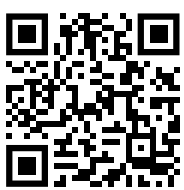
Explaining the Postgres Query Optimizer

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.

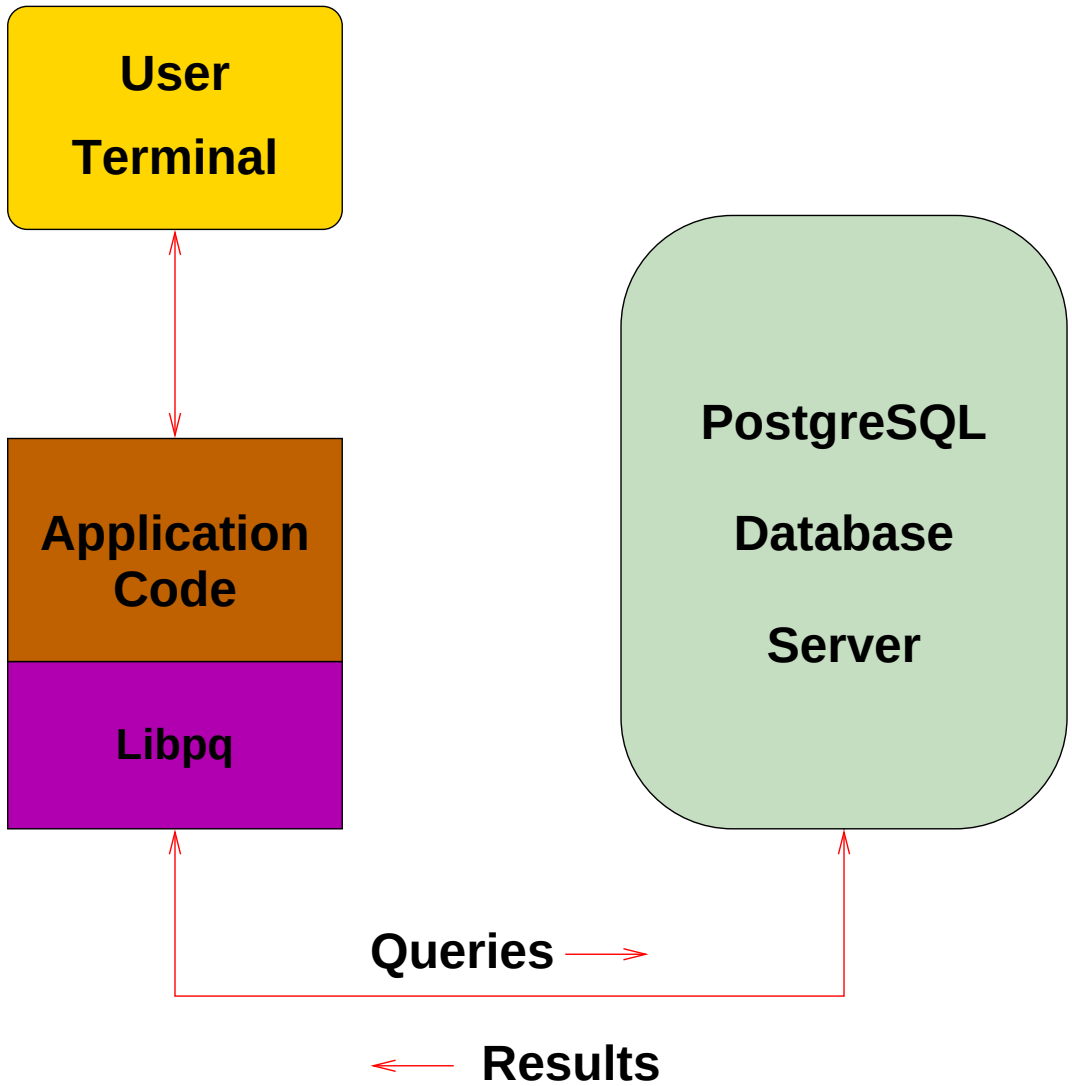
https://momjian.us/presentations

Creative Commons Attribution License

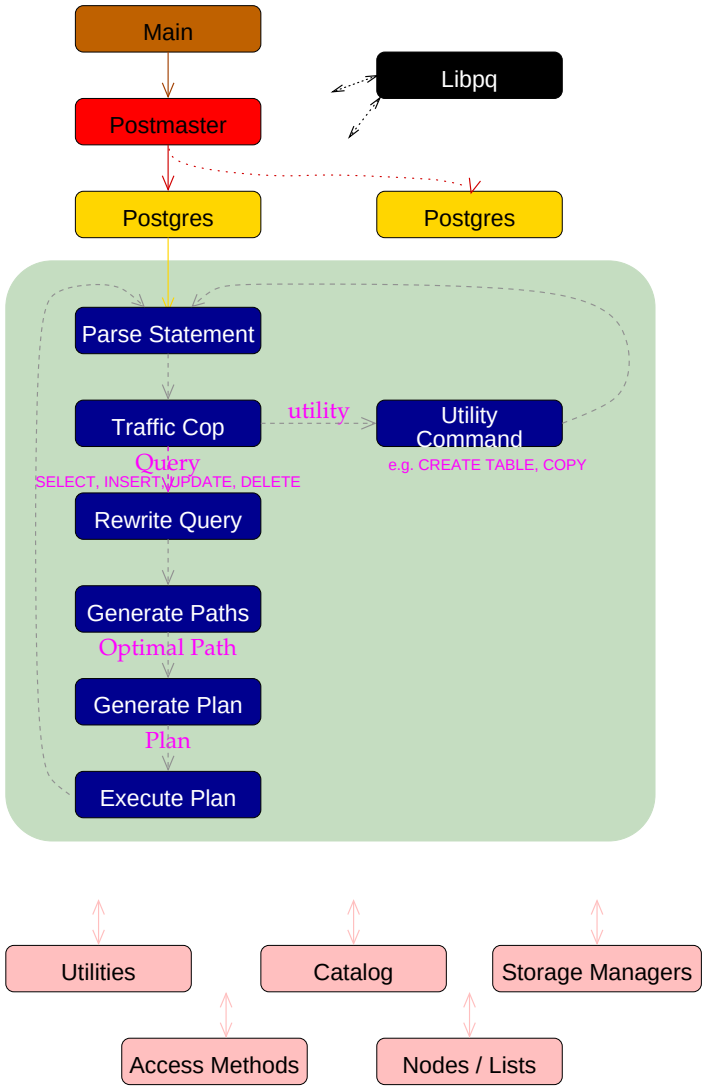


Last updated: July, 2020

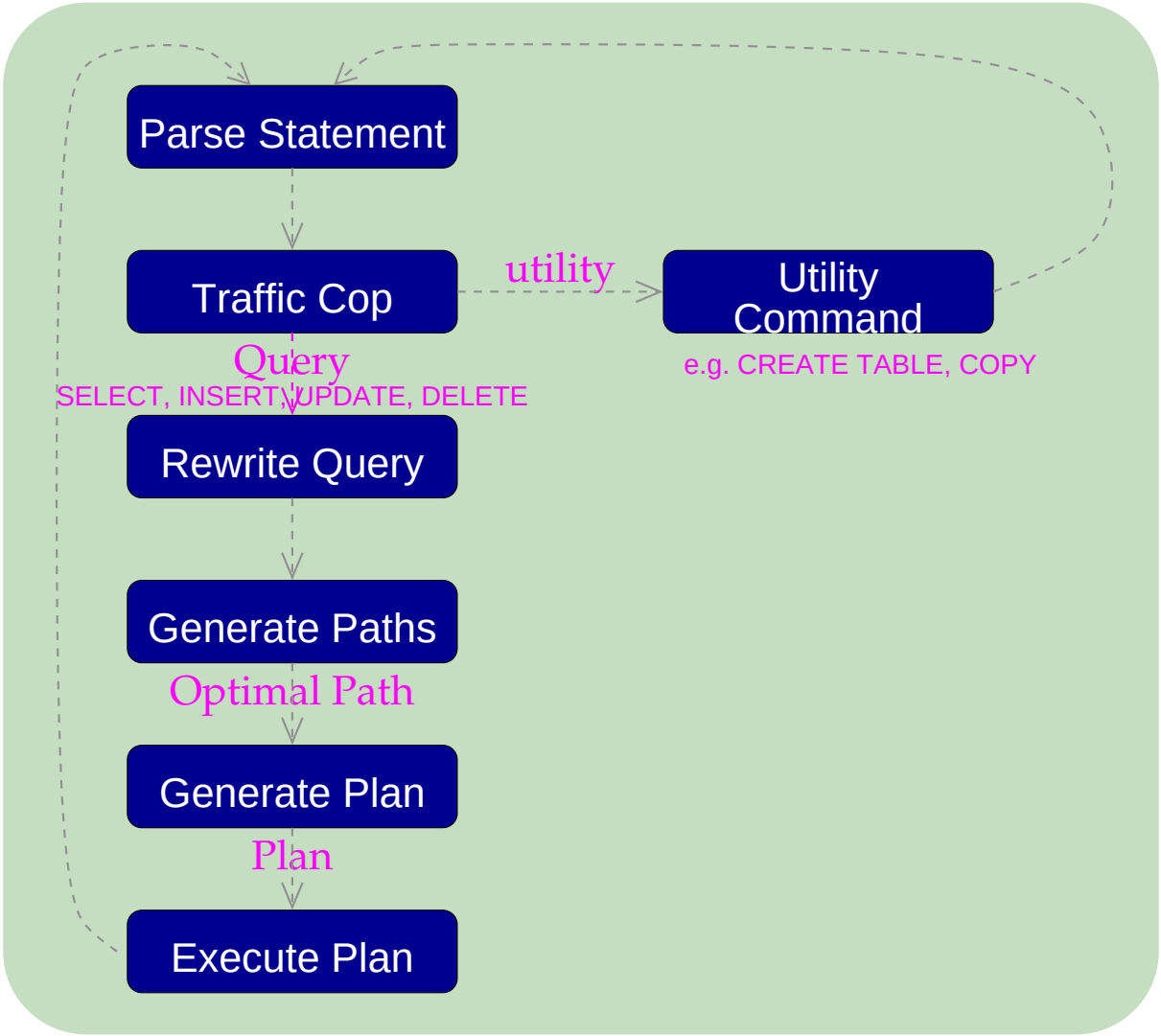
Postgres Query Execution



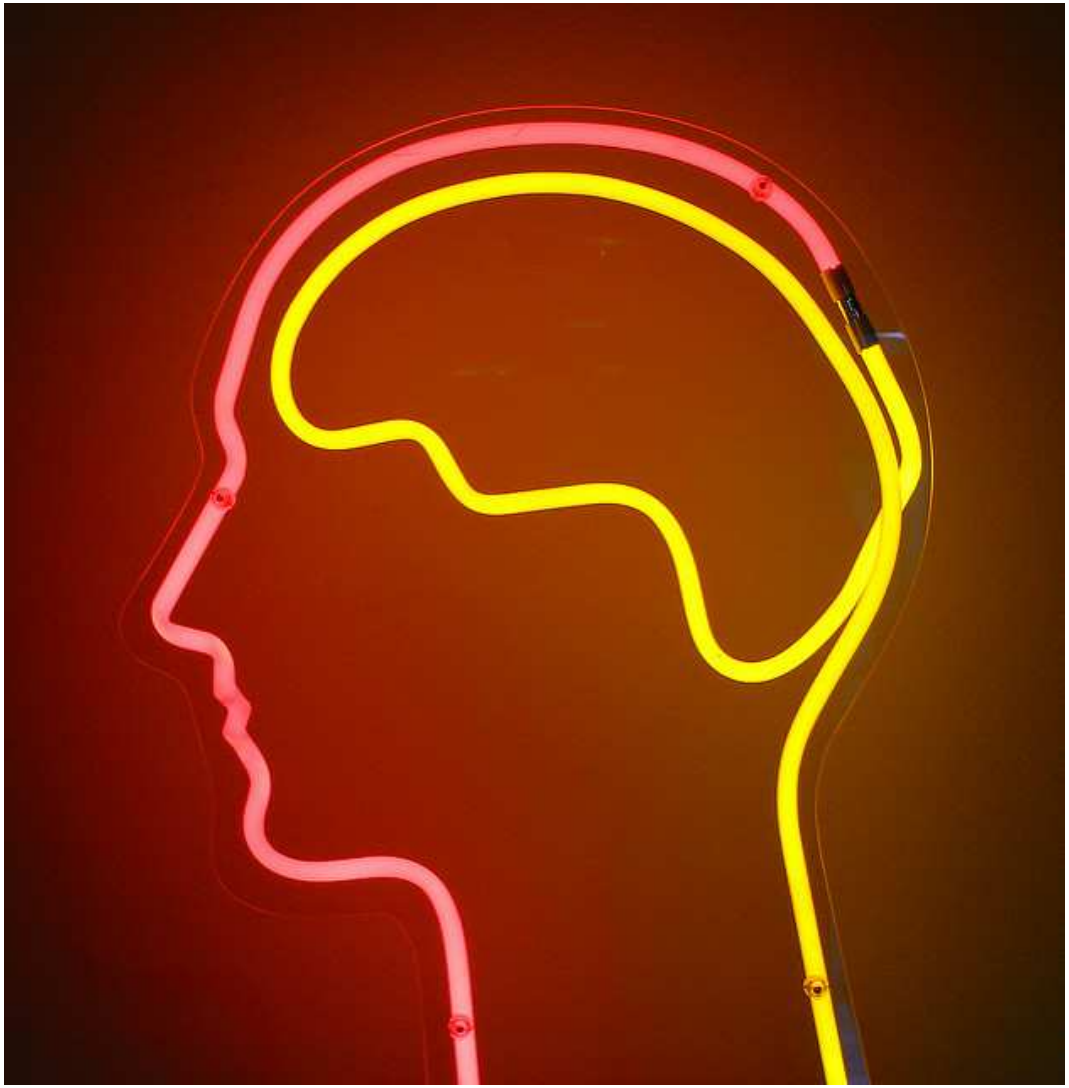
Postgres Query Execution



Postgres Query Execution



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*

```
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
```

Let's Use Just the First Letter of *pg_class.relname*

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

 _
 _
 _
 a
 a
 a
 c
 c
```

Create a Temporary Table with an Index

```
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 https://momjian.us/main/writings/pgsql/optimizer.sql.

Create an EXPLAIN Function

```
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	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?

```
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?

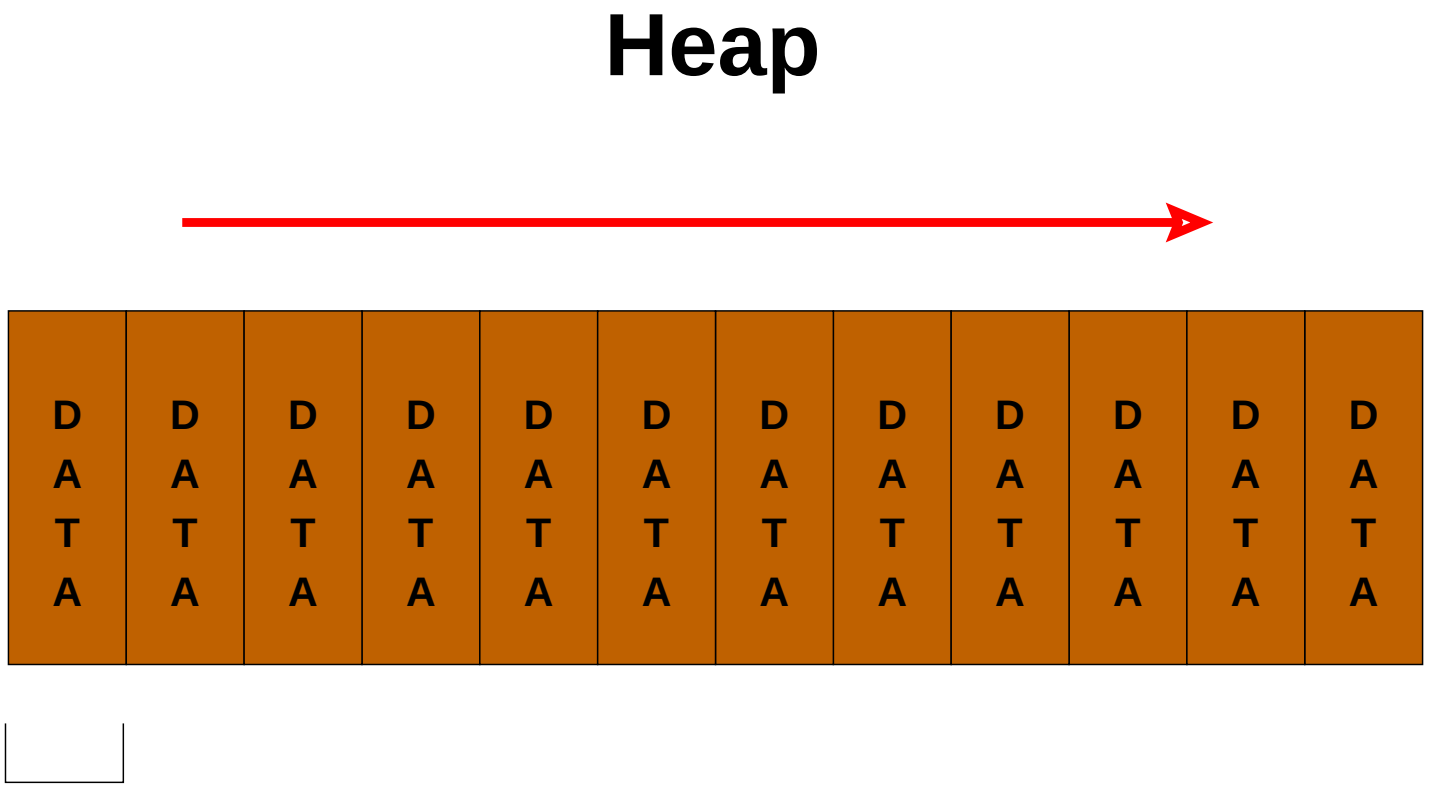
```
EXPLAIN SELECT letter
FROM sample
WHERE letter = 'k';
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using i_sample on sample  (cost=0.00..8.27 rows=1 width=32)
   Index Cond: (letter = 'k'::text)
```

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.

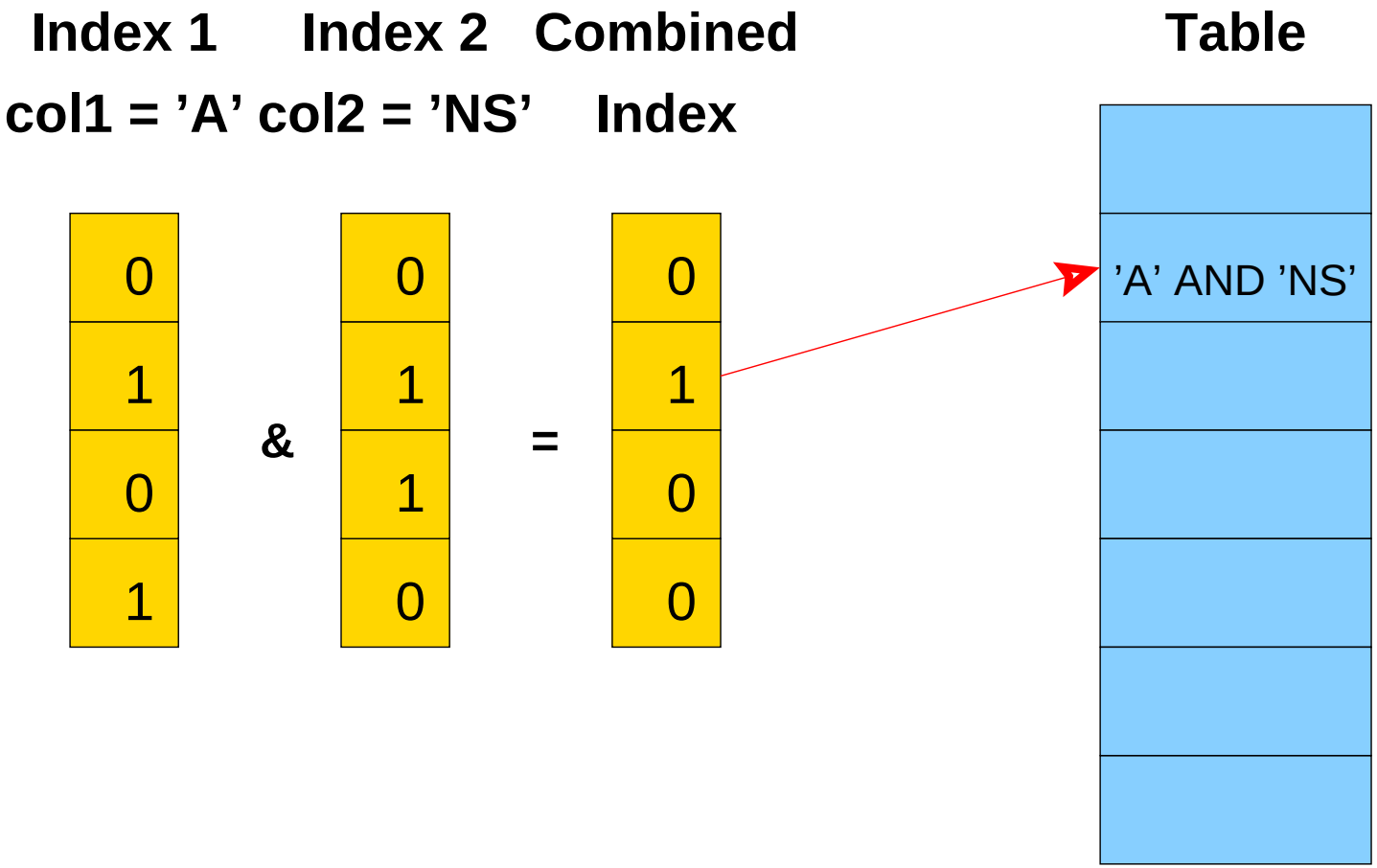
Sequential Scan



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)
```

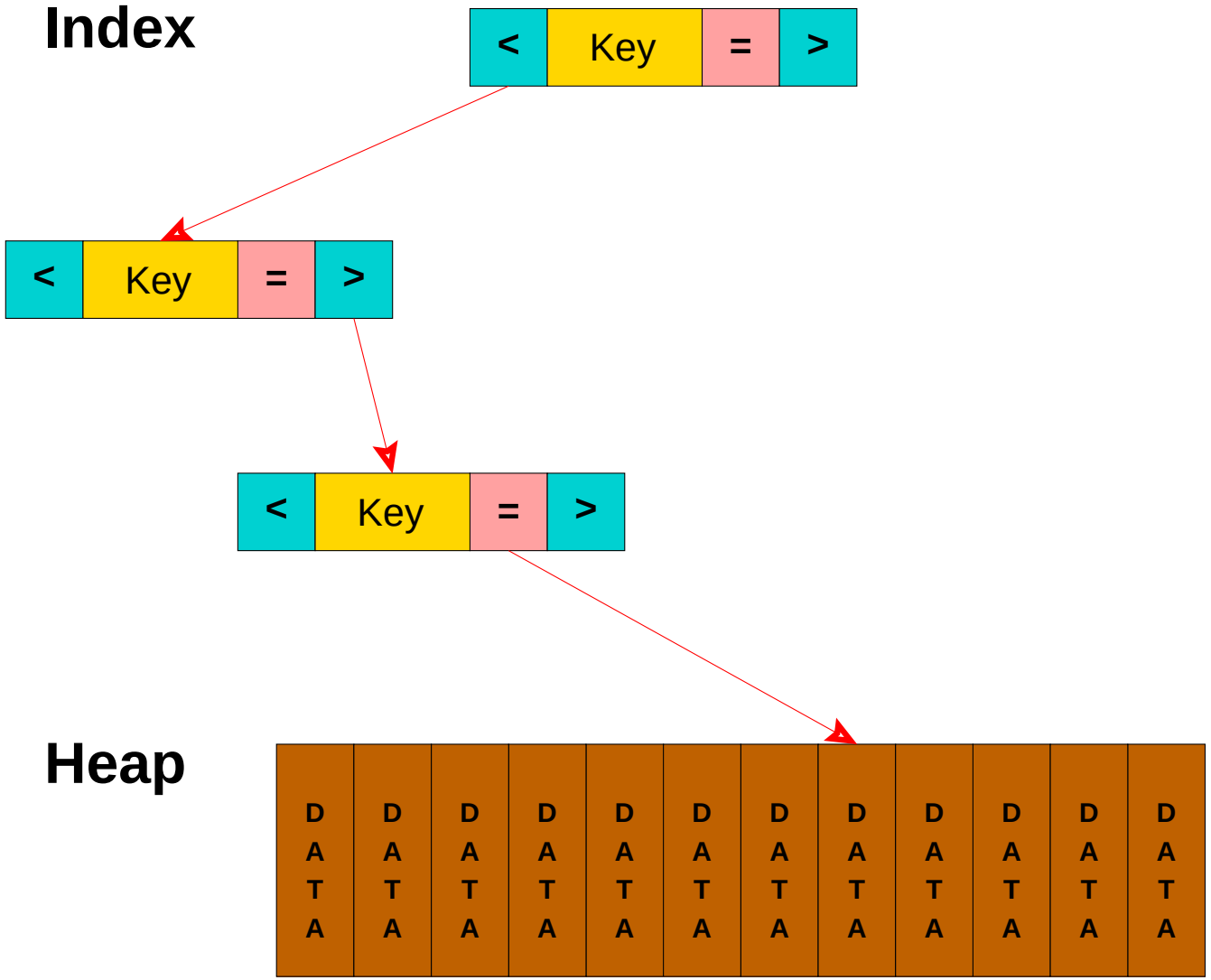
Bitmap Index Scan



An Even Rarer Value Causes an Index Scan

```
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 = >
< Key = >
< Key = >
Heap
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA

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;
```

```
 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)
```

...

OK, Just the First Lines

```
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;
```

Just the First EXPLAIN Lines

```
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)
```

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

```
l | count |                                lookup_letter
---+-------+-----------------------------------------------------------------------
 p |   199 | Index Scan using i_sample on sample  (cost=0.00..39.33 rows=199 width=
 s |     9 | Index Scan using i_sample on sample  (cost=0.00..22.14 rows=9 width=2)
 c |     8 | Index Scan using i_sample on sample  (cost=0.00..19.84 rows=8 width=2)
 r |     7 | Index Scan using i_sample on sample  (cost=0.00..19.82 rows=7 width=2)
 t |     5 | Index Scan using i_sample on sample  (cost=0.00..15.21 rows=5 width=2)
 d |     4 | Index Scan using i_sample on sample  (cost=0.00..15.19 rows=4 width=2)
 v |     4 | Index Scan using i_sample on sample  (cost=0.00..15.19 rows=4 width=2)
 f |     4 | Index Scan using i_sample on sample  (cost=0.00..15.19 rows=4 width=2)
 _ |     3 | Index Scan using i_sample on sample  (cost=0.00..12.88 rows=3 width=2)
 a |     3 | Index Scan using i_sample on sample  (cost=0.00..12.88 rows=3 width=2)
 u |     3 | Index Scan using i_sample on sample  (cost=0.00..12.88 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)

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 *pg_proc.oid*?

```
SELECT oid
FROM pg_proc
ORDER BY 1
LIMIT 8;
 oid
-----
  31
  33
  34
  35
  38
  39
  40
  41
```

Create Temporary Tables from *pg_proc* and *pg_class*

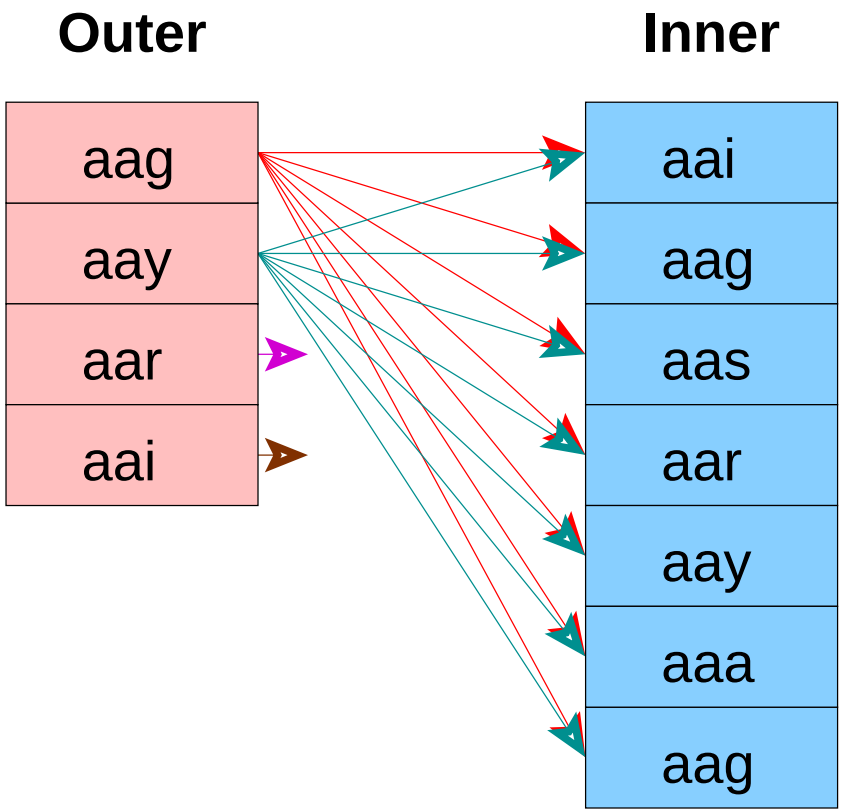
```
CREATE TEMPORARY TABLE sample1 (id, junk) AS
        SELECT oid, repeat('x', 250)
        FROM pg_proc
        ORDER BY random();  -- add rows in random order

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

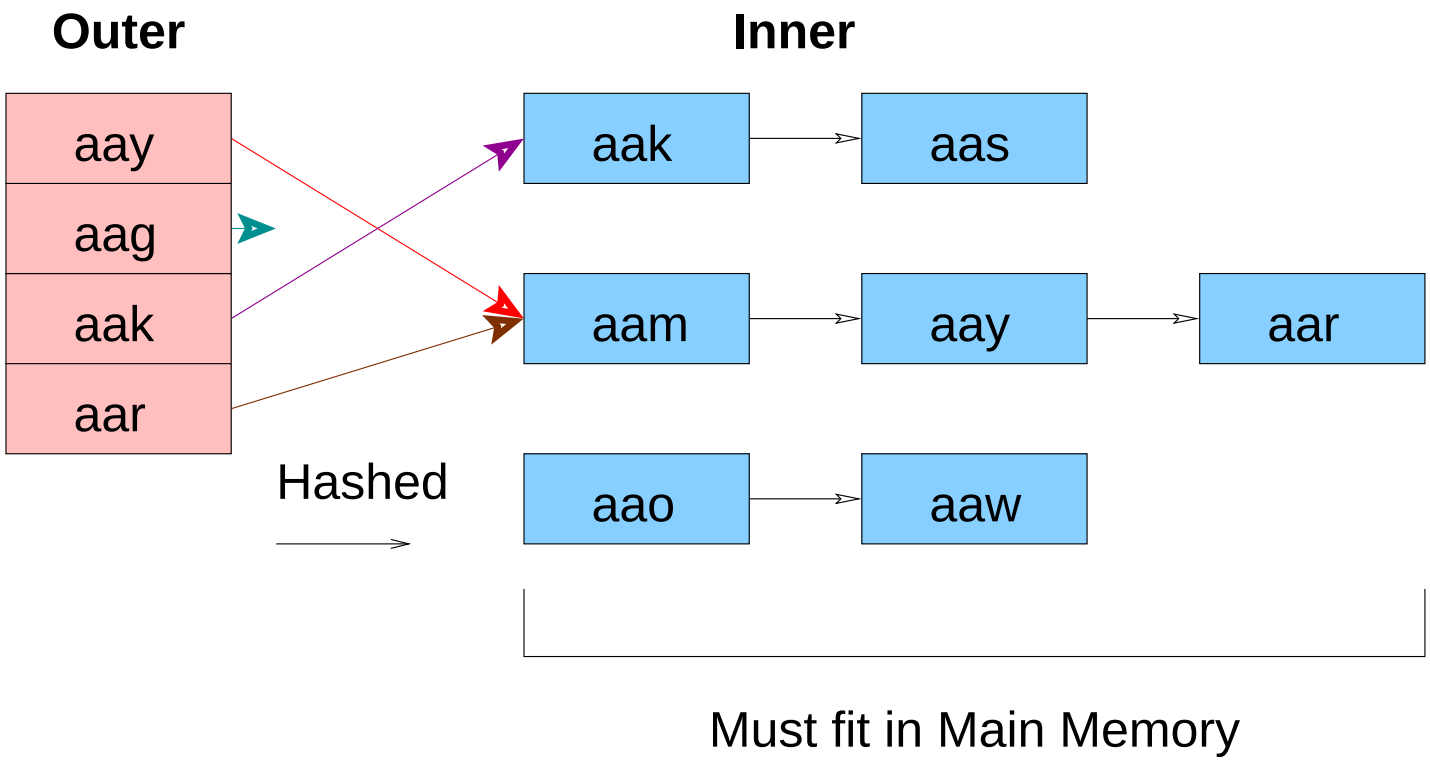
Pseudocode for Nested Loop Join with Inner Sequential Scan

```
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
Inner
aay
aag
aak
aar
Hashed
aak
aas
aam
aay
aar
aao
aaw
Must fit in Main Memory

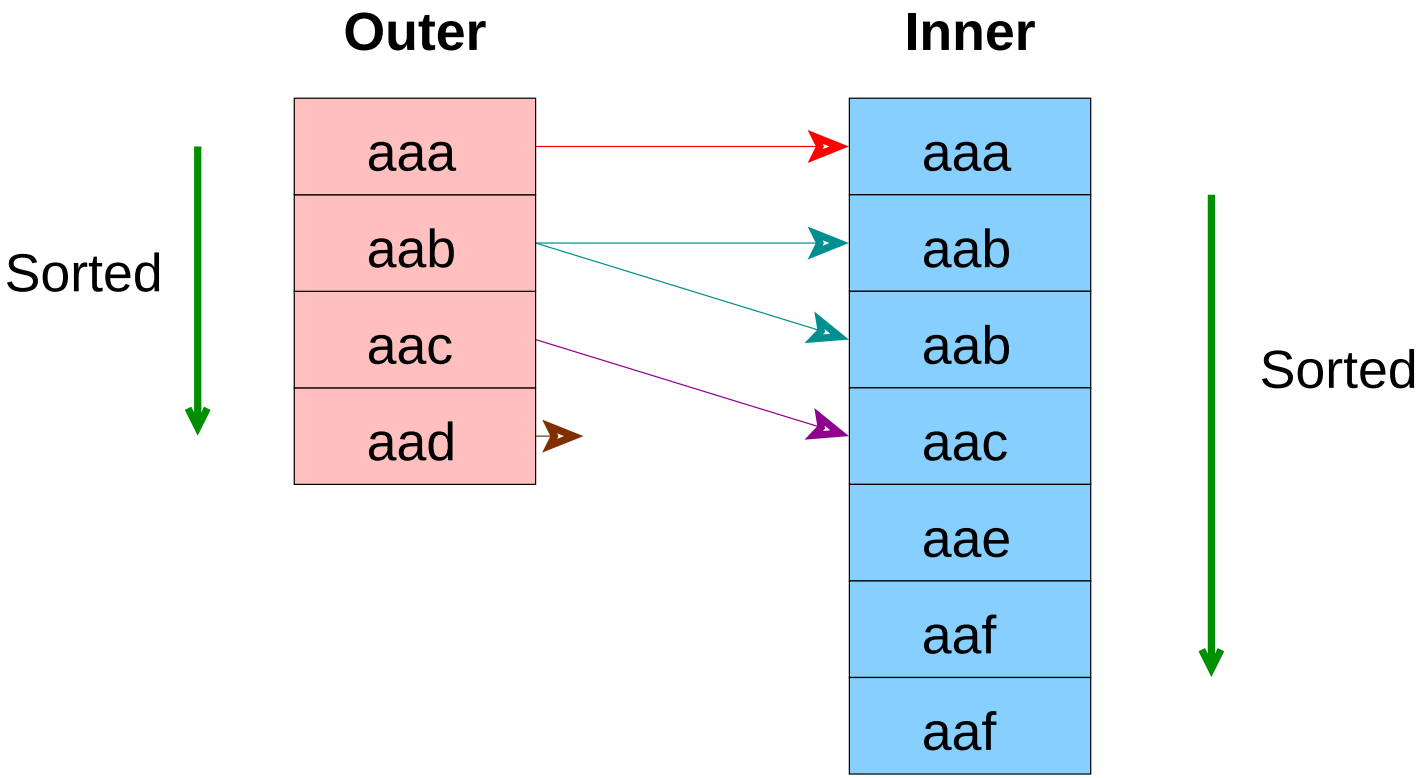
Pseudocode for Hash Join

```
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]);
```

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)
```

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;
```

Order of Joined Relations Is Insignificant

```
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
-------------------------------------------------------------------------
 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
-------------------------------------------------------------------------
 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)
```

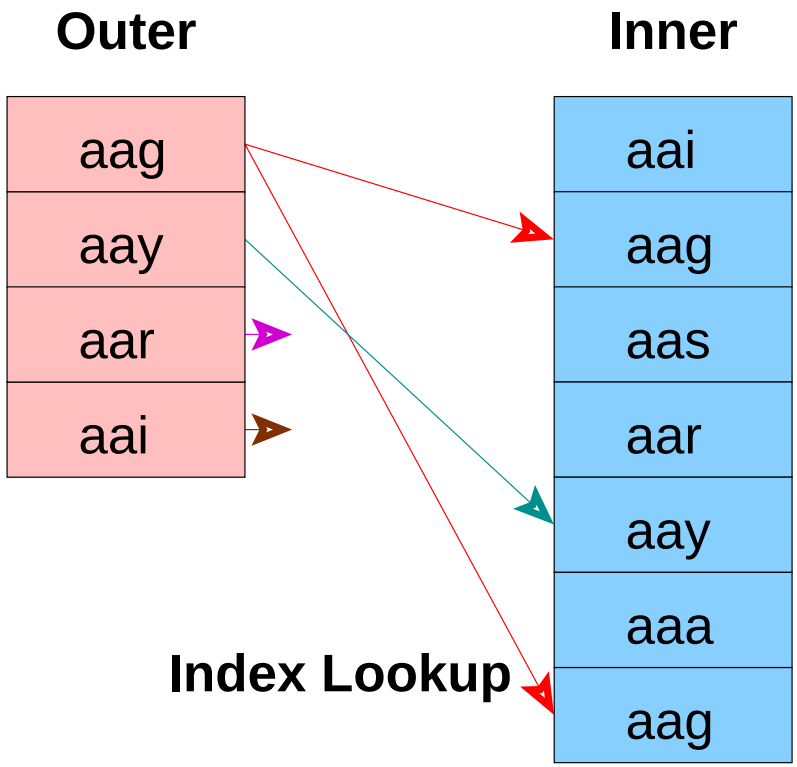
Create Indexes

```
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
-------------------------------------------------------------------------------
 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);
```

Query Restrictions Affect Join Usage

```
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'.

All 'junk' Columns Begin with 'xxx'

```
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)
```

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=0.00..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)
```

LIMIT 10

```
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

https://momjian.us/presentations

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