

Разуваев Владислав Валерьевич, 8-924-353-63-19,
razuvaevvladislav060@gmail.com

Задача:

Описать реализацию Hash Join в PostgreSQL

```
postgres=# create table t1 as select gen a, gen % 100 b from generate_series(1, 10000) gen;
SELECT 10000
postgres=# create table t2 as select gen a, gen % 100 b from generate_series(1, 10000) gen;
SELECT 10000
postgres=# analyze t1;
ANALYZE
postgres=# analyze t2;
ANALYZE
postgres=# explain (analyze, costs off) select t1.*, t2.* from t1 join t2 using(a);
          QUERY PLAN
-----
Hash Join (actual time=2.217..5.807 rows=10000 loops=1)
  Hash Cond: (t1.a = t2.a)
    -> Seq Scan on t1 (actual time=0.062..0.798 rows=10000 loops=1)
    -> Hash (actual time=2.102..2.103 rows=10000 loops=1)
      Buckets: 16384  Batches: 1  Memory Usage: 519kB
      -> Seq Scan on t2 (actual time=0.020..0.818 rows=10000 loops=1)
Planning Time: 1.056 ms
Execution Time: 6.245 ms
(8 rows)
```

1. Описать алгоритм соединения таблиц, соответствующий приведенному выше плану запроса;
2. Описать существующую реализацию hash-таблицы, используемой при соединении;
3. Описать, как обрабатываются значения Null;
4. Описать логику обработки ситуации, когда hash-таблица не помещается в память доступную процессу;

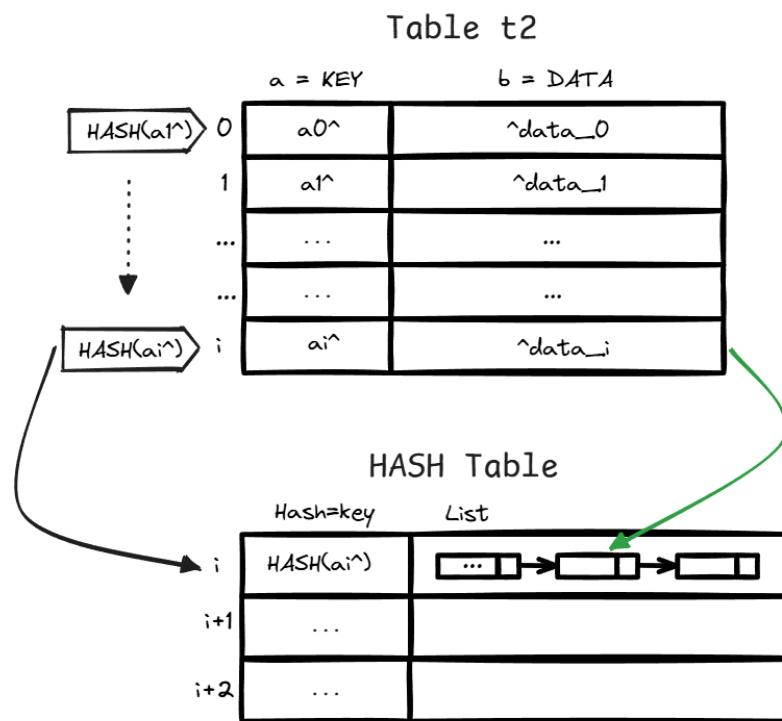
Описание необходимо сопроводить ссылками на код на зеркале репозитория на GitHub (<https://github.com/postgres/postgres>). За основу взять тег, соответствующий версии 15.2.

Table t1		Table t2		
a = KEY	b = DATA	a = KEY	b = DATA	
0	a0	data_0	a0^	^data_0
1	a1	data_1	a1^	^data_1
...
...
i	ai	data_i	ai^	^data_i

1. Описание алгоритма объединения таблиц в примере

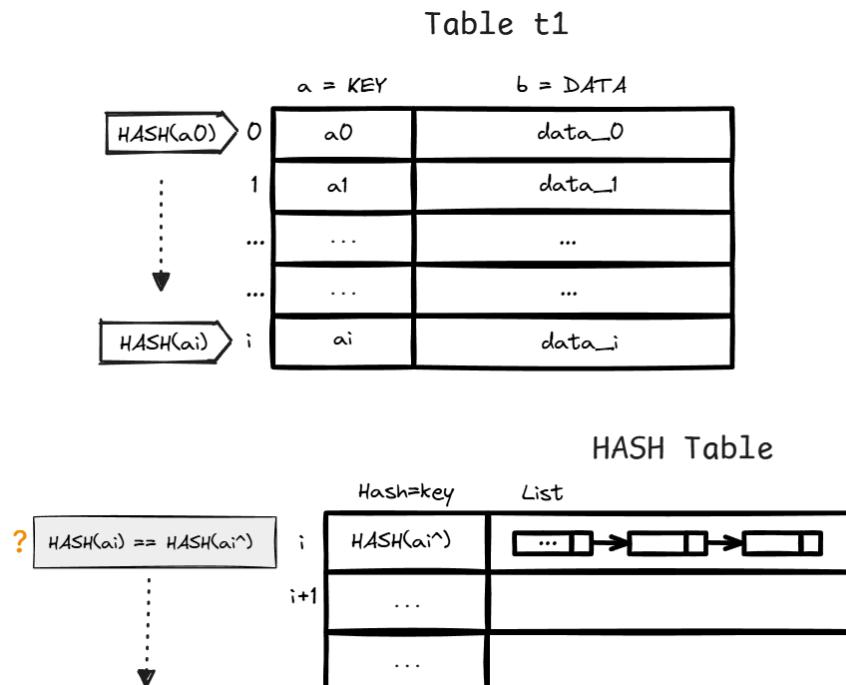
1. PostgreSQL выбирает таблицу **t2** и полностью читает её (*build input*).
Другая таблица (t1) идёт как *probe input*.
2. Для каждой строки **t2** вычисляется хеш по колонке **a**.
3. Результат хэш-функции определяет "бакет" (ячейку в хэш-таблице), куда попадёт строка.

4. Строки группируются в структуру, похожую на массив списков (bucket array).



5. Сканирование t1:

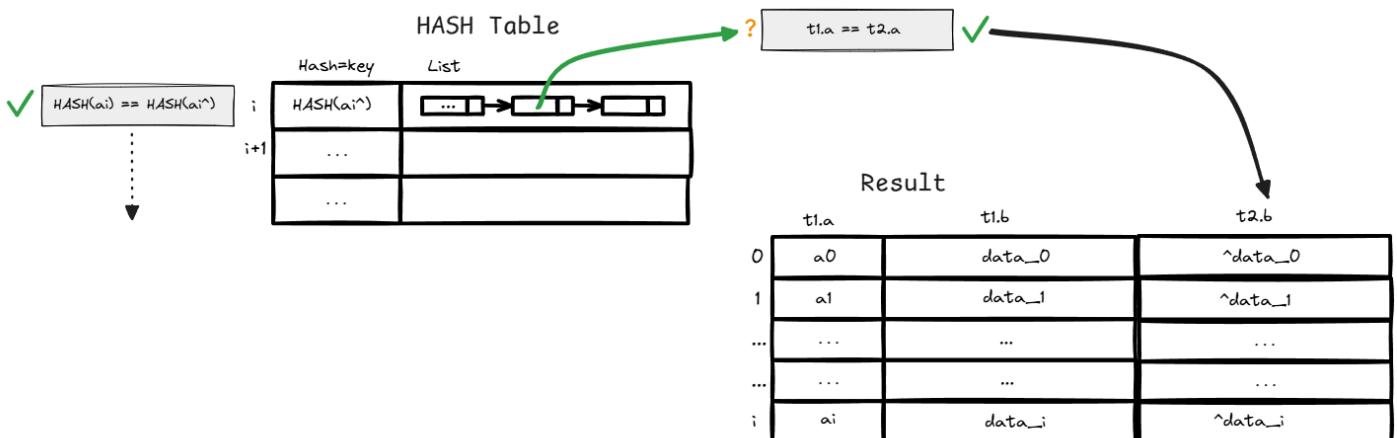
- PostgreSQL начинает построчно читать **t1**.
- Для каждой строки вычисляется хеш по тому же ключу **a**.
- По значению хеша определяется соответствующий бакет в уже построенной таблице.



6. Сравнение строк:

- Из бакета берутся кандидаты.

- Проверяется условие соединения ($t1.a = t2.a$).
- Совпадшие строки объединяются и формируют результат.



Функции [ExecHashJoin\(\)](#), [ExecInitHashJoin\(\)](#) инициализируют и выполняют хеш-соединение (Создание хэш-таблицы, выделение памяти, настройка параметров, сканирование внешней таблицы построчно и др.).

2. Реализация hash-таблицы, используемой в объединении

Hash-таблица реализована из набора ячеек (бакетов), в которых строки соединяемой таблицы (build input) хранятся в виде связных списков. buckets это массив, в каждом элементе которого хранится указатель на начало связного списка, и все строки с одинаковым хешем (остаток от деления хеш-значения на количество бакетов в таблице) попадают в один список.

Хеш-функция реализована в [ExecHashGetValue\(\)](#)

Основные структуры это [HashJoinTableData](#) [1]: хранит все параметры хеш-таблицы, а именно количество бакетов, массив указателей на первый элемент каждого списка, текущий батч, учет занимаемой памяти, ссылки на хеш-функции и функции сравнения. [HashJoinTupleData](#): представляет одну строку из build-таблицы внутри хеш-таблицы, а именно сохраняет хеш-значение, сохраняет данные строки и указатель на следующий элемент в ячейке-бакете.

Основные функции это [ExecHashTableCreate\(\)](#): выделяет память под структуру [1], расчитывает количество бакетов и инициализирует массив; это [ExecHashTableInsert\(\)](#): Для каждой строки вычисляет хеш-значение

(через [ExecHashGetValue\(\)](#)), определяет бакет и добавляет элемент в связанный список; это [ExecScanHashBucket\(\)](#): при обработке probe-строк из второй таблицы ищет совпадения (вычисляет хеш -> находит правильный бакет -> перебирает элементы списка, сравнивая ключи).

3. Обработка NULL

На этапе построения хеш-таблицы, перед вычислением хеша проводится проверка взодных данных (ключа) на значение NULL. Если ключ NULL, строка либо не попадает в хеш таблицу, либо сохраняется отдельно (в зависимости от типа join). То есть строка с ключом NULL не используется в хеш-таблице.

На этапе поиска совпадений, при значении ключа во второй таблице (probe input) равным NULL, PostgreSQL пропускает строку, так как совпадений точно не будет. Это делается в [ExecHashGetValue\(\)](#), которая возвращает флаг можно/нельзя хешировать.

При INNER JOIN строки с ключом NULL полностью отбрасываются, при LEFT JOIN строка с ключом NULL (из probe input) не найдет совпадений, но будет выдана в результат, где колонки из второй таблицы также будут NULL; при RIGHT JOIN строки с NULL могут попасть в результат только из таблицы, которая не нашла пару. [ExecHashJoinOuterGetTuple\(\)](#) — получение строк из probe input. [ExecQual\(\)](#) — проверка условия соединения, в том числе корректная обработка NULL по SQL-логике.

4. Логика обработки ситуации нехватки памяти для хеш-таблицы

При выполнении Hash Join вся таблица t2 загружается в память. Если достигается лимит выделенной памяти, то всю структуру нельзя сохранять в оперативке. Когда размер хеш-таблицы начинает превышать лимит `work_mem`, PostgreSQL вычисляет необходимое количество частей. Каждая строка по значению своего хеша распределяется в один из батчей. Текущий активный батч хранится в памяти, а все остальные выгружаются во временные файлы на диск (используются `BufFile`, `LogicalTape` для работы с временными файлами). Далее PostgreSQL сначала обрабатывает batch 0:

строит по нему хеш-таблицу и сканирует вторую таблицу (probe input), чтобы найти совпадения. Когда batch 0 закончен, память освобождается, загружается следующий и все повторяется, но только с теми строками из probe input, относящимися к текущему batch. [ExecHashIncreaseNumBatches\(\)](#) перерасчёты количества батчей, [ExecHashJoinSaveTuple\(\)](#) сохранение строк в файлы на диск, [ExecHashJoinGetSavedTuple\(\)](#) восстановление строк при обработке батча.