сложность оценивается как:

- **Операции построения хэш-таблицы**: Для одной таблицы (обычно меньшей) создается хэш-таблица. Если в таблице **R** содержится **N** записей, то построение хэш-таблицы имеет сложность **O(N)**.
- **Поиск соответствий во второй таблице**: Для каждой записи из второй таблицы **S** выполняется поиск в хэш-таблице. Если во второй таблице **M** записей, то поиск для каждой записи будет иметь сложность **O(1)** (в среднем), что дает общую сложность **O(M)** для поиска.

Таким образом, общая **средняя сложность Hash Join**:

- **O(N + M)** — для создания хэш-таблицы и последующего поиска.

Однако в худшем случае, при неравномерном распределении данных в хэш-таблице или при нехватке оперативной памяти (когда происходит сброс на диск), сложность может увеличиться из-за коллизий в хэш-таблице или операций с диском.

HASH JOIN запросы могут эффективно обрабатывать **большие, неотсортированные, неиндексированные** входные данные. Они полезны для промежуточных результатов в сложных запросах, поскольку:

- Промежуточные результаты не индексируются (если только они явно не сохраняются на диске и затем не индексируются) и часто не имеют подходящей сортировки для следующей операции в плане запроса.
- Оптимизаторы запросов оценивают только размеры промежуточных результатов. Поскольку в сложных запросах оценки могут быть очень неточными, алгоритмы обработки промежуточных результатов должны быть не только эффективными, но и изящно деградировать, если промежуточный результат окажется намного больше ожидаемого.

HASH JOIN позволяет сократить использование денормализации. Денормализация обычно используется для достижения более высокой производительности за счет сокращения операций **JOIN**, несмотря на опасность избыточности, например, несогласованности обновлений. **HASH JOIN** снижают необходимость денормализации. **HASH JOIN** позволяют использовать вертикальное разбиение (представление групп столбцов одной таблицы в отдельных файлах или индексах) для физического проектирования баз данных.

*В следующих разделах описаны различные типы **HASH JOIN**: **in-memory**, **grace** и **recursive**.*

▼ REMOTE

!Важно! Данный аргумент применим только к **INNER JOIN**.

Принцип работы + к чему применим:

- Соединение между таблицами, находящимися на разных серверах или узлах в распределённых системах.
- Применим для соединений данных, которые распределены по нескольким физическим местоположениям.

Производительность:

- **Сложность**: Зависит от сетевых задержек и объёма передаваемых данных.
- Эффективен при небольших данных, передаваемых между узлами.

Когда используем:

- В распределённых системах, где данные хранятся на разных серверах или кластерах.

Плюсы:

- Позволяет соединять данные из разных источников.
- Эффективен для небольших таблиц в удалённых системах.

Минусы:

- Может быть медленным при больших объёмах данных из-за сетевых задержек.
- Зависит от скорости сети и пропускной способности каналов связи.

▼ Стандарты оформления вида SQL-XX и ANSI-XX

Существуют принятые на родине **SQL** стандарты оформления кода, где **“-XX”** обычно указывает на год принятия стандарта. В контексте использования **JOIN** эти стандарты говорят нам следующее.

Дело в том, что практически любой **JOIN** можно записать альтернативно, заменяя его на комбинацию дискретных выражений и логических операторов по типу **WHERE**. В некоторых случаях такие формулировки не отличаются по оптимальности от своих аналогов. Однако сам факт использования этого ключевого слова позволяет отделить ***логику отношений*** элементов и ***логику фильтрации*** (**WHERE**), а также гораздо более человеко-читаем и понятен при взгляде со стороны.

Вывод: во всех случаях применения ***логики отношений*** используйте подходящий **JOIN**.

⚠️ **Инфа, про которую надо почитать, разобрать и красиво оформить: Источник: Jon Heller - Pro Oracle SQL Development_ Best Practices 6 глава (вроде) либо 7**

”ANSI join syntax and inline views are the keys to writing great SQL statements. The ANSI join syntax uses the JOIN keyword, instead of a comma-separated list of tables. Inline views are nested subqueries in the FROM clause.”

Что это значит? ANSI - American National Standards Institute, Organization for Standardization (ISO). То есть всякие стандарты. И вот ANSI join, это когда мы соединяем таблицы с помощью ключевого слова JOIN:

```
-- Аналог без JOIN
SELECT A.A, B.B, C.C
FROM aaa AS A, bbb AS B, ccc AS C
WHERE
    A.B = B.ID
AND B.C = C.ID
AND C.ID = @param
```

https://stackoverflow.com/questions/1599050/will-ansi-join-vs-non-ansi-join-queries-perform-differently - разбор, почему важно писать “как человек, а не через запятые”:

You should use the ANSI-92 syntax for several of reasons

- The use of the JOIN clause separates the relationship logic from the filter logic (the WHERE) and is thus cleaner and easier to understand.
- It doesn't matter with this particular query, but there are a few circumstances where the older outer join syntax (using +) is ambiguous and the query results are hence implementation dependent - or the query cannot be resolved at all. These do not occur with ANSI-92

- It's good practice as most developers and dba's will use ANSI-92 nowadays and you should follow the standard. Certainly all modern query tools will generate ANSI-92.
- As pointed out by @gbn, it does tend to avoid accidental cross joins

set operators are UNION, UNION ALL, EXCEPT, INTERCEPT etc. - в эту же тему.

▼ SETS

▼ UNION

Объединяет два или более набора результатов в одну итоговую выборку, исключая дубликаты.

Использование:

```
SELECT column1, column2 FROM table1
UNION
SELECT column1, column2 FROM table2;
```

Требования:

- Количество столбцов и их типы данных в обоих запросах должны совпадать.

Особенности:

- Удаляет повторяющиеся строки в результате запроса.
- Порядок столбцов должен быть одинаковым в обеих частях запроса.
- По умолчанию сортирует результат по возрастанию (как будто `ORDER BY` по всем столбцам).

Минусы: Может снижать производительность из-за необходимости удалять дубликаты и выполнять сортировку.

▼ UNION ALL

Работает аналогично оператору `UNION`, но не удаляет дубликаты, что повышает производительность.

Использование:

```
SELECT column1, column2 FROM table1
UNION ALL
SELECT column1, column2 FROM table2;
```

Требования:

- Также требует одинаковое количество столбцов и соответствие типов данных в запросах.

Особенности:

- Возвращает все строки, включая дубли.
- Быстрее, чем `UNION`, так как не требуется проверка на уникальность.

Когда использовать:

- Если не требуется удалять дубликаты и важна производительность.

▼ **EXCEPT**

Возвращает уникальные строки, которые присутствуют в первом наборе данных, но отсутствуют во втором.

- **Использование**:

```
SELECT column1, column2 FROM table1
EXCEPT
SELECT column1, column2 FROM table2;
```

Требования:

- Оба набора данных должны иметь одинаковое количество столбцов с одинаковыми типами данных.

Особенности:

- Удаляет дубликаты перед сравнением.

- Порядок столбцов должен совпадать.

Когда использовать:

- Для сравнения наборов данных, когда нужно найти записи, которые уникальны для первой выборки.

Минусы: Работает медленнее, если объем данных большой, из-за необходимости удаления дубликатов и выполнения сравнения.

▼ **INTERCEPT**

Возвращает только те строки, которые присутствуют в обоих наборах данных.

Использование:

```
SELECT column1, column2 FROM table1
INTERSECT
SELECT column1, column2 FROM table2;
```

Требования:

- Оба запроса должны возвращать одинаковое количество столбцов с одинаковыми типами данных.

Особенности:

- Удаляет дубликаты из обоих наборов данных перед сравнением.
- Порядок столбцов должен быть одинаковым.

Когда использовать:

- Для получения пересечения данных из двух наборов.

Минусы: Как и `EXCEPT`, `INTERSECT` может **медленно работать на больших объемах данных** из-за операции удаления дубликатов.

Рекомендации:

- **UNION** и **UNION ALL** используются для объединения результатов, но если дубликаты не критичны и важна производительность, выбирайте **UNION ALL**.
- **EXCEPT** полезен для нахождения различий в данных (например, для поиска строк, которых не хватает в одной таблице).

- **INTERSECT** идеально подходит для нахождения общих элементов в двух таблицах или наборах данных.
- Вообще при решении задач **лучше использовать доп. условия, нежели sets**, потому что для добавления/удаления обе таблицы сортируются (чтобы понять, что удалить). Это затратно и первфоманс не очень :c