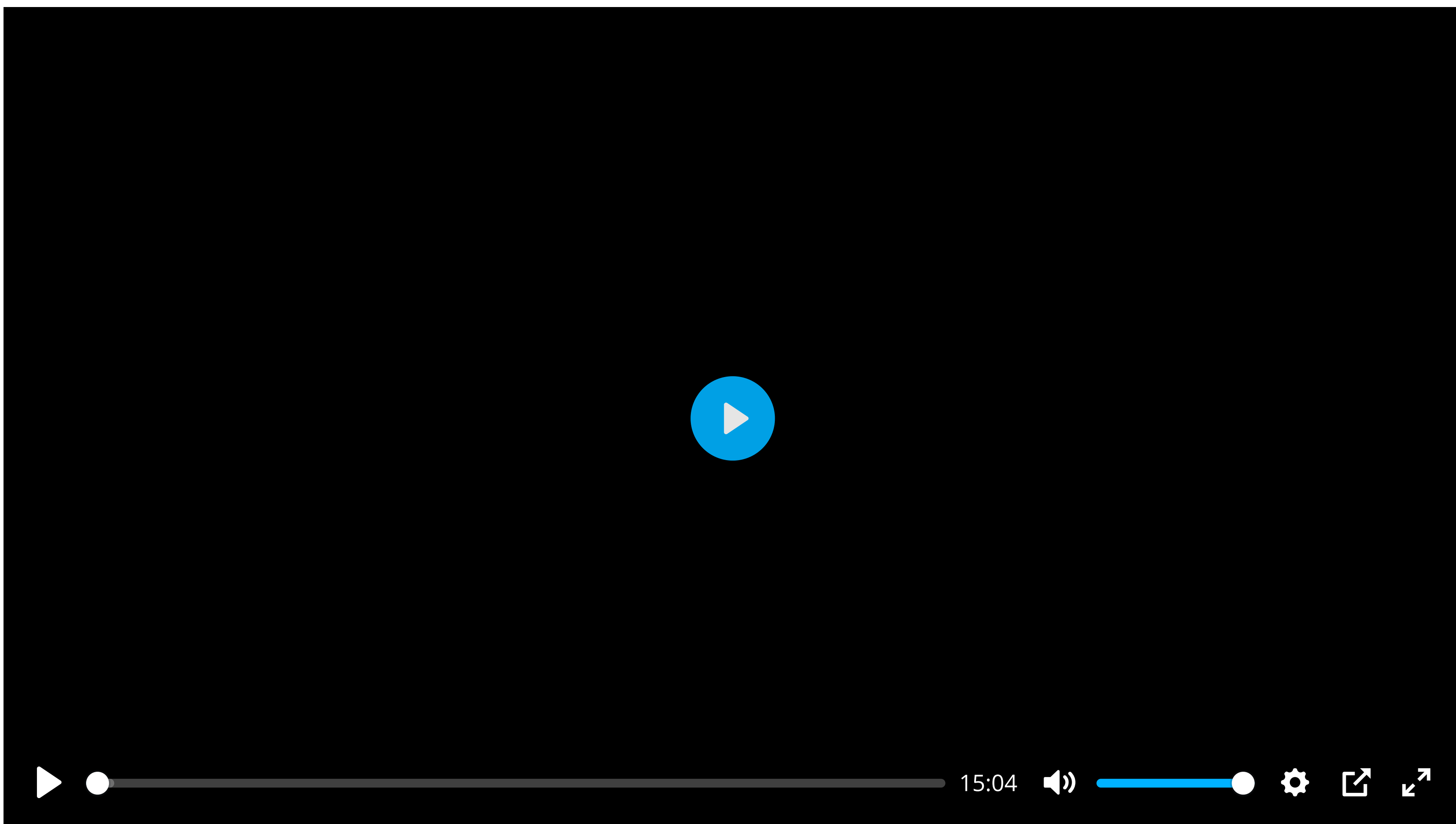


Текстовая расшифровка видео:

ОБЪЕДИНЕНИЯ

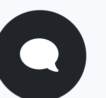
- Связанные сущности;
- Теория множеств;
- Join с условием;
- Nested Loop Join;
- Nested Loop Join – Cartesian Product;
- Hash Join;
- Merge Join;
- Join Elimination;
- Спойлеры – Data Vault и Anchor Modeling;
- Dbtvault.

Связанные сущности

Основные положения:

- В случае OLTP почти всегда много таблиц, связанных между собой.
- В случае OLAP бывают разные варианты.
- Самый простой способ – использовать **LEFT, RIGHT** и **INNER JOIN**.
- Также для особых случаев есть **FULL OUTER JOIN, CROSS JOIN** и еще несколько типов.

Вопрос: если можно JOIN'ить по произвольной колонке, то зачем нужны foreign keys?



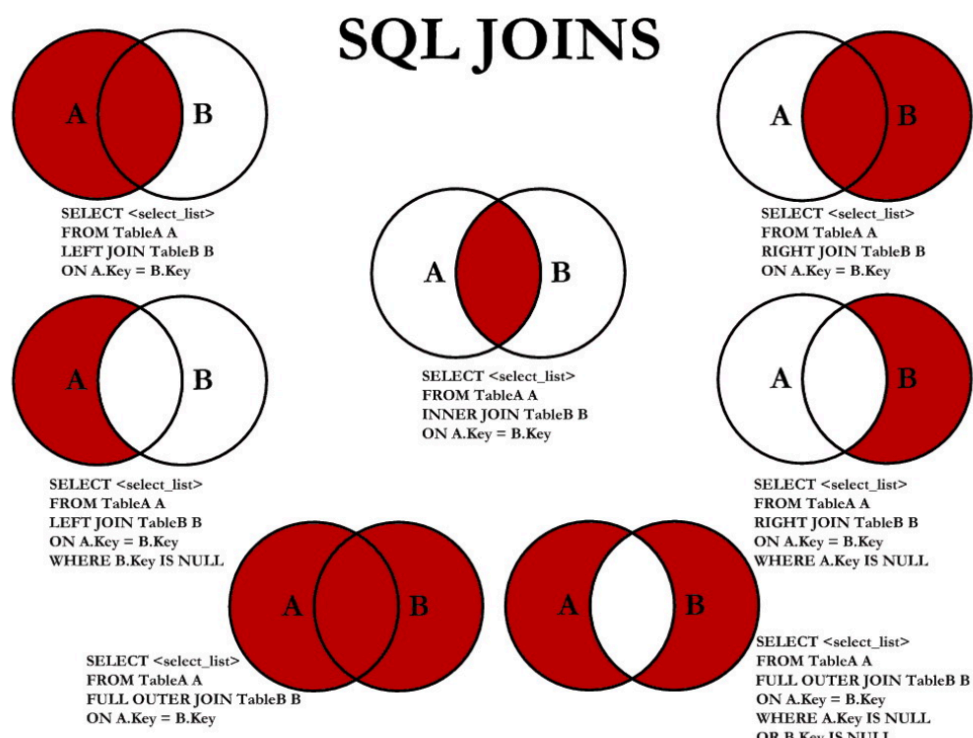
Ответ: **внешние ключи** – это механизм ограничения, который проверяет, если мы вставляем данные в таблицу, то айдишник внешней сущности, который мы вставляем, также есть в связанной таблице. База не даст вставить то, что не имеет правильной связи.

Виды взаимосвязей:

- One-to-One;
- One-to-Many;
- Many-to-Many.

Теория множеств

Часто описывают варианты объединений в контексте теории множеств:



Диаграммы Венна не нужны?

Join с условием

Ранее, мы уже видели, когда записи объединяются не по равным значениям параметров, а по условию, где один параметр больше другого.

Вопрос: что происходит, когда мы делаем join между значениями, не указывая колонку объединения?

Ответ: произойдет natural join. Колонки будут браться на основе общих имен. Скорее всего индексов по этим колонкам не будет. В такой ситуации база выберет **Nested Loop Join**.

Nested Loop Join

Nested Loop Join – это разновидность алгоритма соединения. Это то, что первым приходит в голову при вопросе: «как сджойнить два набора данных по какому-то набору полей?».

Основные положения:

- Буквально два цикла – внешний (outer) и внутренний (inner)
- Используется примерно всегда, когда мы к одной записи слева приделываем много записей справа
- Лучше всего работает, когда один из наборов данных небольшой, а второй – большой
- В идеале условие должно попадать в индекс.

Nested Loop Join – Cartesian Product

Пример, когда мы обязаны использовать Nested Loop Join:

```
EXPLAIN SELECT *
FROM aircrafts_data a1
  CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;

                    QUERY PLAN
-----------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1                      outer set
       (cost=0.00..1.09 rows=9 width=72)
   -> Materialize  (cost=0.00..1.14 rows=5 width=72)     inner set
       -> Seq Scan on aircrafts_data a2
           (cost=0.00..1.11 rows=5 width=72)
           Filter: (range > 5000)
(7 rows)
```

Представьте себе cross join (это cartigen product). Это декартово произведение, когда объединяются два датасета, и мы составляем все пары значений.

Ссылка на почитать

Hash Join

Базовая оптимизация, которую мы можем применить, – **Hash Join**.

Мы можем пройтись по одной табличке и составить в памяти hash-таблицу, то есть пару ключ-значение, где ключом будет выступать то самое поле, по которому мы делаем join.

Основные положения:

- Тоже два цикла – внешний (outer) и внутренний (inner).
- Сначала строим хэш-таблицу по внутреннему циклу, а потом во внешнем проверяем совпадения.
- Выбирается, когда данных с обеих сторон относительно много.
- Существующие индексы никак не помогают.
- Хэш-таблица должна влезать в work_mem.

Рассмотрим пример запроса:

```
EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;

               QUERY PLAN
-------------------------------------------
 Hash Join
   Hash Cond: (tf.ticket_no = t.ticket_no)
   -> Seq Scan on ticket_flights tf
   -> Hash
       -> Seq Scan on tickets t (5 rows)
```

Мы можем сделать опрос join'ом. Получится следующее:

- Seq Scan'ом проходимся по одной табличке и составляем hash-таблицу;
- Еще одним циклом, делая look up в hash, проходимся по второй.

Merge Join

Вопрос: как «жить» в хранилище, где хранится гигантское количество таблиц, большое количество join'ов?

Ответ: мы уже говорили, что можно получить большой профит от данных, физически отсортированных на диске (либо данные, либо упорядоченный индекс по этим данным). Если данные заранее отсортированы по ключу, по которому происходит join, вся операция может свестись к линейной. Мы будем проводить операцию merge, как в случае merge sort.

Кратко:

- Объединяем два отсортированных списка;
- Выбирается, когда есть индексы по полям JOIN'а в обеих табличках;
- Идеальный вариант даже для очень больших таблиц.

```
EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;

                          QUERY PLAN
----------------------------------------------------------------
 Merge Join
   Merge Cond: (t.ticket_no = tf.ticket_no)
   -> Index Scan using tickets_pkey on tickets t
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
(4 rows)
```

Ссылка на почитать

Join Elimination

Когда у нас много разных таблиц, связанных между собой через другие, можно представить ситуацию, что в объединении участвуют таблицы, из которых мы самостоятельно данные не читаем. Они выступают в роли промежуточных таблиц, объединяющих другие.

Подобные таблицы БД может исключить из запроса, чтобы данные физически не поднимались. Таким образом уменьшается количество join'ов, и сам запрос выполняется быстрее.

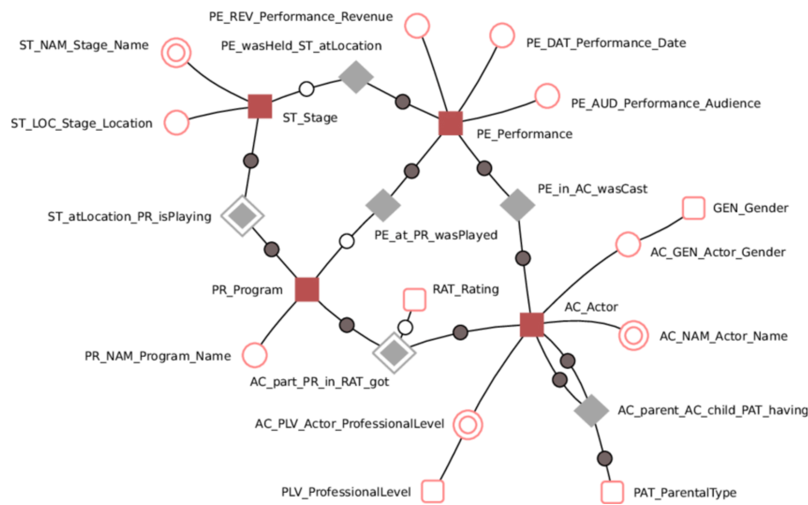
Данный подход называется «**Join Elimination**». Многие современные базы его поддерживают, включая PostgreSQL, Greenplum, MySQL и др.

```
SELECT c.name, COUNT(*)
FROM actor a
JOIN film_actor fa USING (actor_id)
JOIN film f USING (film_id)
JOIN film_category fc USING (film_id)
JOIN category c USING (category_id)
WHERE actor_id = 1
GROUP BY c.name
ORDER BY COUNT(*) DESC

SELECT c.name, COUNT(*)
FROM film_actor fa
JOIN film_category fc USING (film_id)
JOIN category c USING (category_id)
WHERE actor_id = 1
GROUP BY c.name
ORDER BY COUNT(*) DESC
```

Глубокий разбор

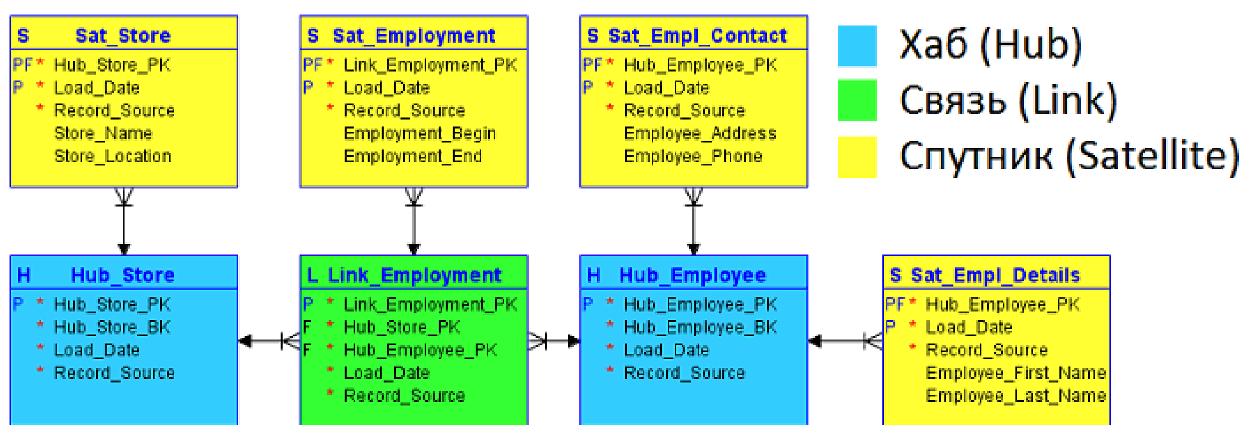
Спойлеры – Data Vault и Anchor Modeling



Если мы хотим создать большое распределенное хранилище с большим количеством разных сущностей, то берем методологию и рубим на таблички. Многие скажут, что на join'ах можно потерять всю производительность.

Если во всех табличках есть суррогатные ключи, и мы делаем join, оптимизатор может применить **merge join** (если данные физически отсортированы).

Чтобы удостовериться, что данные физически отсортированы, можно применить кластерный индекс.



Dbtvault

Dbtvault – это инструмент, который позволяет на основе YAML генерировать SQL-запросы.

Для него есть плагин «**AutomateDV**». Его основная задача: создавать в БД структуры на основе методологии проектирования Data Vault

https://automate-dv.readthedocs.io/en/latest/

Как работает:
Внутри находится большое количество макросов, написанных на Jinja Template. Раскладывание данных из плоских таблиц в структуру из разных табличек, связанных между собой линками (промежуточными таблицами), осуществляется с помощью CTE.

Как вам урок?

Изучил, далее

Слёрм ©

+7 (495) 248-05-80

Лицензия №ДЛ-1368 от 22.08.2019

Политика конфиденциальности

Публичная оферта