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: February, 2019
Postgres Query Execution

User Terminal

Application Code

Libpq

Queries

Results

PostgreSQL Database Server
Postgres Query Execution

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

**Utility**

- **Parse Statement**
- **Traffic Cop**
  - **Query**
    - **SELECT, INSERT, UPDATE, DELETE**
  - **Rewrite Query**
  - **Generate Paths**
    - **Optimal Path**
  - **Generate Plan**
  - **Execute Plan**

**Command**

- **Utility**
  - **e.g. CREATE TABLE, COPY**

**Storage Managers**

- **Catalog**
- **Utilities**
- **Access Methods**
- **Nodes / Lists**
- **Storage Managers**
Postgres Query Execution

1. Parse Statement
2. Traffic Cop
3. Rewrite Query
4. Generate Paths
5. Generate Plan
6. Execute Plan

Utility

Optimal Path

SELECT, INSERT, UPDATE, DELETE

e.g. CREATE TABLE, COPY
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 \textit{pg\_class\_relname}

\begin{verbatim}
SELECT relname
FROM pg_class
ORDER BY 1
LIMIT 8;

    relname
------------------------
 _pg_foreign_data_wrappers
 _pg_foreign_servers
 _pg_user_mappings
 administrable_role_authorizations
 applicable_roles
 attributes
 check_constraint_routine_usage
 check_constraints
\end{verbatim}
Let’s Use Just the First Letter of `pg_class.relname`

```sql
SELECT substring(relname, 1, 1)
FROM pg_class
ORDER BY 1
LIMIT 8;
```

<table>
<thead>
<tr>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
</tr>
<tr>
<td>_</td>
</tr>
<tr>
<td>_</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>
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/pgsql/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?

```
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?

<table>
<thead>
<tr>
<th>letter</th>
<th>count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>199</td>
<td>78.7</td>
</tr>
<tr>
<td>s</td>
<td>9</td>
<td>3.6</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>3.2</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>2.8</td>
</tr>
<tr>
<td>t</td>
<td>5</td>
<td>2.0</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>1.6</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>1.6</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>1.6</td>
</tr>
<tr>
<td>u</td>
<td>3</td>
<td>1.2</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>1.2</td>
</tr>
<tr>
<td>_</td>
<td>3</td>
<td>1.2</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td>0.8</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>k</td>
<td>1</td>
<td>0.4</td>
</tr>
</tbody>
</table>
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)
```
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

```sql
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 A T A
D A T A
D A T A
D A T A
D A T A
D A T A
D A T A
D A T A

8K
```
EXPLAIN SELECT letter
FROM sample
WHERE letter = 'd';

---

QUERY PLAN

<table>
<thead>
<tr>
<th>Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recheck Cond: (letter = 'd'::text)</td>
</tr>
<tr>
<td>-&gt; Bitmap Index Scan on i_sample (cost=0.00..4.28 rows=4 width=0)</td>
</tr>
<tr>
<td>Index Cond: (letter = 'd'::text)</td>
</tr>
</tbody>
</table>
Bitmap Index Scan

Index 1  Index 2  Combined
col1 = 'A' col2 = 'NS'  Index

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)
Let’s Look at All Values and their Effects

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;

<table>
<thead>
<tr>
<th>l</th>
<th>count</th>
<th>lookup_letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>199</td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=199 width=2)</td>
</tr>
<tr>
<td>s</td>
<td>9</td>
<td>Filter: (letter = 'p'::text)</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=8 width=2)</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>Filter: (letter = 's'::text)</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=7 width=2)</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>Filter: (letter = 'r'::text)</td>
</tr>
</tbody>
</table>

...
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>char</th>
<th>count</th>
<th>lookup_letter</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>199</td>
<td></td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=199 width=2)</td>
</tr>
<tr>
<td>s</td>
<td>9</td>
<td></td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=9 width=2)</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td></td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=8 width=2)</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td></td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=7 width=2)</td>
</tr>
<tr>
<td>t</td>
<td>5</td>
<td></td>
<td>Bitmap Heap Scan on sample (cost=4.29..12.76 rows=5 width=2)</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td></td>
<td>Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td></td>
<td>Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td></td>
<td>Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td></td>
<td>Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2)</td>
</tr>
<tr>
<td>_</td>
<td>3</td>
<td></td>
<td>Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2)</td>
</tr>
<tr>
<td>u</td>
<td>3</td>
<td></td>
<td>Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2)</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td></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></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></td>
<td>Index Scan using i_sample on sample (cost=0.00..8.27 rows=1 width=2)</td>
</tr>
</tbody>
</table>
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

| count | lookup_letter |
|-------+---------------|
| 199   | p             |
| 9     | s             |
| 8     | c             |
| 7     | r             |
| 5     | t             |
| 4     | d             |
| 4     | v             |
| 4     | f             |
| 3     | _             |
| 3     | a             |
| 3     | u             |
| 2     | e             |
| 1     | i             |
| 1     | k             |

RESET ALL;
This Was the Optimizer’s Preference

<table>
<thead>
<tr>
<th>Letter</th>
<th>Count</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>199</td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=199 width=2)</td>
</tr>
<tr>
<td>s</td>
<td>9</td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=9 width=2)</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=8 width=2)</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>Seq Scan on sample (cost=0.00..13.16 rows=7 width=2)</td>
</tr>
<tr>
<td>t</td>
<td>5</td>
<td>Bitmap Heap Scan on sample (cost=4.29..12.76 rows=5 width=2)</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>Bitmap Heap Scan on sample (cost=4.28..12.74 rows=4 width=2)</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2)</td>
</tr>
<tr>
<td>_</td>
<td>3</td>
<td>Bitmap Heap Scan on sample (cost=4.27..11.38 rows=3 width=2)</td>
</tr>
<tr>
<td>u</td>
<td>3</td>
<td>Bitmap Heap Scan on sample (cost=4.27..11.38 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>
Which Join Method?

- Nested Loop
  - With Inner Sequential Scan
  - With Inner Index Scan
- Hash Join
- Merge Join
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>34</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>39</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.
Join the Two Tables with a Tight Restriction

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]);
Join the Two Tables with a Looser Restriction

EXPLAIN SELECT sample1.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample2.id > 33;

QUERY PLAN

---------------------------------------------------------------------
Hash Join  (cost=30.50..950.88  rows=20424  width=32)
  Hash Cond: (sample1.id = sample2.id)
  ->  Seq Scan on sample1  (cost=0.00..180.63  rows=9963  width=36)
  ->  Hash  (cost=25.38..25.38  rows=410  width=4)
      ->  Seq Scan on sample2  (cost=0.00..25.38  rows=410  width=4)
          Filter: (id > 33::oid)
Hash Join

Outer

aay
aag
aak
aar

Inner

aak → aas
aam → aay → aar
ao → aw

Hashed

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]);
Join the Two Tables with No Restriction

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) |

38 / 56
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++;
    if (outer[i] < inner[j])
        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;
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)
Outer Joins Can Affect Optimizer Join Usage

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

 QUERY PLAN

<table>
<thead>
<tr>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>131.76..148.26</td>
<td>260</td>
<td>254</td>
</tr>
</tbody>
</table>

Hash Left Join (cost=131.76..148.26 rows=260 width=254)

<table>
<thead>
<tr>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00..12.60</td>
<td>260</td>
<td>4</td>
</tr>
</tbody>
</table>

-> 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
Pseudocode for Nested Loop Join with Inner Index Scan

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

<table>
<thead>
<tr>
<th>Plan</th>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit</td>
<td>0.00..1.83</td>
<td>1</td>
<td>258</td>
</tr>
<tr>
<td>Nested Loop</td>
<td>0.00..477.02</td>
<td>260</td>
<td>258</td>
</tr>
<tr>
<td>Index Scan using i_sample2 on sample2</td>
<td>0.00..52.15</td>
<td>260</td>
<td>258</td>
</tr>
<tr>
<td>Index Scan using i_sample1 on sample1</td>
<td>0.00..1.62</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Index Cond: (sample1.id = sample2.id)
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/