Installation

MacOS

```
brew install postgresql@16
```

macOS packages

```
# Find installation directory
brew --prefix postgresql@16
# It will returan a path like: /opt/homebrew/opt/postgresql@16
```

- Add PostgreSQL to PATH:
  - Edit your shell configuration file (e.g., .zshrc or .bash_profile) and add following line:
  - `export PATH="/opt/homebrew/opt/postgresql@16/bin:$PATH"`
- Reload shell configuration: `source ~/.zshrc`
  - if using bash: `source ~/.bash_profile`
- Validate the installation: `psql --version`

Linux (Ubuntu)

```
sudo apt install postgresql
```

Linux downloads (Ubuntu)

Explaining the Postgres Query Optimizer

Video - Explaining the Postgres Query Optimizer
SQL file
Slides

```
-- create a temporary table `sample`
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 for `sample`
CREATE INDEX i_sample on sample (letter);

-- create a function `lookup_letter` to generate a result set containing the
query execution plan
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;

-- Show 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;
```

Scan Method

Is the Distribution important?

- Distribution of `sample` table

letter	count	%
p	345	83.5
c	13	3.1
r	12	2.9
f	6	1.5
s	6	1.5
t	6	1.5
u	5	1.2

letter	count	%
_	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

Example 1 - the most common value

```
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: `text`
-> Bitmap Index Scan on i_sample (cost=0.00..4.16 rows=2 width=0)
Index Cond: `text`

Explanation:

1. **Bitmap Heap Scan on sample (cost=4.16..10.07 rows=2 width=32)**:
   • **Bitmap Heap Scan**: This indicates that PostgreSQL will use a bitmap heap scan to retrieve the rows from the sample table.
   • **cost=4.16..10.07**: This represents the estimated cost of executing the query. The first number (4.16) is the startup cost, which is the cost before any rows are produced. The second number (10.07) is the total cost, which is the cost after all rows have been retrieved.
   • **rows=2**: This is the estimated number of rows that will be returned by the query.
   • **width=32**: This represents the estimated average size of each row in bytes.
2. **Recheck Cond: `text`** :
   • This indicates that the condition letter = 'p' will be rechecked for each row retrieved by the bitmap heap scan. This recheck is necessary to ensure that only the rows matching the condition are returned.

3. **Bitmap Index Scan on i_sample (cost=0.00..4.16 rows=2 width=0)**:
   • **Bitmap Index Scan**: This indicates that PostgreSQL will use a bitmap index scan on the i_sample index to find the rows that match the condition.
   • **cost=0.00..4.16**: This represents the estimated cost of performing the bitmap index scan. The first number (0.00) is the startup cost, and the second number (4.16) is the total cost.
   • **rows=2**: This is the estimated number of index entries that will be retrieved.
   • **width=0**: represents the estimated size of each index entry in bytes (which is usually zero since index entries are not full rows).
4. **Index Cond: `text`** :
   - This indicates that the bitmap index scan will use the condition `letter = 'p'` to find the relevant entries in the `i_sample` index.

Explanation of the Query Plan Steps

1. **Bitmap Index Scan on i_sample**:
   - PostgreSQL begins by scanning the `i_sample` index using the condition `letter = 'p'`. This step is efficient because it only scans the index, not the entire table.
   - The bitmap index scan is used to quickly identify the locations of the rows in the `sample` table that meet the condition.
2. **Bitmap Heap Scan on sample**:
   - After identifying the relevant rows using the bitmap index scan, PostgreSQL performs a bitmap heap scan on the `sample` table. This step involves accessing the actual table rows.
   - The bitmap heap scan uses the information from the bitmap index scan to locate the specific rows in the `sample` table.
   - The condition `letter = 'p'` is rechecked to ensure accuracy, as there might be some discrepancies or need for confirmation based on how the bitmap index scan marks the rows.

Cost Analysis

- The `cost=4.16..10.07` for the bitmap heap scan indicates that this operation is moderately expensive in terms of I/O and processing time. The initial cost (4.16) is relatively low, but the total cost (10.07) reflects the overall effort to retrieve and verify the rows.
- The `cost=0.00..4.16` for the bitmap index scan suggests that the index scan is quite efficient, with a minimal startup cost and a total cost of 4.16. This efficiency is due to the index allowing for quick lookups of the relevant rows.

Summary

In summary, the query plan shows an efficient use of the `i_sample` index to quickly find rows where `letter = 'p'` in the `sample` table. The bitmap index scan identifies the locations of the relevant rows, and the bitmap heap scan retrieves these rows from the table, rechecking the condition to ensure that only matching rows are returned. The use of these techniques minimizes the overall cost and improves the query's performance by avoiding a full table scan.

Example 2 - the rarest value

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

QUERY PLAN
Bitmap Heap Scan on sample (cost=4.16..10.07 rows=2 width=32)
Recheck Cond: `text`
-> Bitmap Index Scan on i_sample (cost=0.00..4.16 rows=2 width=0)
Index Cond: `text`

- exactly the same thing as the most common value 'p'
- Why?
  - Because there's **NO optimizer statistics** for table `sample`.
  - Autovacuum cannot ANALYZE (or VACUUM) temporary tables because these tables are only visible to the creating session.

Analyze

```
-- Run ANALYZE manually to get optimizer statistics for table `sample`
ANALYZE sample;
-- Repeat above EXPLAIN queries for 'p' and 'k'.
```

QUERY PLAN
Seq Scan on sample (cost=0.00..21.16 rows=345 width=2)
Filter: `text`

- A **Sequential Scan** is basically just read the whole table.

QUERY PLAN
Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2)
Index Cond: `text`

- An **Index Scan** is only used when the selectivity is roughly 5-10 percent.

```
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	345	Seq Scan on sample (cost=0.00..21.16 rows=345 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)
i	1	Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2)
k	1	Index Only Scan using i_sample on sample (cost=0.15..8.17 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;
```

l	count	lookup_letter
p	345	Index Only Scan using i_sample on sample (cost=0.15..56.39 rows=345 width=2)
c	13	Index Only Scan using i_sample on sample (cost=0.15..27.68 rows=13 width=2)
r	12	Index Only Scan using i_sample on sample (cost=0.15..25.52 rows=12 width=2)
s	6	Index Only Scan using i_sample on sample (cost=0.15..18.98 rows=6 width=2)
f	6	Index Only Scan using i_sample on sample (cost=0.15..18.98 rows=6 width=2)
t	6	Index Only Scan using i_sample on sample (cost=0.15..18.98 rows=6 width=2)
u	5	Index Only Scan using i_sample on sample (cost=0.15..16.82 rows=5 width=2)
_	5	Index Only Scan using i_sample on sample (cost=0.15..16.82 rows=5 width=2)
v	4	Index Only Scan using i_sample on sample (cost=0.15..14.65 rows=4 width=2)
d	4	Index Only Scan using i_sample on sample (cost=0.15..14.65 rows=4 width=2)
a	3	Index Only Scan using i_sample on sample (cost=0.15..12.49 rows=3 width=2)

l	count	lookup_letter
e	2	Index Only Scan using i_sample on sample (cost=0.15..10.33 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)

```
RESET ALL;

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	345	Seq Scan on sample (cost=0.00..21.16 rows=345 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)
i	1	Index Only Scan using i_sample on sample (cost=0.15..8.17 rows=1 width=2)
k	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: (most important and most popular)
- Merge Join: (for very large tables)

Examples

```
RESET ALL;

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

oid
3
31
33
34
35
38
39
40

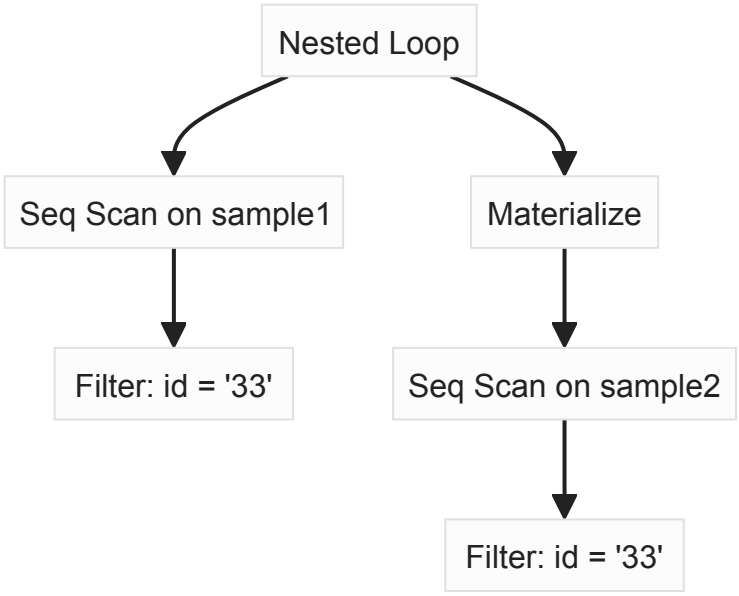
```
CREATE TEMPORARY TABLE sample1 (id, junk) AS
        SELECT oid, repeat('x', 250)
        FROM pg_proc
        ORDER BY random();  -- add rows in random order

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

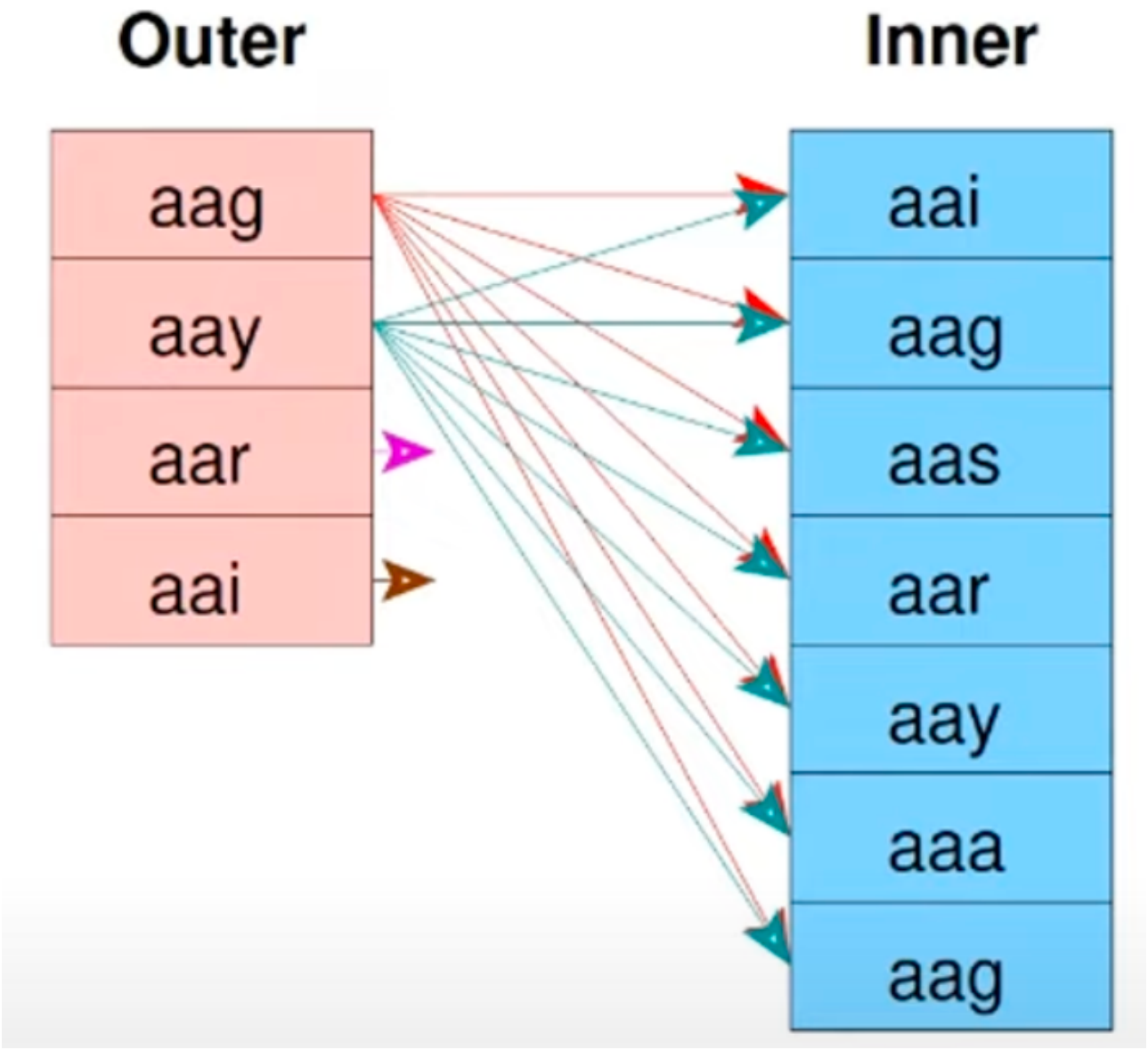
```
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample1.id = 33;
```

```
Nested Loop  (cost=0.00..382.75 rows=810 width=32)
  ->  Seq Scan on sample1  (cost=0.00..331.20 rows=81 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
  - No Setup Requried
  - Used For Small Tables
- Why did the optimizer choose Nested Loop Join with Inner Sequential Scan?
  - reason: we restricted it to only one outer row.

- Outer Table (sample1) only contains 1 row (simple1.id=33)

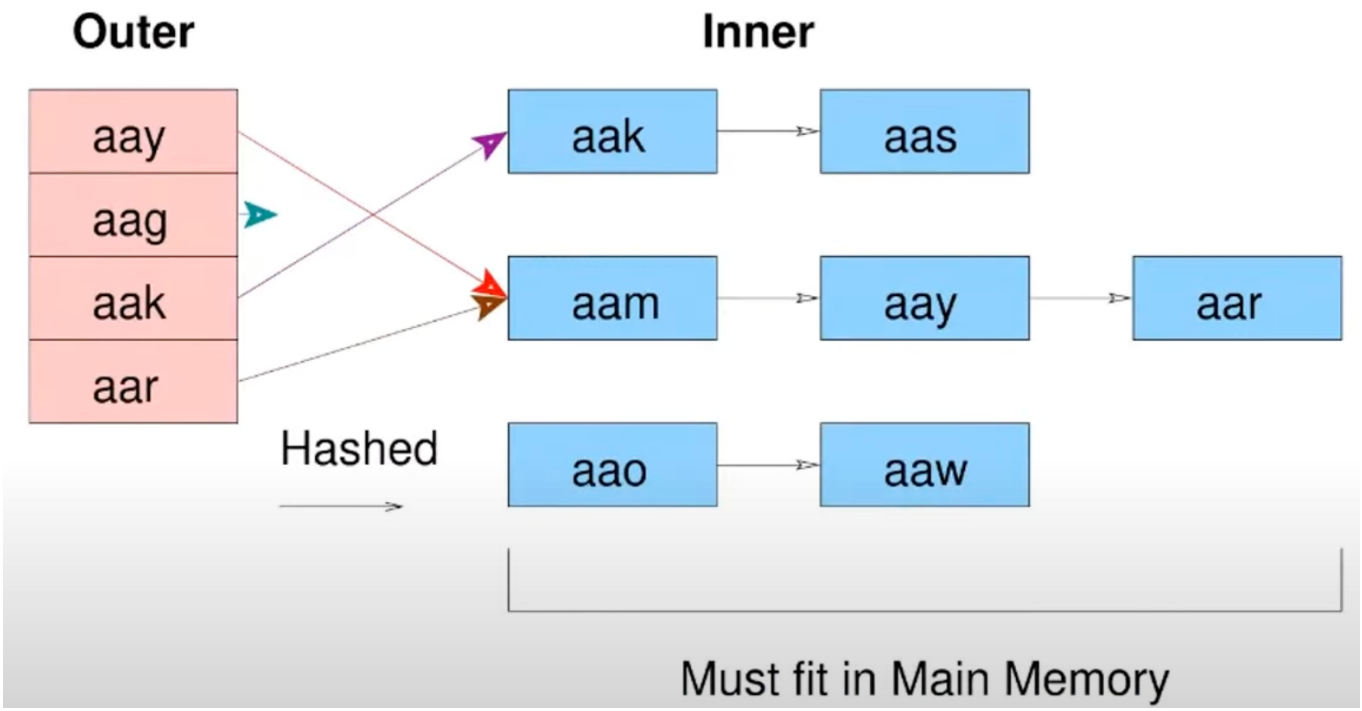


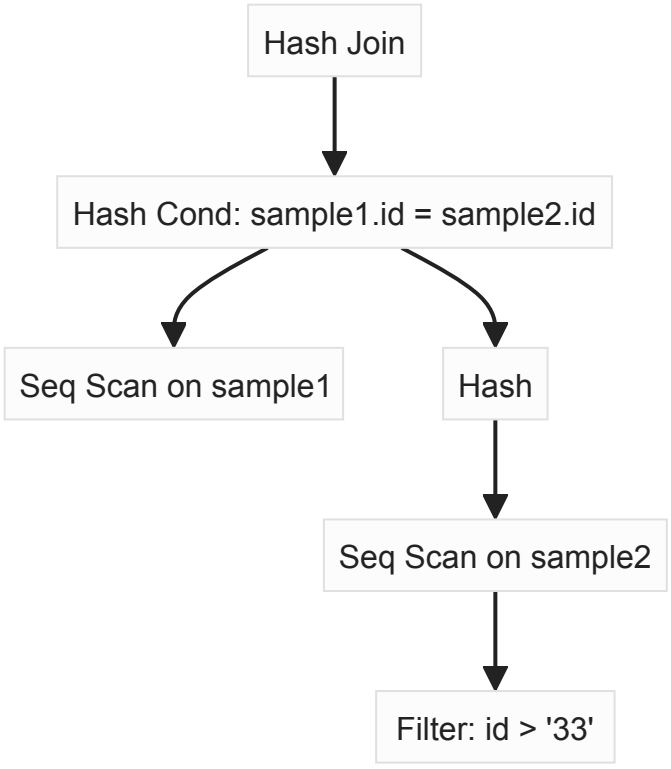

```
// Pseudocode for NLJ with Inner Sequential Scan
for (i = 0; i < length(outer); i++)
        for (j = 0; j < length(inner); j++)
                if (outer[i] == inner[j])
                        output(outer[il, inner [j]) ;
```

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;
```

```
Hash Join  (cost=49.86..2313.09 rows=55027 width=32)
  Hash Cond: (sample1.id = sample2.id)
  ->  Seq Scan on sample1  (cost=0.00..290.56 rows=16256 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)
```





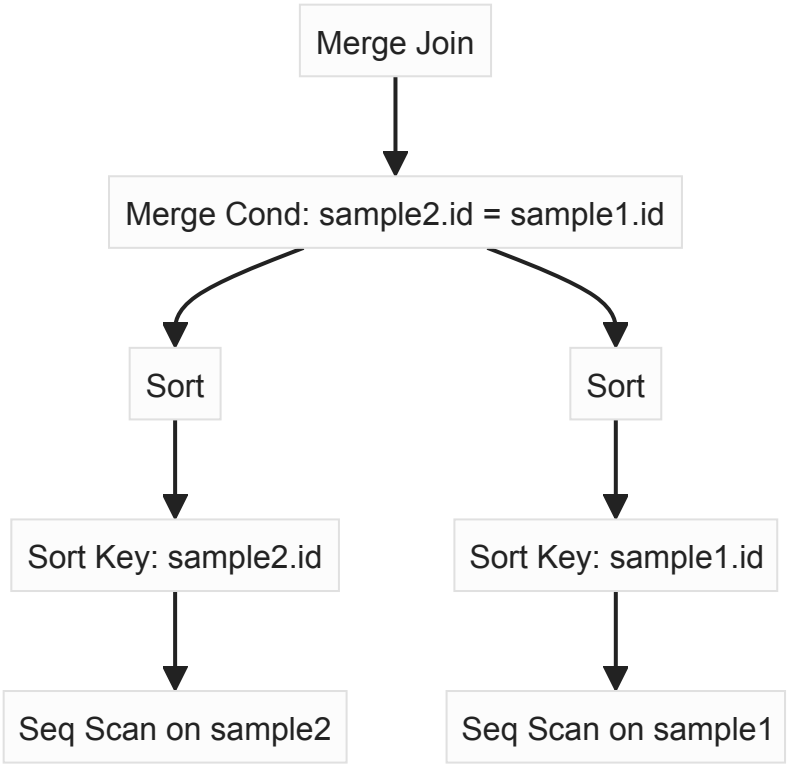
- Hash Join is very good for intermediate types of queries.

```
# First phase: Building the hash table from the inner dataset
for j in range(length(inner)):
    hash_key = hash(inner[j])
    hash_store[hash_key].append(inner[j])

# Second phase: Probing the hash table using the outer dataset
for i in range(length(outer)):
    hash_key = hash(outer[i])
    for j in range(length(hash_store[hash_key])):
        if outer[i] == hash_store[hash_key][j]:
            output(outer[i], hash_store[hash_key][j])
```

Join the Two Tables without Restriction

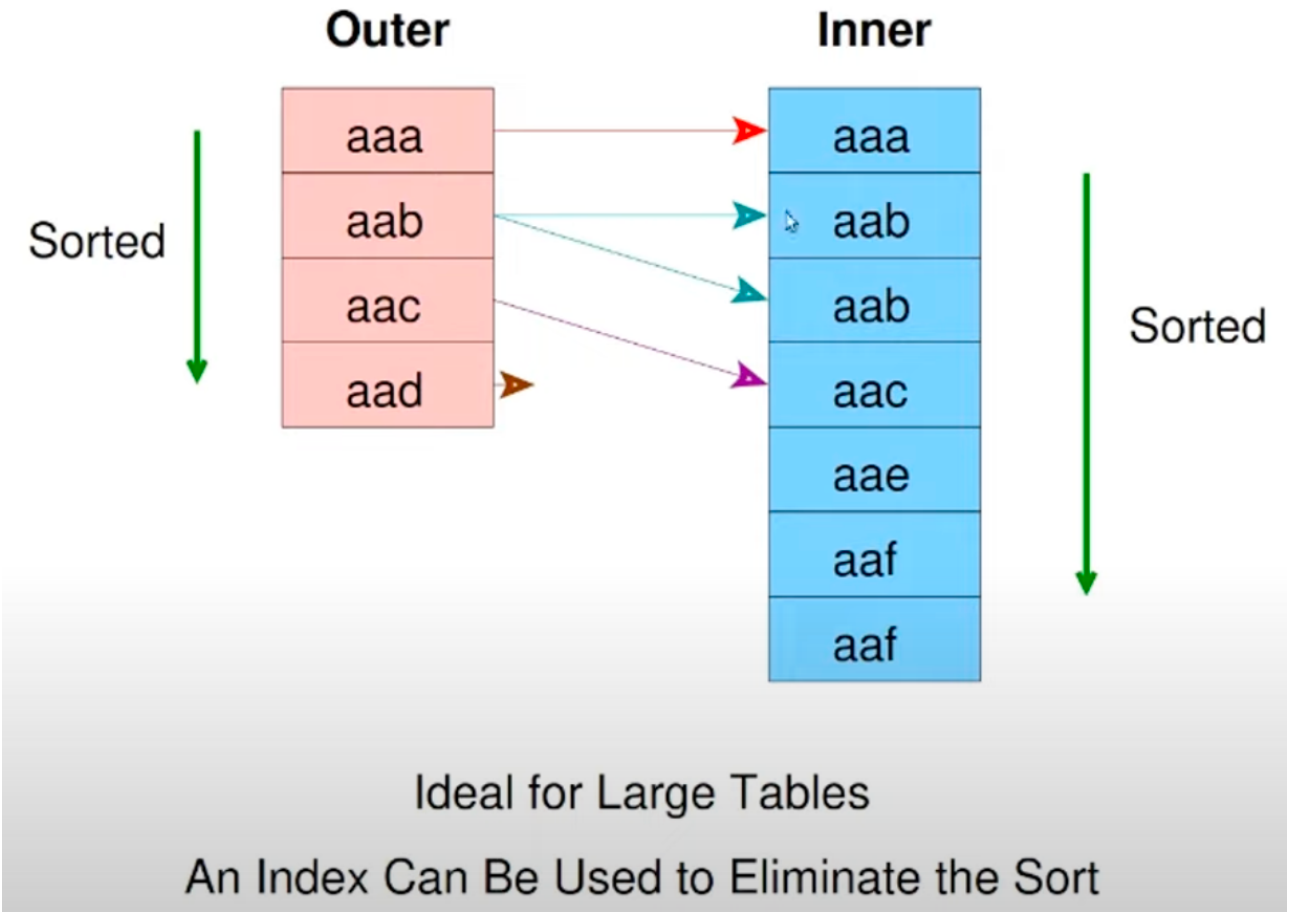
```
EXPLAIN SELECT sample2.junk
FROM sample2 JOIN sample1 ON (sample2.id = sample1.id);
```

```
Merge Join  (cost=1575.53..4063.10 rows=165161 width=32)
  Merge Cond: (sample2.id = sample1.id)
  ->  Sort  (cost=147.97..153.05 rows=2032 width=36)
        Sort Key: sample2.id
        ->  Seq Scan on sample2  (cost=0.00..36.32 rows=2032 width=36)
  ->  Sort  (cost=1427.56..1468.20 rows=16256 width=4)
        Sort Key: sample1.id
        ->  Seq Scan on sample1  (cost=0.00..290.56 rows=16256 width=4)
```



- Merge Join is suitable for large tables

Merge Join



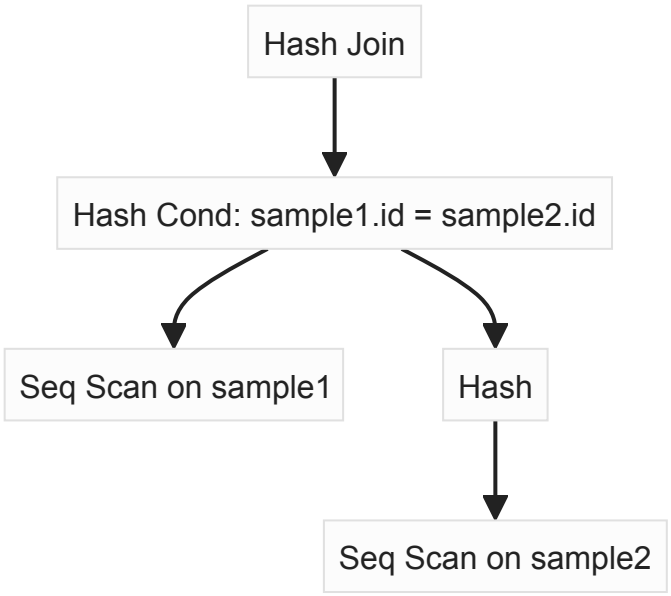

```
sort(outer)
sort(inner)
i = 0
j = 0
save_j = 0

while i < length(outer):
    if outer[i] == inner[j]:
        output(outer[i], inner[j])
    if outer[i] <= inner[j] and j < length(inner):
        j += 1
        if outer[i] < inner[j]:
            save_j = j
    else:
        i += 1
        j = save_j
```

- 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);
```

```
Hash Join  (cost=25.45..203.00 rows=420 width=254)
  Hash Cond: (sample1.id = sample2.id)
  ->  Seq Scan on sample1  (cost=0.00..160.98 rows=3298 width=4)
  ->  Hash  (cost=20.20..20.20 rows=420 width=258)
        ->  Seq Scan on sample2  (cost=0.00..20.20 rows=420 width=258)
```

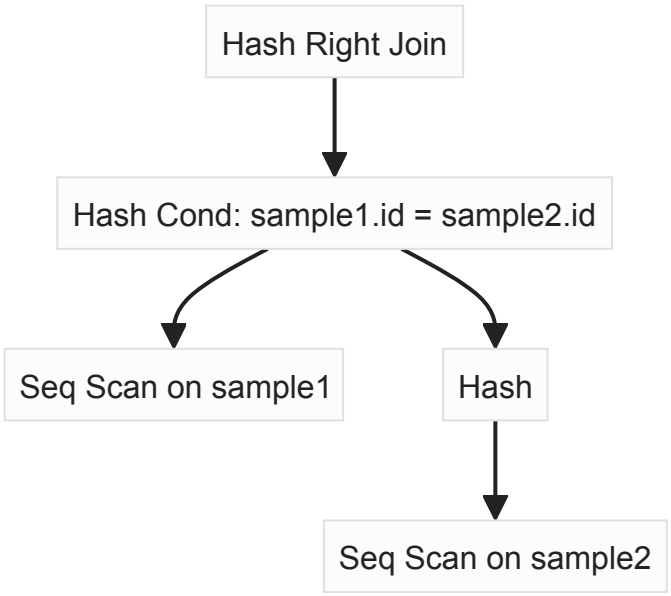


- As the result set is small enough to fit in the memory, the optimizer chooses Hash Join.

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

```
Hash Right Join  (cost=25.45..203.00 rows=420 width=254)
  Hash Cond: (sample1.id = sample2.id)
```

```
  ->  Seq Scan on sample1  (cost=0.00..160.98 rows=3298 width=258)
  ->  Hash  (cost=20.20..20.20 rows=420 width=4)
        ->  Seq Scan on sample2  (cost=0.00..20.20 rows=420 width=4)
```




```
EXPLAIN SELECT sample1.junk
FROM sample1 CROSS JOIN sample2;
```

```
Nested Loop  (cost=0.00..17496.73 rows=1385160 width=254)
  ->  Seq Scan on sample1  (cost=0.00..160.98 rows=3298 width=254)
  ->  Materialize  (cost=0.00..22.30 rows=420 width=0)
        ->  Seq Scan on sample2  (cost=0.00..20.20 rows=420 width=0)
```

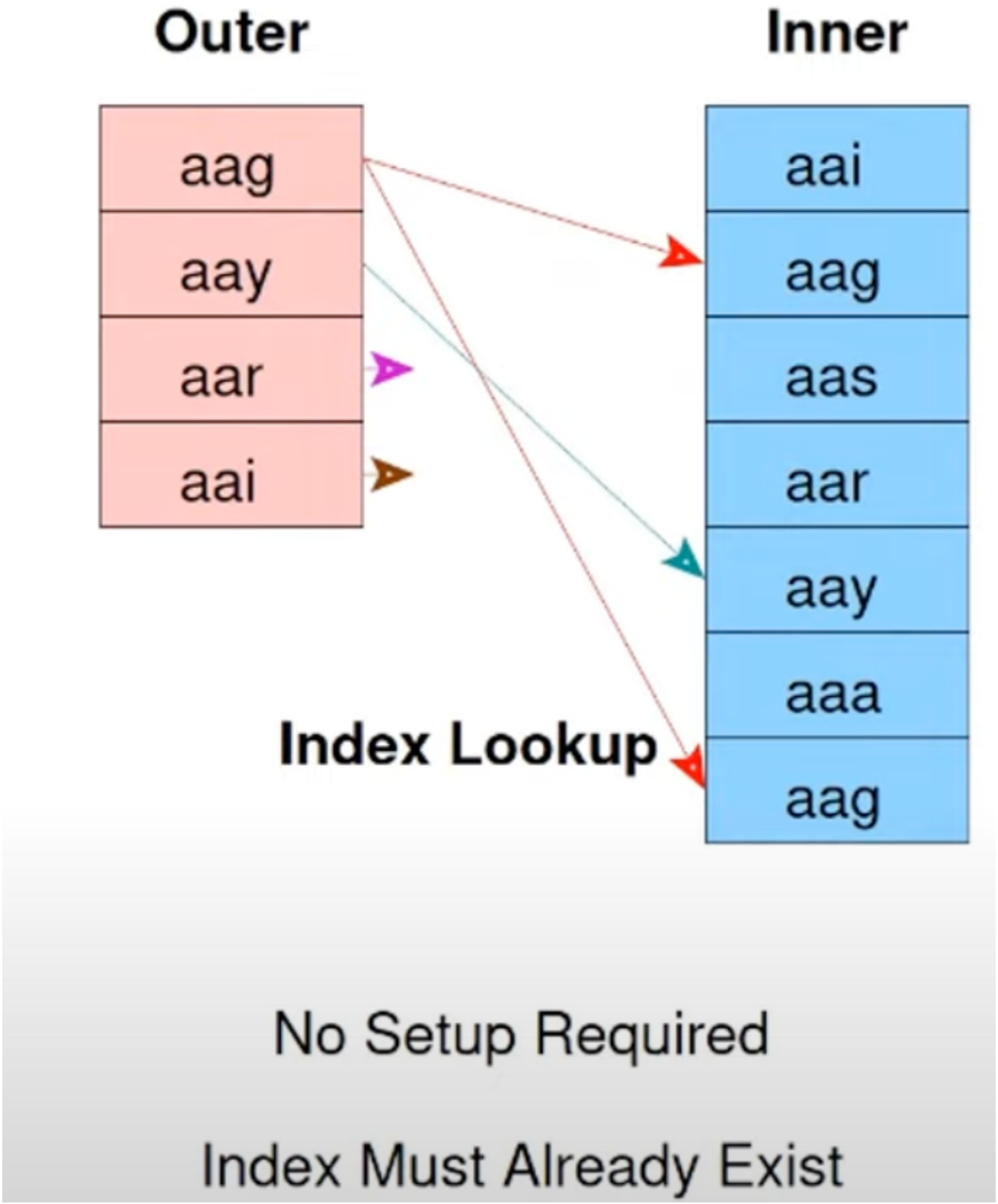
Create Indexes

```
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;
```

```
Nested Loop  (cost=0.55..16.60 rows=1 width=254)
  ->  Index Only Scan using i_sample1 on sample1  (cost=0.28..8.30 rows=1
width=4)
        Index Cond: (id = '33'::oid)
```

```
  ->  Index Scan using i_sample2 on sample2  (cost=0.27..8.29 rows=1
width=258)
        Index Cond: (id = '33'::oid)
```

- NLJ with two sequential scans -> NLJ With two Index Scans
- NLJ with Inner Index Scan



Query Restrictions Affect Join Usage

```
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample2.junk ~ '^aaa';
```

- Although optimizer statistics shows there's no row ~ '^aaa', optimizer assumes there's one row.

```
Nested Loop  (cost=0.28..29.56 rows=1 width=254)
  ->  Seq Scan on sample2  (cost=0.00..21.25 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)
```

```
EXPLAIN SELECT sample2.junk
FROM sample1 JOIN sample2 ON (sample1.id = sample2.id)
WHERE sample2.junk ~ '^xxx';
```

```
Hash Join  (cost=26.50..204.05 rows=420 width=254)
  Hash Cond: (sample1.id = sample2.id)
  ->  Seq Scan on sample1  (cost=0.00..160.98 rows=3298 width=4)
  ->  Hash  (cost=21.25..21.25 rows=420 width=258)
        ->  Seq Scan on sample2  (cost=0.00..21.25 rows=420 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;
```

```
Sort  (cost=221.30..222.35 rows=420 width=254)
  Sort Key: sample2.junk
  ->  Hash Join  (cost=25.45..203.00 rows=420 width=254)
```

```
        Hash Cond: (sample1.id = sample2.id)
        ->  Seq Scan on sample1  (cost=0.00..160.98 rows=3298 width=4)
        ->  Hash  (cost=20.20..20.20 rows=420 width=258)
              ->  Seq Scan on sample2  (cost=0.00..20.20 rows=420 width=258)
```

- With LIMIT, Nested Loop is used

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

```
Limit  (cost=0.55..2.40 rows=1 width=258)
  ->  Nested Loop  (cost=0.55..775.72 rows=420 width=258)
        ->  Index Scan using i_sample2 on sample2  (cost=0.27..86.57
rows=420 width=258)
        ->  Index Only Scan using i_sample1 on sample1  (cost=0.28..1.63
rows=1 width=4)
              Index Cond: (id = sample2.id)
```

What if the optimizer is not sufficient enough.

- **Extended statistics** allow you to create multi-column statistics for a single table. ext
  - Learn how to create extended statistics for column combinations that you actually are accessing in existing queries.
  - see `create_statistics` command
- Increasing the statistics target like the most common value

Q&A

Join Statistics

- Q: How does Postgres generate estimates for rows resulting from joins? Is there a way we can influence this in Postgres?
  - A: extended statistics doesn't really help with cross-table statistics. Join statistics are still underdeveloped.
  - Occasionally, you can use common table expressions with a materialized option. If you use MATERIALIZED keyword, you can pull the part that you want to run

separately into a separate query. Break apart a query that might have had 10 joins into two queries have 5 joins.

Query Plan hints

- Short answer: Postgres doesn't have query plan hints because of the consideration for expansibility.
- Using query plan hints to force a query to be executed in a certain way has a tendency to get baked into the system. However, the problem is that the distribution of the data changes, the amount of data changes, the optimizer gets smarter, those manual query plans may end up hurting your performance.