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.

https://momjian.us/presentations

Creative Commons Attribution License

Last updated: June 2024
Postgres Query Execution
Postgres Query Execution

- Main
- Postmaster
- Postgres
- Generate Plan
- Execute Plan
- Parse Statement
- Traffic Cop
- Rewrite Query
- Generate Paths
- Optimal Path
- Generate Plan
- Utility Command
- Utilities
- Catalog
- Storage Managers
- Access Methods
- Nodes/Lists

https://momjian.us/main/presentations/internals.html#internal_pics
Postgres Query Execution

https://www.highgo.ca/2024/01/26/a-comprehensive-overview-of-postgresql-query-processing-stages/
What Decisions Does the Optimizer Have to Make?

- Scan Method
- Join Method
- Join Order

This blog entry has a great description of the optimizer internals: https://www.highgo.ca/2024/03/22/understand-postgresqls-planner-simple-scan-paths-vs-plans/
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;
```

```
relname
_______________________________
_pg_foreign_data_wrappers
_pg_foreign_servers
_pg_foreign_table_columns
_pg_foreign_tables
_pg_user_mappings
administrable_role_authorizations
applicable_roles
attributes
```
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>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</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);
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?

| letter | count | %   |
|--------+-------+-----|
| p      | 342   | 83.4|
| c      | 13    | 3.2 |
| r      | 12    | 2.9 |
| f      | 6     | 1.5 |
| s      | 6     | 1.5 |
| t      | 6     | 1.5 |
| u      | 5     | 1.2 |
| _      | 5     | 1.2 |
| d      | 4     | 1.0 |
| v      | 4     | 1.0 |
| a      | 3     | 0.7 |
| e      | 2     | 0.5 |
| k      | 1     | 0.2 |
| i      | 1     | 0.2 |
EXPLAIN SELECT letter
FROM sample
WHERE letter = 'p';

QUERY PLAN

-----------------------------------------------------------------------
Bitmap Heap Scan on sample (cost=4.16..10.07 rows=2 width=32)
  Recheck Cond: (letter = 'p'::text)
  -> Bitmap Index Scan on i_sample (cost=0.00..4.16 rows=2 width=0)
    Index Cond: (letter = 'p'::text)
Is the Distribution Important?

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

QUERY PLAN

-----------------------------------------------------------------------
Bitmap Heap Scan on sample (cost=4.16..10.07 rows=2 width=32)
  Recheck Cond: (letter = 'd'::text)
  -> Bitmap Index Scan on i_sample (cost=0.00..4.16 rows=2 width=0)
     Index Cond: (letter = 'd'::text)
Is the Distribution Important?

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

QUERY PLAN

-----------------------------------------------------------------------
| Bitmap Heap Scan on sample (cost=4.16..10.07 rows=2 width=32) |
| Recheck Cond: (letter = 'i)::text) |
| -> Bitmap Index Scan on i_sample (cost=0.00..4.16 rows=2 width=0) |
| Index Cond: (letter = 'i)::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..21.12 rows=342 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

DATA DATA DATA DATA DATA DATA DATA DATA DATA

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=4.18..14.23 rows=4 width=2)
  Recheck Cond: (letter = 'd'::text)
  -> Bitmap Index Scan on i_sample (cost=0.00..4.18 rows=4 width=0)
    Index Cond: (letter = 'd'::text)
Bitmap Index Scan

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

\[
\begin{array}{c|c|c|c}
\text{Index 1} & \text{Index 2} & \text{Combined} \\
0 & 0 & 0 \\
1 & 1 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
\end{array}
\]

'\text{A} AND '\text{NS}'
EXPLAIN SELECT letter
FROM sample
WHERE letter = 'i';

QUERY PLAN

Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2)
  Index Cond: (letter = 'i'::text)
Index Scan

Index

< Key = >

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
D A T A
Index-Only Scan

Index

< Key = >

< Key = >

< Key = >
Let's Look at All Values and their Effects

```sql
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>letter</th>
<th>count</th>
<th>lookup_letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>342</td>
<td>Seq Scan on sample (cost=0.00..21.12 rows=342 width=2)</td>
</tr>
<tr>
<td>p</td>
<td>342</td>
<td>Filter: (letter = 'p':text)</td>
</tr>
<tr>
<td>c</td>
<td>13</td>
<td>Bitmap Heap Scan on sample (cost=4.25..20.69 rows=13 width=2)</td>
</tr>
<tr>
<td>c</td>
<td>13</td>
<td>Recheck Cond: (letter = 'c':text)</td>
</tr>
<tr>
<td>c</td>
<td>13</td>
<td>-&gt; Bitmap Index Scan on i_sample (cost=0.00..4.25 rows=13 width=0)</td>
</tr>
<tr>
<td>c</td>
<td>13</td>
<td>Index Cond: (letter = 'c':text)</td>
</tr>
<tr>
<td>r</td>
<td>12</td>
<td>Bitmap Heap Scan on sample (cost=4.24..20.14 rows=12 width=2)</td>
</tr>
<tr>
<td>r</td>
<td>12</td>
<td>Recheck Cond: (letter = 'r':text)</td>
</tr>
<tr>
<td>r</td>
<td>12</td>
<td>-&gt; Bitmap Index Scan on i_sample (cost=0.00..4.24 rows=12 width=0)</td>
</tr>
<tr>
<td>r</td>
<td>12</td>
<td>Index Cond: (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;
| l | count | lookup_letter |
|---+-------+----------------------------------------------------------------|
| p | 342   | Seq Scan on sample (cost=0.00..21.12 rows=342 width=2)            |
| c | 13    | Bitmap Heap Scan on sample (cost=4.25..20.69 rows=13 width=2)      |
| r | 12    | Bitmap Heap Scan on sample (cost=4.24..20.14 rows=12 width=2)      |
| f | 6     | Bitmap Heap Scan on sample (cost=4.19..17.25 rows=6 width=2)       |
| t | 6     | Bitmap Heap Scan on sample (cost=4.19..17.25 rows=6 width=2)       |
| s | 6     | Bitmap Heap Scan on sample (cost=4.19..17.25 rows=6 width=2)       |
| u | 5     | Bitmap Heap Scan on sample (cost=4.19..15.86 rows=5 width=2)       |
| _ | 5     | Bitmap Heap Scan on sample (cost=4.19..15.86 rows=5 width=2)       |
| d | 4     | Bitmap Heap Scan on sample (cost=4.18..14.23 rows=4 width=2)       |
| v | 4     | Bitmap Heap Scan on sample (cost=4.18..14.23 rows=4 width=2)       |
| a | 3     | Bitmap Heap Scan on sample (cost=4.17..12.31 rows=3 width=2)       |
| e | 2     | Bitmap Heap Scan on sample (cost=4.16..10.07 rows=2 width=2)       |
| k | 1     | Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2) |
| i | 1     | Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2) |

Results will vary based on the clustering of values in heap pages.
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>l</th>
<th>count</th>
<th>lookup_letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>342</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..56.35 rows=342 width=2)</td>
</tr>
<tr>
<td>c</td>
<td>13</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..27.69 rows=13 width=2)</td>
</tr>
<tr>
<td>r</td>
<td>12</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..25.53 rows=12 width=2)</td>
</tr>
<tr>
<td>s</td>
<td>6</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..18.98 rows=6 width=2)</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..18.98 rows=6 width=2)</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..18.98 rows=6 width=2)</td>
</tr>
<tr>
<td>u</td>
<td>5</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..16.82 rows=5 width=2)</td>
</tr>
<tr>
<td>_</td>
<td>5</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..16.82 rows=5 width=2)</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..14.66 rows=4 width=2)</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..14.66 rows=4 width=2)</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..12.49 rows=3 width=2)</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..10.33 rows=2 width=2)</td>
</tr>
<tr>
<td>k</td>
<td>1</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2)</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2)</td>
</tr>
</tbody>
</table>

RESET ALL;
| l | count | lookup_letter |
|---+-------+----------------|
| p | 342   | Seq Scan on sample (cost=0.00..21.12 rows=342 width=2) |
| c | 13    | Bitmap Heap Scan on sample (cost=4.25..20.69 rows=13 width=2) |
| r | 12    | Bitmap Heap Scan on sample (cost=4.24..20.14 rows=12 width=2) |
| f | 6     | Bitmap Heap Scan on sample (cost=4.19..17.25 rows=6 width=2) |
| t | 6     | Bitmap Heap Scan on sample (cost=4.19..17.25 rows=6 width=2) |
| s | 6     | Bitmap Heap Scan on sample (cost=4.19..17.25 rows=6 width=2) |
| u | 5     | Bitmap Heap Scan on sample (cost=4.19..15.86 rows=5 width=2) |
| _ | 5     | Bitmap Heap Scan on sample (cost=4.19..15.86 rows=5 width=2) |
| d | 4     | Bitmap Heap Scan on sample (cost=4.18..14.23 rows=4 width=2) |
| v | 4     | Bitmap Heap Scan on sample (cost=4.18..14.23 rows=4 width=2) |
| a | 3     | Bitmap Heap Scan on sample (cost=4.17..12.31 rows=3 width=2) |
| e | 2     | Bitmap Heap Scan on sample (cost=4.16..10.07 rows=2 width=2) |
| k | 1     | Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2) |
| i | 1     | Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2) |
Which Join Method?

- Nested Loop
  - With Inner Sequential Scan
  - With Inner Index Scan
- Hash Join
- Merge Join
What Is in `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>3</td>
</tr>
<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>
</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..364.14 rows=770 width=32)
  ->  Seq Scan on sample1  (cost=0.00..313.09 rows=77 width=4)
      Filter: (id = '33':'oid')
  ->  Materialize  (cost=0.00..41.45 rows=10 width=36)
      ->  Seq Scan on sample2  (cost=0.00..41.40 rows=10 width=36)
          Filter: (id = '33':'oid')
Nested Loop Join with Inner Sequential Scan

Outer
- aag
- aay
- aar
- aai

Inner
- aai
- aag
- aas
- aar
- aay
- aaa
- aag

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

Hash Join  (cost=49.86..2189.32 rows=52017 width=32)
    Hash Cond: (sample1.id = sample2.id)
    ->  Seq Scan on sample1  (cost=0.00..274.67 rows=15367 width=36)
    ->  Hash  (cost=41.40..41.40 rows=677 width=4)
          ->  Seq Scan on sample2  (cost=0.00..41.40 rows=677 width=4)
              Filter: (id > '33'::oid)
Hash Join

Outer

- aay
- aag
- aak
- aar

Inner

- aak
- aas
- aam
- aay
- aar
- aao
- aaw

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=1491.22..3843.32 rows=156129 width=32) |
| Merge Cond: (sample2.id = sample1.id) |
| -> Sort  (cost=147.97..153.05 rows=2032 width=4) |
|   Sort Key: sample2.id |
|   -> Seq Scan on sample2  (cost=0.00..36.32 rows=2032 width=4) |
| -> Sort  (cost=1343.26..1381.67 rows=15367 width=36) |
|   Sort Key: sample1.id |
|   -> Seq Scan on sample1  (cost=0.00..274.67 rows=15367 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++;
    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=1491.22..3843.32 rows=156129 width=32) |
| Merge Cond: | (sample2.id = sample1.id)                     |
| -> Sort     | (cost=147.97..153.05 rows=2032 width=36)     |
| Sort Key:   | sample2.id                                    |
| -> Seq Scan | (cost=0.00..36.32 rows=2032 width=36)         |
| -> Sort     | (cost=1343.26..1381.67 rows=15367 width=4)   |
| Sort Key:   | sample1.id                                    |
| -> Seq Scan | (cost=0.00..274.67 rows=15367 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.
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

Hash Join (cost=25.38..195.17 rows=417 width=254)
  Hash Cond: (sample1.id = sample2.id)
  -> Seq Scan on sample1 (cost=0.00..153.45 rows=3245 width=4)
  -> Hash (cost=20.17..20.17 rows=417 width=258)
    -> Seq Scan on sample2 (cost=0.00..20.17 rows=417 width=258)
EXPLAIN SELECT sample1.junk
FROM sample1 RIGHT OUTER JOIN sample2 ON (sample1.id = sample2.id);

QUERY PLAN

Hash Right Join (cost=25.38..195.17 rows=417 width=254)
Hash Cond: (sample1.id = sample2.id)
  -> Seq Scan on sample1 (cost=0.00..153.45 rows=3245 width=258)
  -> Hash (cost=20.17..20.17 rows=417 width=4)
    -> Seq Scan on sample2 (cost=0.00..20.17 rows=417 width=4)
EXPLAIN SELECT sample1.junk
FROM sample1 CROSS JOIN sample2;

QUERY PLAN

-------------------------------
Nested Loop  (cost=0.00..17089.22 rows=1353165 width=254)
  ->  Seq Scan on sample1  (cost=0.00..153.45 rows=3245 width=254)
  ->  Materialize  (cost=0.00..22.26 rows=417 width=0)
      ->  Seq Scan on sample2  (cost=0.00..20.17 rows=417 width=0)
CREATE INDEX i_sample1 on sample1 (id);

CREATE INDEX i_sample2 on sample2 (id);
Nested Loop with Inner Index Scan Now Possible

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

QUERY PLAN

<table>
<thead>
<tr>
<th>Nested Loop (cost=0.55..16.60 rows=1 width=254)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&gt; Index Only Scan using i_sample1 on sample1 (cost=0.28..8.30 rows=1 width=4)</td>
</tr>
<tr>
<td>Index Cond: (id = '33'::oid)</td>
</tr>
<tr>
<td>-&gt; Index Scan using i_sample2 on sample2 (cost=0.27..8.29 rows=1 width=258)</td>
</tr>
<tr>
<td>Index Cond: (id = '33'::oid)</td>
</tr>
</tbody>
</table>
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=0.28..29.52 rows=1 width=254)
  -> Seq Scan on sample2 (cost=0.00..21.21 rows=1 width=258)
      Filter: (junk ~ '^aaa'::text)
  -> Index Only Scan using i_sample1 on sample1 (cost=0.28..8.30 rows=1 width=4)
      Index Cond: (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=26.42..196.21 rows=417 width=254)
  Hash Cond: (sample1.id = sample2.id)
     ->  Seq Scan on sample1 (cost=0.00..153.45 rows=3245 width=4)
     ->  Hash  (cost=21.21..21.21 rows=417 width=258)
          ->  Seq Scan on sample2 (cost=0.00..21.21 rows=417 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)
ORDER BY 1;

QUERY PLAN

<table>
<thead>
<tr>
<th>Plan</th>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort (cost=213.32..214.36)</td>
<td>213.32</td>
<td>417</td>
<td>254</td>
</tr>
<tr>
<td>Sort Key: sample2.junk</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-&gt; Hash Join (cost=25.38..195.17)</td>
<td>25.38</td>
<td>417</td>
<td>254</td>
</tr>
<tr>
<td>Hash Cond: (sample1.id = sample2.id)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-&gt; Seq Scan on sample1 (cost=0.00..153.45)</td>
<td>0.00</td>
<td>3245</td>
<td>4</td>
</tr>
<tr>
<td>-&gt; Hash (cost=20.17..20.17)</td>
<td>20.17</td>
<td>417</td>
<td>258</td>
</tr>
<tr>
<td>-&gt; Seq Scan on sample2 (cost=0.00..20.17)</td>
<td>0.00</td>
<td>417</td>
<td>258</td>
</tr>
</tbody>
</table>
LIMIT Can Affect Join Usage

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

QUERY PLAN

Limit (cost=0.55..2.33 rows=1 width=258)
  -> Nested Loop (cost=0.55..742.75 rows=417 width=258)
    -> Index Scan using i_sample2 on sample2 (cost=0.27..86.52 rows=417 width=258)
    -> Index Only Scan using i_sample1 on sample1 (cost=0.28..1.56 rows=1 width=4)
      Index Cond: (id = sample2.id)

Sort is unneeded since an index is being used on the outer side.
EXPLAIN SELECT sample2.id, sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
ORDER BY 1
LIMIT 10;

QUERY PLAN

Limit (cost=0.55..18.35 rows=10 width=258)
  -> Nested Loop (cost=0.55..742.75 rows=417 width=258)
    -> Index Scan using i_sample2 on sample2 (cost=0.27..86.52 rows=417 width=258)
    -> Index Only Scan using i_sample1 on sample1 (cost=0.28..1.56 rows=1 width=4)
      Index Cond: (id = sample2.id)
Limit 100 Switches to Merge 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=11.00..170.51 rows=100 width=258)
  -> Merge Join (cost=11.00..676.13 rows=417 width=258)
     Merge Cond: (sample1.id = sample2.id)
       -> Index Only Scan using i_sample1 on sample1 (cost=0.28..576.91 rows=3245)
       -> Index Scan using i_sample2 on sample2 (cost=0.27..86.52 rows=417 width=258)

Merge join is normally used for large joins, but the indexes eliminate the need for sorting both sides band LIMIT reduces the number of index entries that need to be accessed.
EXPLAIN SELECT sample2.id, sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
ORDER BY 1
LIMIT 1000;

QUERY PLAN

Limit (cost=213.32..214.36 rows=417 width=258)
  -> Sort (cost=213.32..214.36 rows=417 width=258)
    Sort Key: sample2.id
  -> Hash Join (cost=25.38..195.17 rows=417 width=258)
    Hash Cond: (sample1.id = sample2.id)
    -> Seq Scan on sample1 (cost=0.00..153.45 rows=3245 width=4)
    -> Hash (cost=20.17..20.17 rows=417 width=258)
      -> Seq Scan on sample2 (cost=0.00..20.17 rows=417 width=258)

For LIMIT 1000, index lookups are considered to be too expensive to partially execute
the join, so a hash join is fully executed, which is then sorted and the LIMIT applied.
VACUUM Causes Merge Join Again

-- updates the visibility map
VACUUM sample1, sample2;

EXPLAIN SELECT sample2.id, sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
ORDER BY 1
LIMIT 1000;

QUERY PLAN

Limit (cost=40.67..150.78 rows=420 width=258)
  -> Merge Join (cost=40.67..150.78 rows=420 width=258)
     Merge Cond: (sample1.id = sample2.id)
     -> Index Only Scan using i_sample1 on sample1 (cost=0.28..97.75 rows=3298)
     -> Sort (cost=38.50..39.55 rows=420 width=258)
        Sort Key: sample2.id
        -> Seq Scan on sample2 (cost=0.00..20.20 rows=420 width=258)

VACUUM reduces the cost of index-only scans by making heap access less likely.

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

QUERY PLAN

Merge Join  (cost=40.67..150.78 rows=420 width=258)
  Merge Cond: (sample1.id = sample2.id)
  ->  Index Only Scan using i_sample1 on sample1  (cost=0.28..97.75 rows=3298 width=4)
  ->  Sort  (cost=38.50..39.55 rows=420 width=258)
      Sort Key: sample2.id
  ->  Seq Scan on sample2  (cost=0.00..20.20 rows=420 width=258)
## Same Join, Different Plans

<table>
<thead>
<tr>
<th>Query Modifier</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>No LIMIT</td>
<td>Hash join</td>
</tr>
<tr>
<td>LIMIT 1</td>
<td>Nested loop join with two index scans</td>
</tr>
<tr>
<td>LIMIT 10</td>
<td>“”</td>
</tr>
<tr>
<td>LIMIT 100</td>
<td>Merge join with two index scans</td>
</tr>
<tr>
<td>LIMIT 1000</td>
<td>Hash join</td>
</tr>
<tr>
<td>VACUUM, LIMIT 1000</td>
<td>Merge join with index-only scan and sort</td>
</tr>
<tr>
<td>No LIMIT</td>
<td>“”</td>
</tr>
</tbody>
</table>

The last two are different from previous matching lines because of VACUUM.
Conclusion

https://momjian.us/presentations

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