Multidimensional search strategies for composite B-Tree indexes

Peter Geoghegan

geogpete@amazon.com
pg@bowt.ie

Overview

1. **Skip scan: Overview**
   Summary of the feature, with real examples

2. **Skip Scan runtime cost profile and the optimizer**
   Adaptive/dynamic design philosophy

3. **Postgres 18 OR transformation work**
   Summary of related/enabling optimizer work

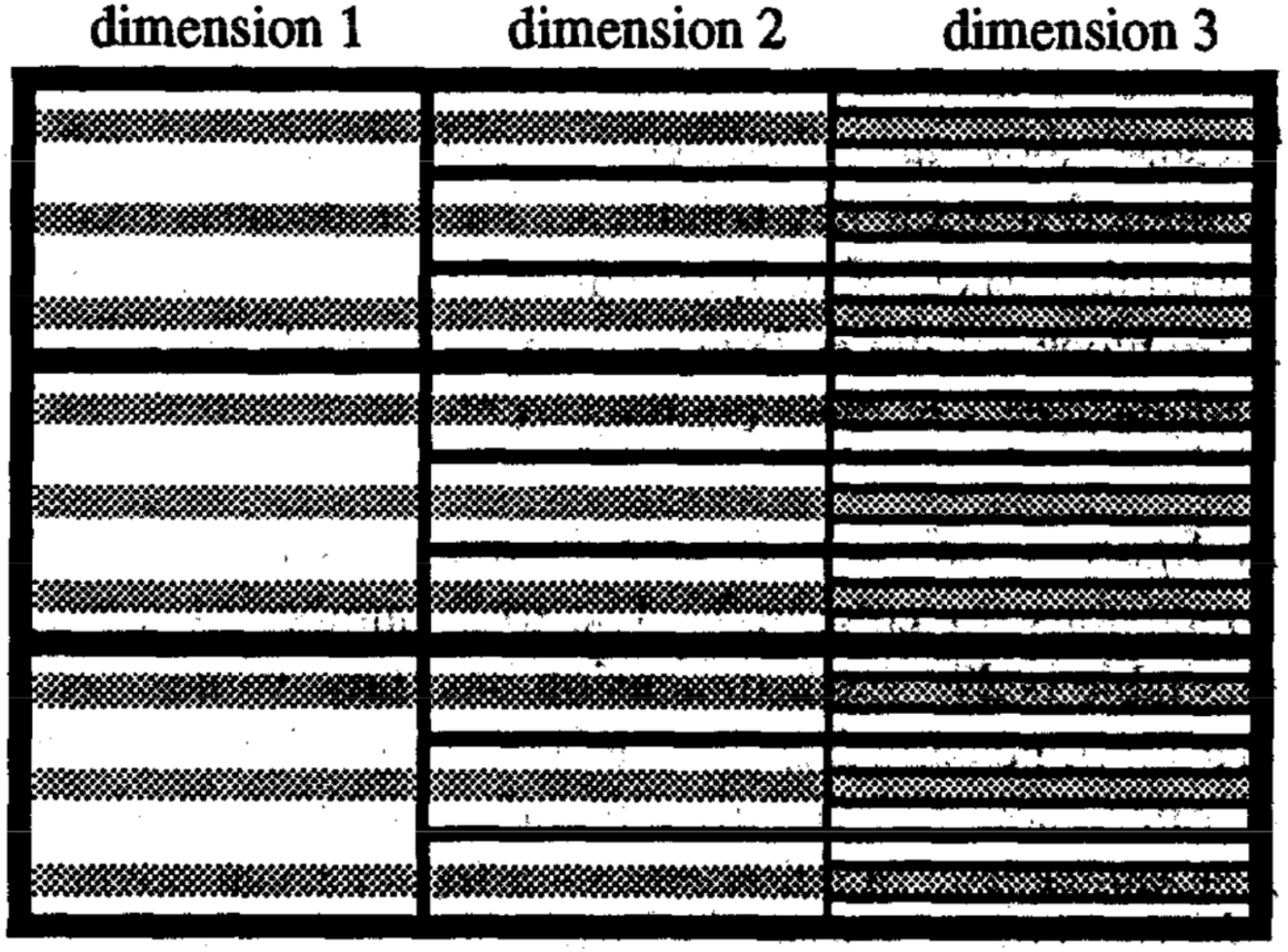
4. **MDAM style "general OR optimization"**
   Possible areas for future improvements

Skip Scan: Overview

The skip scan optimization allows the system to make more effective use of existing multicolumn B-Tree indexes in certain contexts

- Used when a prefix of one or more columns has an **"=" condition omitted** in SQL statement's predicate
- Treats the index as a **"series of logical subindexes"** (one subindex per distinct value in skipped prefix column)
- Most effective when skipped column has few distinct values (i.e. when there are few "logical subindexes" to consider)
- nbtree implementation influenced by 1995 paper "Efficient Search of Multidimensional B-Trees" (the "MDAM paper")

Figure 2 shows the same B_Tree, but shows what is retrieved when there is only a predicate on the third column (dimension_3 = 2).



Only need to read "`dimension_3 = 2`" tuples in shaded areas (irrelevant unshaded areas will be "skipped over" instead)

Brief PSA: Skip scan != Loose index scan

Skip scan has often been confused with "loose index scan", a feature that MySQL has had for some time (MySQL's skip scan feature was aded much later, in 2018)

- I named this feature skip scan because that's the name used by MySQL, Oracle, and SQLite for their versions of the same feature

- Loose index scan is **more specialized** than skip scan: can only be used with `GROUP BY` queries/`DISTINCT` queries

- Loose index scan "returns groupings" rather than returning tuples, and saves on both index accesses **and heap accesses**

- Superficial similarity (the way that both techniques "skip") seems to throw people off

Skip Scan: Benefits for users

- Gives users acceptable performance with seldom-run queries that might not merit a "dedicated" index

  - Obviously, users *always* pay to store and maintain an index, but *only get a benefit* when the index is **actually scanned**

  - With modern hardware, an index scan that performs tens of thousands of index searches can complete in under 200 milliseconds with a well cached index

- Provides more robust performance, especially when requirements change at short notice

Skip Scan: Benefits for users (cont.)

- Simplifies the "put column for highly selectivity qual first" versus "make index's column order match `ORDER BY`" trade-off. This sometimes comes up when **choosing the ideal index** for an important "`ORDER BY ... LIMIT N`" query

  - Do we prioritize avoiding a sort/terminating the scan early for our "`ORDER BY ... LIMIT N`" query? Should `CREATE INDEX` column order match the query's ORDER BY columns?

  - **OR**, do we prefer to put the most selective index column first (typically a column involving "=" condition) in `CREATE INDEX`, at the cost of having to always read the whole result set and sort it each time?

  - **Worst case** matters a lot. Is `LIMIT 1` or `LIMIT 100` more typical? Will the query sometimes return **0 rows**?

- Skip scan makes it safer to prioritize avoiding a sort/avoiding reading all matching data in the index (i.e. makes it's safer to favor `LIMIT N` ending the scan early)

  - Typical case where it might help involves a "date between x and y" condition, plus some additional selective "=" condition on another column

```
-- Create and load example table, "tab":
create table tab (a int4, b int4);
CREATE TABLE
create index multicol on tab(a, b);
CREATE INDEX

-- 10 distinct values in "a", 50k distinct "b" values:
insert into tab (a, b) select i, j from generate_series(1, 10) i,
generate_series(1, 50_000) j;
INSERT 0 500000
```

```
-- Query (uses skip scan):
select * from tab
where b = 5_000;

-- Skip scan uses a ScalarArrayOp-like array internally:
select * from tab
where a = ANY('{every possible value}') -- does "IS NULL" matching
and b = 5_000;
```

```
-- Index-only scan, applies skip scan optimization:
=# explain (analyze, costs off, timing off)
   select * from tab
   where b = 5_000;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Only Scan using multicol on tab (rows=10.00 loops=1)
   Index Cond: (b = 5000)
   Heap Fetches: 0
   Index Searches: 12
   Buffers: shared hit=37
 Planning Time: 0.017 ms
 Execution Time: 0.023 ms
(7 rows)
```

Skip Scan and ScalarArrayOps

Postgres 18 adds nbtree skip scan, which builds directly on work on nbtree ScalarArrayOp index quals from Postgres 17

- ScalarArrayOpExprs (AKA SAOPs) are how the system represents things like "`where a in (1, 2, 3)`" and "`where a = ANY('{1, 2, 3}')`"

- Postgres 17 work made nbtree scans navigate the index dynamically, based on **physical index characteristics**
  - Typically, "`where a in (1, 2, 3)`" uses only 1 index search in Postgres 17 and 18 -- earlier versions **always** used 3

  - "`where a in (10_000, 20_000, 30_000)`" likely *will* still perform 3 index searches (as with prior Postgres versions), since that's *probably* still the fastest approach

  - When and where we skip (i.e. the number of index searches) is **determined dynamically**

- Postgres 18 **reuses** this infrastructure for skip scan

```
-- "Equivalent" SAOP index-only scan:
=# explain (analyze, costs off, timing off)
   select * from tab
   where a in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) and b = 5_000;
                                      QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using multicol on tab (rows=10.00 loops=1)
   Index Cond: ((a = ANY ('{1,2,3,4,5,6,7,8,9,10}'::integer[])) AND (b = 5000))
   Heap Fetches: 0
   Index Searches: 10
   Buffers: shared hit=31
 Planning Time: 0.028 ms
 Execution Time: 0.022 ms
(7 rows)
```

```
-- Index-only scan, applies "range" skip array:
=# explain (analyze, costs off, timing off)
   select * from tab
   where a between 1 and 10 and b = 5_000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using multicol on tab (rows=10.00 loops=1)
   Index Cond: ((a >= 1) AND (a <= 10) AND (b = 5000))
   Heap Fetches: 0
   Index Searches: 10
   Buffers: shared hit=31
 Planning Time: 0.025 ms
 Execution Time: 0.022 ms
(7 rows)
```

SAOP arrays and skip arrays

Arrays (whether SAOP or skip type arrays) advance in lockstep with the scan's progress through the index's key space

- Find the best match for a given attribute value from an index tuple, using binary search of attribute's array

- Advances to next closest array element ("next" in terms of scan direction) when no exact match exists

- Lower-order arrays "roll over" to higher-order arrays when there's no exact match **and** no next closest array match remains (i.e. when we "reach the end of the array")

  - When this happens, the array is reset to its first element, and the next most significant array must increment its array element in turn

  - "Skip support" is type-aware/opclass infrastructure, that "increments" skip arrays (e.g., when we've reached the end of "a = 5" matches, increments to "a = 6")

  - When the **most significant** array "tries to roll over", we just **end the top-level scan** (all tuples matching any possible set of array elements were already returned)

```
-- Uses "mdam_idx" on (dept, sdate, item_class, store) columns:
=# explain (analyze, costs off, timing off)
select
  dept,
  sdate,
  item_class,
  store,
  sum(total_sales)
from
  sales_mdam_paper
where
  -- "dept" column omitted from qual
  sdate between '1995-06-01' and '1995-06-30'
  and item_class in (20, 35, 50)
  and store in (200, 250)
group by dept, sdate, item_class, store
order by dept, sdate, item_class, store;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------
 GroupAggregate (actual rows=18000.00 loops=1)
   Group Key: dept, sdate, item_class, store
   Buffers: shared hit=54014
   ->  Index Scan using mdam_idx on sales_mdam_paper (actual rows=18000.00 loops=1)
         Index Cond: ((sdate >= '1995-06-01'::date) AND (sdate <= '1995-06-30'::date) AND ...
         Index Searches: 9002
         Buffers: shared hit=54014
 Planning:
   Buffers: shared hit=133
 Planning Time: 0.550 ms
 Execution Time: 45.910 ms
(11 rows)
```

```
dept=-∞, date='1995-06-01', item_class=20, store=200
 dept=1, date='1995-06-01', item_class=20, store=200
 dept=1, date='1995-06-01', item_class=20, store=250
 dept=1, date='1995-06-01', item_class=35, store=200
 dept=1, date='1995-06-01', item_class=35, store=250

... (omitted: 8994 similar accesses)...

dept=100, date='1995-06-30', item_class=50, store=200
dept=100, date='1995-06-30', item_class=50, store=250
dept=101, date='1995-06-01', item_class=20, store=200
```

- `Index Searches: 9002`

  - 100 departments × 30 days × 3 item classes × 2 stores = **18,000 rows returned**

  - 9,000 index searches return 2 rows due to **physical index characteristics**: each pair of "store = 200" and "store = 250" tuples appear close together, on the same index leaf page

  - Plus 2 "extra" searches for non-existent "dept=-∞" and "dept=101" entries

Overview

1. **Skip scan: Overview**
   Summary of the feature, with real examples

2. **Skip Scan runtime cost profile and the optimizer**
   Adaptive/dynamic design philosophy

3. **Postgres 18 OR transformation work**
   Summary of related/enabling optimizer work

4. **MDAM style "general OR optimization"**
   Possible areas for future improvements

Architectural goals

- Accurately modeling the costs and benefits of skipping is hard in general

- No **new optimizer paths** that have to **compete** with traditional full index scan paths

  - Optimizer generates the same index paths as before (though `btcostestimate()` accounts for skipping)

  - If there's **only one choice**, there are no wrong choices

  - Make all decisions about skipping **at runtime**

Architectural goals (cont.)

In order to reuse existing/standard optimizer index paths, skip scan has to work alongside all existing functionality

- As discussed already, SAOP array/IN() list quals can be used freely
- Mark/restore (for scans used by a merge join) works
- All index scan optimizer paths generate useful path keys
  - Useful with "`ORDER BY ... LIMIT`", etc
  - Supports backwards scans/`ORDER BY` with `DESC` columns
  - Scrollable cursors work (can scan back and forth)
- *All* of these requirements were satisfied by reusing Postgres 17 SAOP mechanisms (**no new code needed** in Postgres 18)

Dynamic/adaptive scans

Making life easier for the optimizer makes life harder for the executor/nbtree scan code

- Occasionally, the fastest plan really does need to perform a traditional full index scan

  - Typically an index-only scan

  - Skipping cannot help, but *considering* skipping **shouldn't unduly slow down** these scans

- nbtree has various runtime strategies that help

  - Also helps with individual subsets of an index that naturally "require a full index scan" due to data skew

Perhaps the **most complex aspect is the cost analysis** required for a cost-based compile-time decision between a full index scan, range scan, and selective probes.

A **dynamic run-time strategy** might be most efficient and **robust** against cardinality estimation errors, cost estimation errors, data skew, etc.

— **Goetz Graefe, Modern B-Tree Techniques**

Dynamic/adaptive scans (cont.)

Skip arrays "anchor" the scan to index's key space via the scan's arrays/current set of array elements (when no conventional "=" constraint can be taken from the query)

- This enables "skipping within a page"
  - Postgres 17 SAOP patch added this mechanism: the "look-ahead" optimization
  - Used by earlier MDAM sales example query (each pair of "store = 200" and "store = 250" tuples not *that* close together)
  - Also helps with scans that only perform "moderate skipping"
- Skip arrays also enable optimizations that avoid evaluating scan keys that were proven to be **guaranteed** to be satisfied by **every possible tuple on a page** via an up-front check of the page's low and high tuples
  - First implemented in Postgres 17, by work from Alexander Korotkov
  - Postgres 18 much more effective here, particularly with complicated quals; new improved mechanism is "array aware"

“The number of distinct values is not the true criterion, however. The alternative query execution plan typically is to scan the entire index with large sequential I/O operations.

The probing plan is faster than the scanning plan if the data volume for each distinct leading B-tree key value is so large that a scan takes longer than a single probe. Note that this efficiency comparison must include not only I/O but also **the effort for predicate evaluation**.”

— **Goetz Graefe, Modern B-Tree Techniques**

```
=# explain (analyze, costs off, timing off)
   select a, b
   from skiptest
   where a between 0 and 10_000_000 and b = 50;

                                   QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using skiptest_a_b_idx on skiptest (actual rows=0.00 loops=1)
   Index Cond: ((a >= 0) AND (a <= 10000000) AND (b = 50))
   Heap Fetches: 0
   Index Searches: 1
   Buffers: shared hit=27325
 Planning:
   Buffers: shared hit=9
 Planning Time: 0.071 ms
 Execution Time: 147.001 ms
(9 rows)
```

- `Index Searches: 1`
  - No chance of "skipping" here, since there are as many distinct "a" values as there are tuples read
  - Skip arrays nevertheless make query execution much faster
  - `_bt_readpage` function determines inequality keys on "a" must already be satisfied

Adapting to real world data distributions

So far, all of our examples have used synthetic data with uniform random distribution

- Uniform data helpful when explaining underlying concepts
- **Real world data** often has some kind of skew, though
  - A few **"heavy hitters"** dominate, with a **long tail** of almost-unique values
  - Also contributes to difficulties with cost estimation
- Legitimately need to vary our strategy *during the same index scan*
  - An individual scan may apply either optimization, as data skew/ physical index characteristics dictate

Overview

1. **Skip scan: Overview**
   Summary of the feature, with real examples

2. **Skip Scan runtime cost profile and the optimizer**
   Adaptive/dynamic design philosophy

3. **Postgres 18 OR transformation work**
   Summary of related/enabling optimizer work

4. **MDAM style "general OR optimization"**
   Possible areas for future improvements

Postgres 18 OR transformation

- Work in Postgres 18 from Alena Rybakina, Alexander Korotkov, Andrei Lepikhov, and Pavel Borisov complements recent nbtree improvements

- Transforms OR lists into array/ScalarArrayOp representation that can be passed down to index scans

  - OR lists are semantically equivalent to IN() constructs (per SQL standard)

  - Avoids BitmapOr plans, which are sometimes *much* slower

- Relevant Postgres 18 commits:

  - "Transform OR-clauses to SAOP's during index matching" commit

  - "Allow usage of match_orclause_to_indexcol() for joins" commit

  - "Convert 'x IN (VALUES ...)' to 'x = ANY ...' when appropriate" commit

```
diff --git a/src/test/regress/expected/create_index.out b/src/test/regress/expected/create_index.out
index d3358dfc3..e4d117e47 100644
--- a/src/test/regress/expected/create_index.out
+++ b/src/test/regress/expected/create_index.out
@@ -1844,18 +1844,11 @@ DROP TABLE onek_with_null;
 EXPLAIN (COSTS OFF)
 SELECT * FROM tenk1
   WHERE thousand = 42 AND (tenthous = 1 OR tenthous = 3 OR tenthous = 42);
-                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
-
- Bitmap Heap Scan on tenk1
-   Recheck Cond: (((thousand = 42) AND (tenthous = 1)) OR ((thousand = 42) AND (tenthous = 3)) OR ....
-   ->  BitmapOr
-         ->  Bitmap Index Scan on tenk1_thous_tenthous
-               Index Cond: ((thousand = 42) AND (tenthous = 1))
-         ->  Bitmap Index Scan on tenk1_thous_tenthous
-               Index Cond: ((thousand = 42) AND (tenthous = 3))
-         ->  Bitmap Index Scan on tenk1_thous_tenthous
-               Index Cond: ((thousand = 42) AND (tenthous = 42))
-(9 rows)
+                                      QUERY PLAN
+---------------------------------------------------------------------------------------
+ Index Scan using tenk1_thous_tenthous on tenk1
+   Index Cond: ((thousand = 42) AND (tenthous = ANY ('{1,3,42}'::integer[])))
+(2 rows)
```

OR transformation: goals

- Executor overhead adds up with BitmapOr type plans
  - One index scan is faster than many, due to per-node executor costs, which add up with more complicated queries
- BitmapOr + Bitmap index scan approach is very **general**, but comes with **notable downsides** compared to an approach that uses a single scan for everything
- Perhaps most useful as an "enabling transformation" - enables the use of the SAOP nbtree index scan mechanism from Postgres 17
  - Useful sort order/path keys can be used
  - Enables all the usual tricks, such as allowing the scan to terminate early with an "`ORDER BY ... LIMIT`" query
  - Enables index-only scans

```
-- Add more "tenthous" values, add a LIMIT 2:
=# explain (analyze, costs off, timing off)
select * from tenk1
   where thousand = 42 AND
   (tenthous = 42 or tenthous = 1042 or tenthous = 2042 or tenthous = 3042)
   limit 2;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Limit (actual rows=2.00 loops=1)
   Buffers: shared hit=4
   ->  Index Scan using tenk1_thous_tenthous on tenk1 (actual rows=2.00 loops=1)
         Index Cond: ((thousand = 42) AND (tenthous = ANY ('{42,1042,2042,3042}'::integer[])))
         Index Searches: 1
         Buffers: shared hit=4
 Planning Time: 0.053 ms
 Execution Time: 0.018 ms
(8 rows)
```

```
-- Add more "tenthous" values, remove "LIMIT 2", select only indexed columns:
=# explain (analyze, costs off, timing off)
select thousand, tenthous from tenk1
   where thousand = 42 AND
   (tenthous = 42 or tenthous = 1042 or tenthous = 2042 or tenthous = 3042);
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using tenk1_thous_tenthous on tenk1 (actual rows=4.00 loops=1)
   Index Cond: ((thousand = 42) AND (tenthous = ANY ('{42,1042,2042,3042}'::integer[])))
   Heap Fetches: 0
   Index Searches: 1
   Buffers: shared hit=3
 Planning Time: 0.049 ms
 Execution Time: 0.014 ms
(7 rows)
```

Overview

1. **Skip scan: Overview**
   Summary of the feature, with real examples

2. **Skip Scan runtime cost profile and the optimizer**
   Adaptive/dynamic design philosophy

3. **Postgres 18 OR transformation work**
   Summary of related/enabling optimizer work

4. **MDAM style "general OR optimization"**
   Possible areas for future improvements

Improving OR optimization

- Optimizer work added to Postgres 18 (discussed in last section) only applicable to a few important cases involving OR clauses

- Can we **generalize this idea**, to make it work with more complicate OR constructs?
  - MDAM paper describes more sophisticated OR transformations/optimizations
  - This is outside my area of expertise. **Help wanted!**

Advanced OR optimization

- MDAM paper's final section, "General OR optimization", describes how this is possible

  - Duplicate elimination is a big problem with OR optimization in general (e.g., with unrelated optimization that converts a join into a `UNION`, to speed up star schema queries)

  - MDAM performs duplicate elimination **"before any data is read"** via analysis, as opposed to *actually* eliminating duplicates (e.g., by using a TID bitmap, or by eliminating duplicate TIDs)

  - As with simple skip scan/OR transformation, reduces everything to a series of disjoint "single value" accesses in **index key space order**, which behave like **one continuous index scan**

  - Unlike skip scan, each access can use different operators, etc

- **Picture in Postgres 18** with more complicated ORs is **mixed**, though we're generally still forced to use BitmapOr plans...

```
=# explain (analyze, costs off, timing off)
select *
from
  sales_mdam_paper
where ((dept >= 1 and dept <= 3) or (dept > 4 and dept <= 8))
and sdate in ('1995-02-01', '1995-02-03')
and item_class = 5;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on sales_mdam_paper (actual rows=4200.00 loops=1)
   Recheck Cond: ...
   Filter: ...
   Heap Blocks: exact=617
   Buffers: shared hit=699
   ->  BitmapOr ...
         Buffers: shared hit=82 -- total # of index buffer hits
         ->  Bitmap Index Scan on mdam_idx (actual rows=1800.00 loops=1)
               Index Cond: ((dept >= 1) AND (dept <= 3) AND (sdate = ANY (...) AND (item_class
               Index Searches: 6
               Buffers: shared hit=35
         ->  Bitmap Index Scan on mdam_idx (actual rows=2400.00 loops=1)
               Index Cond: ((dept > 4) AND (dept <= 8) AND (sdate = ANY (...) AND (item_class =
               Index Searches: 8
               Buffers: shared hit=47
 Planning Time: 0.064 ms
 Execution Time: 1.054 ms
(17 rows)
```

Observations on BitmapOr Postgres 18 plan

- Some things work well here already!

  - Planner can push down SAOP qual, as well as non-array scalar = condition as index quals

  - Each individual index scan does "range skip scan"

  - No (or minimal) repeat reads of index leaf pages here

- But (in this example) we spend more than twice as much time on heap access

  - In Postgres 18, the problem is no longer the cost of scanning the index

  - **617 heap buffer hits** vs. only **82 index buffer hits**

- Trick here is to **get an Index-only scan** that offers the **best of both worlds**

  - Earlier we saw an example where this happened, involving a simple OR list/clause

  - Postgres 18 *can* already do all this with similar "`dept between 1 and 8 and ...`" query, but this query isn't supposed to return "`dept = 4`" rows

```
=# explain (analyze, costs off, timing off)
select dept, sdate, store, item_class
from
  sales_mdam_paper
where ((dept >= 1 and dept <= 3) or (dept > 4 and dept <= 8))
and sdate in ('1995-02-01', '1995-02-03')
and item_class = 5;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Only Scan using mdam_idx on sales_mdam_paper (actual rows=4200.00 loops=1)
   Index Cond: ((sdate = ANY ('{1995-02-01,1995-02-03}'::date[])) AND (item_class = 5))
   Filter: (((dept >= 1) AND (dept <= 3)) OR ((dept > 4) AND (dept <= 8)))
   Rows Removed by Filter: 55800
   Heap Fetches: 0
   Index Searches: 202
   Buffers: shared hit=1408
 Planning Time: 0.051 ms
 Execution Time: 6.718 ms
(9 rows)
```

Observations on "OR" Index-only scan Postgres 18 plan

- Index-only scan eliminates heap buffer hits, but is still **significantly slower**!

  - Plan uses **filter quals**, leading to much less efficient skip scan/index navigation due to "`dept`" column's inequalities not being used as index quals

  - Using "`dept between 1 and 8 and dept != 4 and ...`" spelling of the query has similar problems, also involving filter quals

- Costing is inaccurate, which certainly doesn't help

  - BitmapOr plan's total cost: **8059.63**

  - Index-only scan plan's total cost: **2676.37**

  - In reality, the BitmapOr scan plan is **almost 7x faster**

- Postgres 18 effectively imposes a **false choice** between minimizing heap accesses and efficient index scans -- MDAM paper gives us a way forward

General OR Optimization

One of the most powerful aspects of MDAM is that it supports predicates with any combination of ORs and ANDs. This is accomplished by associating the predicates with different predicate sets in a variant of disjunctive normal form. IN lists are treated as a group. Therefore, the resulting predicate sets are not truly in disjunctive normal form. We will call each predicate set a disjunct. So let us take the following complex expression as an example:

```
((item_class=10 and date between "06/04/95" and
  06/25/95)  OR dept IN (2, 4, 5))
and
((dept=4 and item_class=5) OR
  (item_class IN (5,10) and (date="06/04/95" OR
                             dept=2)))
```

Retrieval	Dept	Date	Item_Class
1	<2	="06/04/95"	=10
2	=2	<"06/04/95"	=5 or =10
3	=2	>="06/04/95" & <="06/25/95"	=10
4	=2	>"06/25/95"	=5 or =10
5	>2 & <4	="06/04/95"	=10
6	=4	<"06/04/95"	=5
7	=4	="06/04/95"	=5 or =10
8	=4	>"06/04/95"	=5
9	=5	="06/04/95"	=5
10	=5	="06/04/95"	=10
11	>5	="06/04/95"	=10

Table 3

Implementing MDAM style "general OR optimization"

- Currently, optimizer cannot perform OR transformation outside of the confines of BitmapOr (barring simple OR list transformation case)

  - My example query's Bitmap index scans perform **disjoint** accesses, but the planner isn't aware of that

  - My query deliberately made things easy, but it wouldn't be quite so easy if (say) the pair of OR'd "`dept`" ranges **overlapped**

- As we saw, MDAM OR optimization handles these not-so-easy cases

  - Need to ensure that **no duplicates** can ever be returned

  - We'll need to do the same thing to implement OR optimization; otherwise, it doesn't seem of much practical use to real world queries

-

Conclusions

Skip Scan works by treating a composite index as a multidimensional structure

Can be combined with ScalarArrayOp index conditions generated from "= ANY(...)" and "IN(...)" constructs

OR transformation is therefore more important than ever

These techniques reduce index scan costs directly, and sometimes indirectly enable query plans that perform fewer heap accesses by allowing a scan to end early, or by avoiding use of filter quals

More advanced "MDAM style" OR transformations are feasible, and are enabled by skip scan