Explaining the Postgres Query Optimizer

Bruce Momjian

The optimizer is the "brain" of the database, interpreting SQL queries and determining the fastest method of execution. This talk uses the EXPLAIN command to show how the optimizer interprets queries and determines optimal execution.

Creative Commons Attribution License  http://momjian.us/presentations

Last updated: September, 2018
Postgres Query Execution

- User Terminal
- Application Code
- Libpq
- PostgreSQL Database Server

Queries

Results
Postgres Query Execution

- **Main**
- **Postmaster**
- **Postgres**
- **Libpq**

**Steps:**
1. **Parse Statement**
2. **Traffic Cop**
   - **Query**
   - **Rewrite Query**
3. **Generate Paths**
   - **Optimal Path**
4. **Generate Plan**
5. **Execute Plan**

**Related Components:**
- **Utilities**
- **Catalog**
- **Storage Managers**
- **Access Methods**
- **Nodes / Lists**

**Example Queries:**
- CREATE TABLE, COPY
- SELECT, INSERT, UPDATE, DELETE

**Utility Commands:**
- CREATE TABLE
- COPY

**Access Methods:**
- Catalog
- Storage Managers
- Utilities
- Nodes / Lists
Parse Statement

Traffic Cop

Rewrite Query

Generate Paths

Optimal Path

Generate Plan

Execute Plan

Utility Command
e.g. CREATE TABLE, COPY

Query
SELECT, INSERT, UPDATE, DELETE
The Optimizer Is the Brain

https://www.flickr.com/photos/dierkschaefer/
What Decisions Does the Optimizer Have to Make?

- Scan Method
- Join Method
- Join Order
Which Scan Method?

- Sequential Scan
- Bitmap Index Scan
- Index Scan
A Simple Example Using `pg_class.relname`

```sql
SELECT relname
FROM pg_class
ORDER BY 1
LIMIT 8;
```

<table>
<thead>
<tr>
<th>relname</th>
</tr>
</thead>
<tbody>
<tr>
<td>_pg_foreign_data_wrappers</td>
</tr>
<tr>
<td>_pg_foreign_servers</td>
</tr>
<tr>
<td>_pg_user_mappings</td>
</tr>
<tr>
<td>administrable_role_authorizations</td>
</tr>
<tr>
<td>applicable_roles</td>
</tr>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>check_constraint_routine_usage</td>
</tr>
<tr>
<td>check_constraints</td>
</tr>
</tbody>
</table>
Let’s Use Just the First Letter of `pg_class.relname`

```
SELECT substring(relname, 1, 1)
FROM pg_class
ORDER BY 1
LIMIT 8;
```
CREATE TEMPORARY TABLE sample (letter, junk) AS
    SELECT substring(relname, 1, 1), repeat('x', 250)
    FROM pg_class
    ORDER BY random(); -- add rows in random order

CREATE INDEX i_sample on sample (letter);

All queries used in this presentation are available at http://momjian.us/main/writings/pqsql/optimizer.sql.
CREATE OR REPLACE FUNCTION lookup_letter(text) RETURNS SETOF text AS $$
BEGIN
RETURN QUERY EXECUTE 'EXPLAIN SELECT letter FROM sample WHERE letter = ''' || $1 || '''';
END
$$ LANGUAGE plpgsql;
What is the Distribution of the *sample* Table?

```sql
WITH letters (letter, count) AS (  
    SELECT letter, COUNT(*)  
    FROM sample  
    GROUP BY 1
)

SELECT letter, count, (count * 100.0 / (SUM(count) OVER ()))::numeric(4,1) AS "%"  
FROM letters  
ORDER BY 2 DESC;
```
What is the Distribution of the *sample* Table?

| letter | count | %  |
|--------+-------+----|
| p      | 199   | 78.7|
| s      | 9     | 3.6 |
| c      | 8     | 3.2 |
| r      | 7     | 2.8 |
| t      | 5     | 2.0 |
| v      | 4     | 1.6 |
| f      | 4     | 1.6 |
| d      | 4     | 1.6 |
| u      | 3     | 1.2 |
| a      | 3     | 1.2 |
| _      | 3     | 1.2 |
| e      | 2     | 0.8 |
| i      | 1     | 0.4 |
| k      | 1     | 0.4 |
Is the Distribution Important?

```
EXPLAIN SELECT letter
FROM sample
WHERE letter = 'p';
```

**QUERY PLAN**

```
Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=32)
  Index Cond: (letter = 'p'::text)
```
Is the Distribution Important?

```sql
EXPLAIN SELECT letter
FROM sample
WHERE letter = 'd';
```

**QUERY PLAN**

```
Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=32)
  Index Cond: (letter = 'd'::text)
```
Is the Distribution Important?

EXPLAIN SELECT letter
FROM sample
WHERE letter = 'k';

QUERY PLAN

Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=32)
  Index Cond: (letter = 'k'::text)
Running ANALYZE Causes a Sequential Scan for a Common Value

```
ANALYZE sample;

EXPLAIN SELECT letter
FROM sample
WHERE letter = 'p';
```

**QUERY PLAN**

```
Seq Scan on sample (cost=0.00..13.16 rows=199 width=2)
  Filter: (letter = 'p'::text)
```

Autovacuum cannot ANALYZE (or VACUUM) temporary tables because these tables are only visible to the creating session.
Sequential Scan

Heap

D  D  D  D  D  D  D  D  D  D  D  D  D  D  D
A  A  A  A  A  A  A  A  A  A  A  A  A  A  A
T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
A  A  A  A  A  A  A  A  A  A  A  A  A  A  A

8K
A Less Common Value Causes a Bitmap Index Scan

EXPLAIN SELECT letter
FROM sample
WHERE letter = 'd';

QUERY PLAN

---
Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)
  Recheck Cond: (letter = 'd'::text)
  -> Bitmap Index Scan on i_sample (cost=0.00..4.28 rows=4 width=0)
  Index Cond: (letter = 'd'::text)
---
Bitmap Index Scan

Index 1  Index 2  Combined  Index

col1 = 'A'  col2 = 'NS'

Table

'A' AND 'NS'
EXPLAIN SELECT letter
FROM sample
WHERE letter = 'k';

QUERY PLAN

Index Scan using i_sample on sample  (cost=0.00..8.27 rows=1 width=2)
  Index Cond: (letter = 'k':::text)
Index Scan

Index

< Key = >

Heap

DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA
WITH letter (letter, count) AS (  
    SELECT letter, COUNT(*)  
    FROM sample  
    GROUP BY 1  
)  
SELECT letter AS l, count, lookup_letter(letter)  
FROM letter  
ORDER BY 2 DESC;

| l | count | lookup_letter |
|---+-------+---------------|
| p | 199   | Seq Scan on sample (cost=0.00..13.16 rows=199 width=2)  
| p | 199   | Filter: (letter = 'p '::text) |
| s | 9     | Seq Scan on sample (cost=0.00..13.16 rows=9 width=2)  
| s | 9     | Filter: (letter = 's '::text) |
| c | 8     | Seq Scan on sample (cost=0.00..13.16 rows=8 width=2)  
| c | 8     | Filter: (letter = 'c '::text) |
| r | 7     | Seq Scan on sample (cost=0.00..13.16 rows=7 width=2)  
| r | 7     | Filter: (letter = 'r '::text) |
WITH letter (letter, count) AS ( 
    SELECT letter, COUNT(*)
    FROM sample
    GROUP BY 1
)
SELECT letter AS l, count,
    (SELECT *
    FROM lookup_letter(letter) AS l2
    LIMIT 1) AS lookup_letter
FROM letter
ORDER BY 2 DESC;
<table>
<thead>
<tr>
<th>lookup_letter</th>
<th>count</th>
<th>operation</th>
<th>cost</th>
<th>rows</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>199</td>
<td>Seq Scan on sample</td>
<td>0.00..13.16</td>
<td>199</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>9</td>
<td>Seq Scan on sample</td>
<td>0.00..13.16</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>Seq Scan on sample</td>
<td>0.00..13.16</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>Seq Scan on sample</td>
<td>0.00..13.16</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>t</td>
<td>5</td>
<td>Bitmap Heap Scan on sample</td>
<td>4.29..12.76</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>Bitmap Heap Scan on sample</td>
<td>4.28..12.74</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>Bitmap Heap Scan on sample</td>
<td>4.28..12.74</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>Bitmap Heap Scan on sample</td>
<td>4.28..12.74</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>Bitmap Heap Scan on sample</td>
<td>4.27..11.38</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>_</td>
<td>3</td>
<td>Bitmap Heap Scan on sample</td>
<td>4.27..11.38</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>u</td>
<td>3</td>
<td>Bitmap Heap Scan on sample</td>
<td>4.27..11.38</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td>Index Scan using i_sample on sample</td>
<td>0.00..8.27</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>Index Scan using i_sample on sample</td>
<td>0.00..8.27</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>k</td>
<td>1</td>
<td>Index Scan using i_sample on sample</td>
<td>0.00..8.27</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
We Can Force an Index Scan

SET enable_seqscan = false;

SET enable_bitmapscan = false;

WITH letter (letter, count) AS (  
    SELECT letter, COUNT(*)  
    FROM sample  
    GROUP BY 1
  )  
SELECT letter AS l, count,  
  (SELECT *  
   FROM lookup_letter(letter) AS l2  
   LIMIT 1) AS lookup_letter  
FROM letter  
ORDER BY 2 DESC;
Notice the High Cost for Common Values

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>lookup_letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>199</td>
<td>Index Scan using i_sample on sample (cost=0.00..39.33 rows=199 width=2)</td>
</tr>
<tr>
<td>s</td>
<td>9</td>
<td>Index Scan using i_sample on sample (cost=0.00..22.14 rows=9 width=2)</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>Index Scan using i_sample on sample (cost=0.00..19.84 rows=8 width=2)</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>Index Scan using i_sample on sample (cost=0.00..19.82 rows=7 width=2)</td>
</tr>
<tr>
<td>t</td>
<td>5</td>
<td>Index Scan using i_sample on sample (cost=0.00..15.21 rows=5 width=2)</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>Index Scan using i_sample on sample (cost=0.00..15.19 rows=4 width=2)</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>Index Scan using i_sample on sample (cost=0.00..15.19 rows=4 width=2)</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>Index Scan using i_sample on sample (cost=0.00..15.19 rows=4 width=2)</td>
</tr>
<tr>
<td>_</td>
<td>3</td>
<td>Index Scan using i_sample on sample (cost=0.00..12.88 rows=3 width=2)</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>Index Scan using i_sample on sample (cost=0.00..12.88 rows=3 width=2)</td>
</tr>
<tr>
<td>u</td>
<td>3</td>
<td>Index Scan using i_sample on sample (cost=0.00..12.88 rows=3 width=2)</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td>Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=2)</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=2)</td>
</tr>
<tr>
<td>k</td>
<td>1</td>
<td>Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=2)</td>
</tr>
</tbody>
</table>

RESET ALL;
This Was the Optimizer’s Preference

| l | count | lookup_letter |
|---+-------+-----------------|
| p | 199 | Seq Scan on sample (cost=0.00..13.16 rows=199 width=2) |
| s | 9 | Seq Scan on sample (cost=0.00..13.16 rows=9 width=2) |
| c | 8 | Seq Scan on sample (cost=0.00..13.16 rows=8 width=2) |
| r | 7 | Seq Scan on sample (cost=0.00..13.16 rows=7 width=2) |
| t | 5 | Bitmap Heap Scan on sample (cost=4.29..12.76 rows=5 width=2) |
| f | 4 | Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2) |
| v | 4 | Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2) |
| d | 4 | Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2) |
| a | 3 | Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2) |
| _ | 3 | Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2) |
| u | 3 | Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2) |
| e | 2 | Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=2) |
| i | 1 | Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=2) |
| k | 1 | Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=2) |
Which Join Method?

- Nested Loop
  - With Inner Sequential Scan
  - With Inner Index Scan
- Hash Join
- Merge Join
What Is in \textit{pg\_proc.oid}?

```sql
SELECT oid
FROM pg_proc
ORDER BY 1
LIMIT 8;
```

<table>
<thead>
<tr>
<th>oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>41</td>
</tr>
</tbody>
</table>
CREATE TEMPORARY TABLE sample1 (id, junk) AS
  SELECT oid, repeat('x', 250)
  FROM pg_proc
  ORDER BY random();  -- add rows in random order

CREATE TEMPORARY TABLE sample2 (id, junk) AS
  SELECT oid, repeat('x', 250)
  FROM pg_class
  ORDER BY random();  -- add rows in random order

These tables have no indexes and no optimizer statistics.
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample1.id = 33;

QUERY PLAN

---------------------------------------------------------------------
 Nested Loop   (cost=0.00..234.68 rows=300 width=32)  
   ->  Seq Scan on sample1  (cost=0.00..205.54 rows=50 width=4)  
       Filter: (id = 33::oid)  
   ->  Materialize  (cost=0.00..25.41 rows=6 width=36)  
       ->  Seq Scan on sample2  (cost=0.00..25.38 rows=6 width=36)  
           Filter: (id = 33::oid)
Nested Loop Join
with Inner Sequential Scan

No Setup Required

Used For Small Tables
Pseudocode for Nested Loop Join with Inner Sequential Scan

for (i = 0; i < length(outer); i++)
    for (j = 0; j < length(inner); j++)
        if (outer[i] == inner[j])
            output(outer[i], inner[j]);
EXPLAIN SELECT sample1.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample2.id > 33;

QUERY PLAN

<table>
<thead>
<tr>
<th>Statement</th>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Join</td>
<td>30.50 - 950.88</td>
<td>20424</td>
<td>32</td>
</tr>
<tr>
<td>Hash Cond:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq Scan on sample1</td>
<td>0.00 - 180.63</td>
<td>9963</td>
<td>36</td>
</tr>
<tr>
<td>Hash</td>
<td>25.38 - 25.38</td>
<td>410</td>
<td>4</td>
</tr>
<tr>
<td>Seq Scan on sample2</td>
<td>0.00 - 25.38</td>
<td>410</td>
<td>4</td>
</tr>
<tr>
<td>Filter:</td>
<td>id &gt; 33::oid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash Join

Must fit in Main Memory
Pseudocode for Hash Join

for (j = 0; j < length(inner); j++)
    hash_key = hash(inner[j]);
    append(hash_store[hash_key], inner[j]);
for (i = 0; i < length(outer); i++)
    hash_key = hash(outer[i]);
    for (j = 0; j < length(hash_store[hash_key]); j++)
        if (outer[i] == hash_store[hash_key][j])
            output(outer[i], inner[j]);
EXPLAIN SELECT sample1.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id);

QUERY PLAN

Merge Join  (cost=927.72..1852.95 rows=61272 width=32)
  Merge Cond: (sample2.id = sample1.id)
  ->  Sort  (cost=85.43..88.50 rows=1230 width=4)
      Sort Key: sample2.id
      ->  Seq Scan on sample2  (cost=0.00..22.30 rows=1230 width=4)
  ->  Sort  (cost=842.29..867.20 rows=9963 width=36)
      Sort Key: sample1.id
      ->  Seq Scan on sample1  (cost=0.00..180.63 rows=9963 width=36)
Merge Join

Ideal for Large Tables
An Index Can Be Used to Eliminate the Sort
Pseudocode for Merge Join

sort(outer);
sort(inner);
i = 0;
j = 0;
save_j = 0;
while (i < length(outer))
    if (outer[i] == inner[j])
        output(outer[i], inner[j]);
    if (outer[i] <= inner[j] && j < length(inner))
        j++;
    else
        save_j = j;
else
    i++;
    j = save_j;
EXPLAIN SELECT sample2.junk
FROM sample2 JOIN sample1 ON (sample2.id = sample1.id);

QUERY PLAN

Merge Join  (cost=927.72..1852.95 rows=61272 width=32)
  Merge Cond: (sample2.id = sample1.id)
    ->  Sort  (cost=85.43..88.50 rows=1230 width=36)
       Sort Key: sample2.id
    ->  Seq Scan on sample2  (cost=0.00..22.30 rows=1230 width=36)
    ->  Sort  (cost=842.29..867.20 rows=9963 width=4)
       Sort Key: sample1.id
    ->  Seq Scan on sample1  (cost=0.00..180.63 rows=9963 width=4)

The most restrictive relation, e.g., `sample2`, is always on the outer side of merge joins. All previous merge joins also had `sample2` in outer position.
Add Optimizer Statistics

ANALYZE sample1;

ANALYZE sample2;
This Was a Merge Join without Optimizer Statistics

```
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id);
```

**QUERY PLAN**

<table>
<thead>
<tr>
<th>Plan Type</th>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Join</td>
<td>15.85</td>
<td>260</td>
<td>254</td>
</tr>
<tr>
<td>Hash Cond: (sample1.id = sample2.id)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq Scan on sample1</td>
<td>0.00</td>
<td>2256</td>
<td>4</td>
</tr>
<tr>
<td>Hash</td>
<td>12.60</td>
<td>260</td>
<td>258</td>
</tr>
<tr>
<td>Seq Scan on sample2</td>
<td>0.00</td>
<td>260</td>
<td>258</td>
</tr>
</tbody>
</table>
Outer Joins Can Affect Optimizer Join Usage

EXPLAIN SELECT sample1.junk
FROM sample1 RIGHT OUTER JOIN sample2 ON (sample1.id = sample2.id);

QUERY PLAN

Hash Left Join (cost=131.76..148.26 rows=260 width=254)
  Hash Cond: (sample2.id = sample1.id)
  -> Seq Scan on sample2 (cost=0.00..12.60 rows=260 width=4)
  -> Hash (cost=103.56..103.56 rows=2256 width=258)
    -> Seq Scan on sample1 (cost=0.00..103.56 rows=2256 width=258)
Cross Joins Are Nested Loop Joins without Join Restriction

EXPLAIN SELECT sample1.junk
FROM sample1 CROSS JOIN sample2;

QUERY PLAN

--------------------------------------------------------------------
 Nested Loop  (cost=0.00..7448.81 rows=586560 width=254)
  ->  Seq Scan on sample1  (cost=0.00..103.56 rows=2256 width=254)
  ->  Materialize   (cost=0.00..13.90 rows=260 width=0)
    ->  Seq Scan on sample2   (cost=0.00..12.60 rows=260 width=0)
CREATE INDEX i_sample1 on sample1 (id);

CREATE INDEX i_sample2 on sample2 (id);
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample1.id = 33;

QUERY PLAN

----------------------------------------------------------------------------------
Nested Loop  (cost=0.00..16.55 rows=1 width=254)
  ->  Index Scan using i_sample1 on sample1  (cost=0.00..8.27 rows=1 width=4)
       Index Cond: (id = 33::oid)
  ->  Index Scan using i_sample2 on sample2  (cost=0.00..8.27 rows=1 width=258)
       Index Cond: (sample2.id = 33::oid)
----------------------------------------------------------------------------------
Nested Loop Join with Inner Index Scan

No Setup Required

Index Must Already Exist
for (i = 0; i < length(outer); i++)
    index_entry = get_first_match(outer[j])
    while (index_entry)
        output(outer[i], inner[index_entry]);
        index_entry = get_next_match(index_entry);
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample2.junk ~ '^aaa';

QUERY PLAN

-------------------------------------------------------------------------------
| Nested Loop   | (cost=0.00..21.53 rows=1 width=254) |
|               | -> Seq Scan on sample2  (cost=0.00..13.25 rows=1 width=258) |
|               |   Filter: (junk ~ '^aaa'::text) |
|               | -> Index Scan using i_sample1 on sample1  (cost=0.00..8.27 rows=1 width=4) |
|               |   Index Cond: (sample1.id = sample2.id) |

No junk rows begin with 'aaa'.
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample2.junk ~ '^xxx';

QUERY PLAN

Hash Join  (cost=16.50..131.12 rows=260 width=254)
  Hash Cond: (sample1.id = sample2.id)
  ->  Seq Scan on sample1  (cost=0.00..103.56 rows=2256 width=4)
  ->  Hash  (cost=13.25..13.25 rows=260 width=258)
     ->  Seq Scan on sample2  (cost=0.00..13.25 rows=260 width=258)
        Filter: (junk ~ '^xxx'::text)

Hash join was chosen because many more rows are expected. The smaller table, e.g., sample2, is always hashed.
Without LIMIT, Hash Is Used for this Unrestricted Join

```
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id);

QUERY PLAN

Hash Join  (cost=15.85..130.47 rows=260 width=254)
   Hash Cond: (sample1.id = sample2.id)
   ->  Seq Scan on sample1  (cost=0.00..103.56 rows=2256 width=4)
   ->  Hash   (cost=12.60..12.60 rows=260 width=258)
       ->  Seq Scan on sample2  (cost=0.00..12.60 rows=260 width=258)
```
EXPLAIN SELECT sample2.id, sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
ORDER BY 1
LIMIT 1;

QUERY PLAN

Limit (cost=0.00..1.83 rows=1 width=258)
  -> Nested Loop (cost=0.00..477.02 rows=260 width=258)
    -> Index Scan using i_sample2 on sample2 (cost=0.00..52.15 rows=260 width=258)
    -> Index Scan using i_sample1 on sample1 (cost=0.00..1.62 rows=1 width=4)
      Index Cond: (sample1.id = sample2.id)
```sql
EXPLAIN SELECT sample2.id, sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
ORDER BY 1
LIMIT 10;
```

**QUERY PLAN**

```
Limit (cost=0.00..18.35 rows=10 width=258)
  -> Nested Loop (cost=0.00..477.02 rows=260 width=258)
    -> Index Scan using i_sample2 on sample2 (cost=0.00..52.15 rows=260 width=258)
    -> Index Scan using i_sample1 on sample1 (cost=0.00..1.62 rows=1 width=4)
      Index Cond: (sample1.id = sample2.id)
```
LIMIT 100 Switches to Hash Join

EXPLAIN SELECT sample2.id, sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
ORDER BY 1
LIMIT 100;

QUERY PLAN

Limit (cost=140.41..140.66 rows=100 width=258)
  -> Sort (cost=140.41..141.06 rows=260 width=258)
      Sort Key: sample2.id
      -> Hash Join (cost=15.85..130.47 rows=260 width=258)
          Hash Cond: (sample1.id = sample2.id)
          -> Seq Scan on sample1 (cost=0.00..103.56 rows=2256 width=4)
          -> Hash (cost=12.60..12.60 rows=260 width=258)
              -> Seq Scan on sample2 (cost=0.00..12.60 rows=260 width=258)
Conclusion

http://momjian.us/presentations

https://www.flickr.com/photos/trevorklatko/