DB 2

10 – Matching Queries and Indexes

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 Matching Queries and Indexes

The mere presence of an index I on column $T(a)$ neither guarantees nor warrants the use of I during query evaluation. The **index has to match the query:** can I help to (significantly) reduce evaluation cost?

- Does I index the column(s) referenced in this WHERE predicate p?
- Are *all columns* referenced by this query present in I?
- Does the order of rows in I's sequence set match the row order required by this ORDER BY/GROUP BY clause?
- Accessing I will cause I/O. Do we still save overall I/O because we need to access less pages of T's heap file?

Recall table indexed and its two indexes:

```
CREATE TABLE indexed (a int PRIMARY KEY,
                      b text,
                      c numeric(3,2));
CREATE INDEX indexed_a ON indexed USING btree (a); --
CREATE INDEX indexed_c ON indexed USING btree (c);
```

- Will the following query be supported by an index?

```
SELECT i.a
FROM   indexed AS i
WHERE  degrees(asin(i.c)) = 90 -- recall: i.c ≡ sin(i.a)
```

In the absence of an function-based index, the query will be evaluated by a Seq Scan:

```
DROP TABLE IF EXISTS indexed;
CREATE TABLE indexed (a int PRIMARY KEY,
                      b text,
                      c numeric(3,2));
ALTER INDEX indexed_pkey RENAME TO indexed_a;
CREATE INDEX indexed_c ON indexed USING btree (c);

INSERT INTO indexed(a,b,c)
        SELECT i, md5(i::text), sin(i)
        FROM   generate_series(1,1000000) AS i;

ANALYZE indexed;

EXPLAIN VERBOSE
  SELECT i.a
  FROM   indexed AS i
  WHERE  degrees(asin(i.c)) = 90;
```

QUERY PLAN

Seq Scan on public.indexed i (cost=0.00..29346.44 rows=5000 width=4)
  Output: a
  Filter: (degrees(asin((i.c)::double precision)) = '90'::double precision)

Seq Scan, index unused :-(

Index indexed_c has been built on column c — only selections on c will be supported ⇒ try to rewrite the predicate such that i.c is isolated:

- degrees(x) = y ⇔ x = (y / 180.0) * π
- asin(x) = y ⇔ x = sin(y)

```
EXPLAIN VERBOSE
  SELECT i.a
  FROM   indexed AS i
  WHERE  i.c = sin((90 / 180.0) * pi());
```

QUERY PLAN

Seq Scan on public.indexed i (cost=0.00..24346.00 rows=5000 width=4)
  Output: a
  Filter: ((i.c)::double precision = '1'::double precision)

Seq Scan, index still unused :-(

constant expression evaluated

⚠️ predicate is on c::double (not c)

Apply cast to numeric(3,2) such that the comparison is performed on the type of column c:

```
EXPLAIN VERBOSE
  SELECT i.a
  FROM   indexed AS i
  WHERE  i.c = sin((90 / 180.0) * pi()) :: numeric(3,2);
```

QUERY PLAN

Bitmap Heap Scan on public.indexed i (cost=663.51..10388.67 rows=30333 width=4)
  Output: a
  Recheck Cond: (i.c = 1.00::numeric(3,2))
  -> Bitmap Index Scan on indexed_c (cost=0.00..655.92 rows=30333 width=0)
        Index Cond: (i.c = 1.00::numeric(3,2))

Bitmap Index Scan! :-)

⚠️ predicate is on c

Is it worth it? Yes!

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT i.a
  FROM   indexed AS i
  WHERE  degrees(asin(i.c)) = 90;
```

QUERY PLAN
Seq Scan on public.indexed i (cost=0.00..29346.00 rows=5000 width=4) (actual time=0.075..504.166 rows=31848 loops=1)
Output: a
Filter: (degrees(asin((i.c)::double precision)) = '90'::double precision)
Rows Removed by Filter: 968152
Planning time: 0.139 ms
Execution time: 506.015 ms ⬅

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT i.a
  FROM   indexed AS i
  WHERE  i.c = sin((90 / 180.0) * pi()) :: numeric(3,2);
```

QUERY PLAN
Bitmap Heap Scan on public.indexed i (cost=663.51..10388.67 rows=30333 width=4) (actual time=16.404..35.908 rows=31848 loops=1)
Output: a
Recheck Cond: (i.c = 1.00::numeric(3,2))
Heap Blocks: exact=9346
-> Bitmap Index Scan on indexed_c (cost=0.00..655.92 rows=30333 width=0) (actual time=13.787..13.787 rows=31848 loops=1)
Index Cond: (i.c = 1.00::numeric(3,2))
Planning time: 0.227 ms
Execution time: 37.974 ms ⬅ 10× faster :-)

Indexes on Expressions

The query optimizer essentially sees the following query:

```
SELECT i.a
FROM   indexed AS i
WHERE  ■(i.c) = v
```

- In general, the RDBMS will *not be able* to form the inverse of the "black box" to rewrite the predicate into $\mathtt{i.c} = \blacksquare^{-1}(v)$:
  - $\blacksquare$ may be complex and/or user-defined and the inverse might be hard to find for the system.
  - $\blacksquare$ may not be bijective and thus have no inverse at all.

Indexes on Expressions

In an **expression-based (or: function-based) index** *I*, index entries hold the value of *an expression* over the column(s) of table *T*:

CREATE INDEX *I* **ON** *T* **USING** btree ($\underbrace{e}_{\text{expression/function over columns of } T}$)

- Expression *e* is evaluated at row *insertion/update time*.
  👍, if query speed is more important than update speed.
- Index *I* matches predicates of the form *e θ v*.
- The sequence set of index *I* is ordered by *e*.
- CREATE UNIQUE INDEX …: can protect complex constraints.

Demonstrate: an expression-based index will match the original query:

```
CREATE INDEX indexed_deg_asin_c ON indexed USING btree (degrees(asin(c)));
ANALYZE indexed;

\d indexed
                Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_a" PRIMARY KEY, btree (a)
    "indexed_c" btree (c)
    "indexed_deg_asin_c" btree (degrees(asin(c::double precision)))


EXPLAIN (VERBOSE, ANALYZE)
  SELECT i.a
  FROM   indexed AS i
  WHERE  degrees(asin(i.c)) = 90;  -- matches indexed_deg_asin_c
```

QUERY PLAN

Bitmap Heap Scan on public.indexed i (cost=599.39..10582.73 rows=31867 width=4) (actual time=12.779..33.555 rows=31848 loops=1)
  Output: a
  Recheck Cond: (degrees(asin((i.c)::double precision)) = '90'::double precision)
  Heap Blocks: exact=9346
  -> Bitmap Index Scan on indexed_deg_asin_c (cost=0.00..591.43 rows=31867 width=0) (actual time=10.211..10.211 rows=31848 loops=1)
        Index Cond: (degrees(asin((i.c)::double precision)) = '90'::double precision)
Planning time: 0.182 ms
Execution time: 35.850 ms ⬅ fast! :-)

Other useful examples for expression-based indexes:

- `CREATE INDEX ... USING btree (lower(lastname))`
    - Query:

```
SELECT ...
FROM   ...
WHERE  lower(t.lastname) = lower('Kenobi')
```

- `CREATE INDEX ... USING btree (firstname || ' ' || lastname)`

Consider expression-based index people_age on the user-defined SQL function (UDF) get_age():

```
CREATE FUNCTION get_age(d_o_b date) RETURNS int AS
$$
  SELECT extract(years from age(now(), d_o_b)) :: int
$$                                -- ☗ current system time ⚠
LANGUAGE SQL;

CREATE TABLE people (name text, birthdate date);
CREATE INDEX people_age ON people
  USING btree (get_age(birthdate));  -- expression-based index

SELECT p.name AS adult              -- ⎫
FROM   people AS p                  -- ⎬ intended index use case
WHERE  get_age(p.birthdate) >= 18;  -- ⎭
```

- **Q:** How do you expect the RDBMS to behave?

Demonstrate that PostgreSQL refuses to build the index people_age. Indexed expression must be deterministic (SQL: IMMUTABLE):

- An IMMUTABLE function cannot modify the database and is guaranteed to return the same results given the same arguments forever.
- STABLE: like IMMUTABLE but only within current SQL statement
- VOLATILE: function can do anything

```
DROP FUNCTION IF EXISTS get_age(date);
CREATE FUNCTION get_age(d_o_b date) RETURNS int AS
$$
  SELECT extract(years from age(now(), d_o_b)) :: int
$$
LANGUAGE SQL;

DROP TABLE IF EXISTS people;
CREATE TABLE people (name text, birthdate date);
CREATE INDEX people_age ON people
  USING btree (get_age(birthdate));  -- ⚠ illegal

-- → ERROR:  functions in index expression must be marked IMMUTABLE
```

Index I may be built over a **list of columns** c_i of table T:

```
CREATE INDEX I ON T USING btree (c₁,…,cₙ)
```

- In I's leaf level, the rows of T will be ordered *lexicographically*. Row t_1 is smaller than t_2, iff:

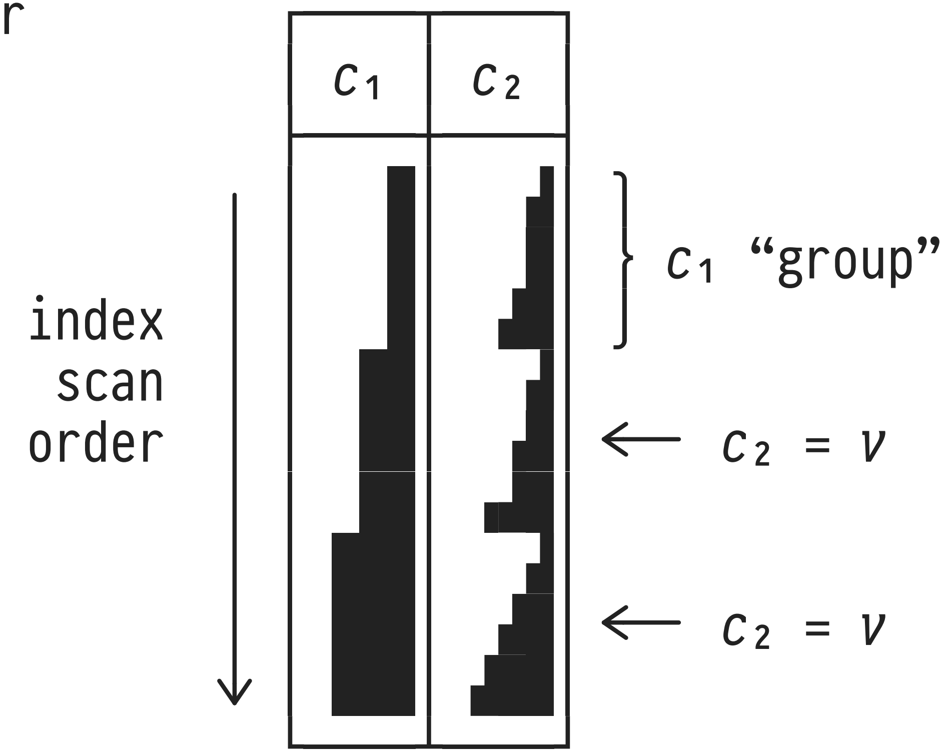
$$
\begin{aligned}
& (t_1.c_1 < t_2.c_1) \\
\vee\ & (t_1.c_1 = t_2.c_1 \wedge t_1.c_2 < t_2.c_2) \\
\vdots\ & \\
\vee\ & (t_1.c_1 = t_2.c_1 \wedge \cdots \wedge t_1.c_{n-1} = t_2.c_{n-1} \wedge t_1.c_n < t_2.c_n)
\end{aligned}
$$

- ⚠️ Row order in indexes on (c_1,c_2) and (c_2,c_1) will be entirely different. **Q:** How about (c_1) and (c_1,c_2)?

Matching Queries With Composite Indexes

Entry order in (c_1, c_2) index I (▄█ ≡ magnitude of values):

- If we scan I in order, we encounter these ascending/repeating patterns of values in columns c_1/c_2:



⇛ Composite index matches a query if its filter predicate refers to a **prefix** of the column list $(c_1, \ldots, c_n)$

Demonstrate how PostgreSQL uses/ignores a composite index based on how a filter predicate matches the index order:

```
-- ❶ Prepare composite index on table indexed:

  DROP INDEX indexed_c;
  DROP INDEX indexed indexed_deg_asin_c;

  \d indexed
                  Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_a" PRIMARY KEY, btree (a)

  ALTER TABLE indexed DROP CONSTRAINT indexed_a;
  CREATE INDEX indexed_c_a ON indexed USING btree (c,a);
  ANALYZE indexed;

  \d indexed
                  Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_c_a" btree (c, a)


  SELECT relname, relkind, relpages FROM pg_class WHERE relname LIKE 'indexed%';
```

relname	relkind	relpages
indexed indexed_c_a	r i	9346 3849

⬅ index smaller than indexed table 👍

```
-- ❷ Visualize index entry order in index indexed_c_a:
```

```
SELECT i.c, i.a FROM indexed AS i ORDER BY i.c, i.a LIMIT 20;
```

c	a
-1.00	11
-1.00	55
-1.00	99
-1.00	124
-1.00	143
-1.00	168
-1.00	187
-1.00	212
-1.00	231
-1.00	256
-1.00	300
-1.00	344
-1.00	388
-1.00	432
-1.00	476
-1.00	501
-1.00	520
-1.00	545
-1.00	564
-1.00	589

(20 rows)

```
-- ❸ Evaluate query with predicate matching the (c,a) index:

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.*
    FROM   indexed AS i
    WHERE  i.c = 0.42;   -- ⬅ (c) is a prefix of (c,a)
```

QUERY PLAN
Bitmap Heap Scan on public.indexed i (cost=89.89..7265.04 rows=3802 width=41) (actual time=4.389..9.263 rows=3531 loops=1)
Output: a, b, c
Recheck Cond: (i.c = 0.42)
Heap Blocks: exact=2964
Buffers: shared hit=2980 ⬅ matching rows found all over table indexed
-> Bitmap Index Scan on indexed_c_a (cost=0.00..88.94 rows=3802 width=0) (actual time=3.006..3.006 rows=3531 loops=1)
Index Cond: (i.c = 0.42)
Buffers: shared hit=16 ⬅ touches few index pages only :-)
Planning time: 0.145 ms
Execution time: 9.785 ms

```
-- ❹ Evaluate query with predicate NOT matching the (c,a) index:

   EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
     SELECT i.*
     FROM   indexed AS i
     WHERE  i.a = 42;   -- ◀ (a) not a prefix of (c,a)
```

QUERY PLAN
Seq Scan on public.indexed i (cost=0.00..21846.00 rows=1 width=41) (actual time=0.038..173.714 rows=1 loops=1)
Output: a, b, c
Filter: (i.a = 42)
Rows Removed by Filter: 999999
Buffers: shared hit=3113 read=6233 ◀ reads all 9346 = (3113 + 6233) pages of table
Planning time: 0.123 ms
Execution time: 173.747 ms

```
-- ❺ Force PostgreSQL to use the (c,a) index despite the non-matching
      predicate: will touch (almost) all pages of the index.

  SET enable_seqscan = off;
  SET enable_indexscan = off;

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.*
    FROM   indexed AS i
    WHERE  i.a = 42;
```

QUERY PLAN
Bitmap Heap Scan on public.indexed i (cost=22896.43..22900.44 rows=1 width=41) (actual time=72.729..72.729 rows=1 loops=1)
Output: a, b, c
Recheck Cond: (i.a = 42)
Heap Blocks: exact=1 ◀ we are lucky: just a single row matches
Buffers: shared hit=3831
-> Bitmap Index Scan on indexed_c_a (cost=0.00..22896.42 rows=1 width=0) (actual time=72.716..72.716 rows=1 loops=1)
Index Cond: (i.a = 42)
Buffers: shared hit=3830 ◀ scans almost all pages of the index :-(
Planning time: 0.088 ms
Execution time: 72.765 ms

```
  SET enable_seqscan = on;
  SET enable_indexscan = on;
```

Multi-Dimensional Queries and Composite Indexes

Composite indexes are designed to support *multi-dimensional* queries whose predicates refer to *multiple* columns:

```
SELECT e(t)
FROM   T AS t
WHERE  p1(t.c1)   -- two dimensions:
  AND  p2(t.c2)   -- c1, c2
```

- **Q:** Shall we build a (c_1,c_2) or a (c_2,c_1) index to support this query?

- 💡 Hmm... What would PostgreSQL do?

Demonstrate how the selectivity of p_1 and p_2 determines which index is used by PostgreSQL. Using table indexed(a,b,c) with indexes

- indexed_a_c(a,c)
- indexed_c_a(c,a)

Predicates:

- p_1: i.c BETWEEN 0.00 AND 0.01
- p_2: i.a BETWEEN 0 AND m, adjust m to influence selectivity

```
-- ❶ Table indexed and its indexes:

  CREATE INDEX indexed_a_c ON indexed USING btree(a,c);
  ANALYZE indexed;

  \d indexed
                Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_a_c" btree (a, c)
    "indexed_c_a" btree (c, a)

  ANALYZE indexed;


-- ❷ Modify parameter m to render p₂ more and more selective such that
   PostgreSQL switches from using index (c,a) to (a,c).  Can perform
   binary search regarding m to find switch point:

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.b
    FROM   indexed AS i
    WHERE  i.c BETWEEN 0.00 AND 0.01  -- p₁ more selective
      AND  i.a BETWEEN 0 AND 10000;   -- p₂ with m = 10000 less selective
```

QUERY PLAN

Bitmap Heap Scan on public.indexed i (cost=101.11..231.64 rows=34 width=33) (actual time=4.403..4.497 rows=56 loops=1)
  Output: b
  Recheck Cond: ((i.c >= 0.00) AND (i.c <= 0.01) AND (i.a >= 0) AND (i.a <= 10000))

```
  Heap Blocks: exact=33
  ->  Bitmap Index Scan on indexed_c_a  (cost=0.00..101.10 rows=34 width=0) (actual time=4.378..4.378 rows=56 loops=1)
        Index Cond: ((i.c >= 0.00) AND (i.c <= 0.01) AND (i.a >= 0) AND (i.a <= 10000))
Planning time: 0.473 ms
Execution time: 4.562 ms
```

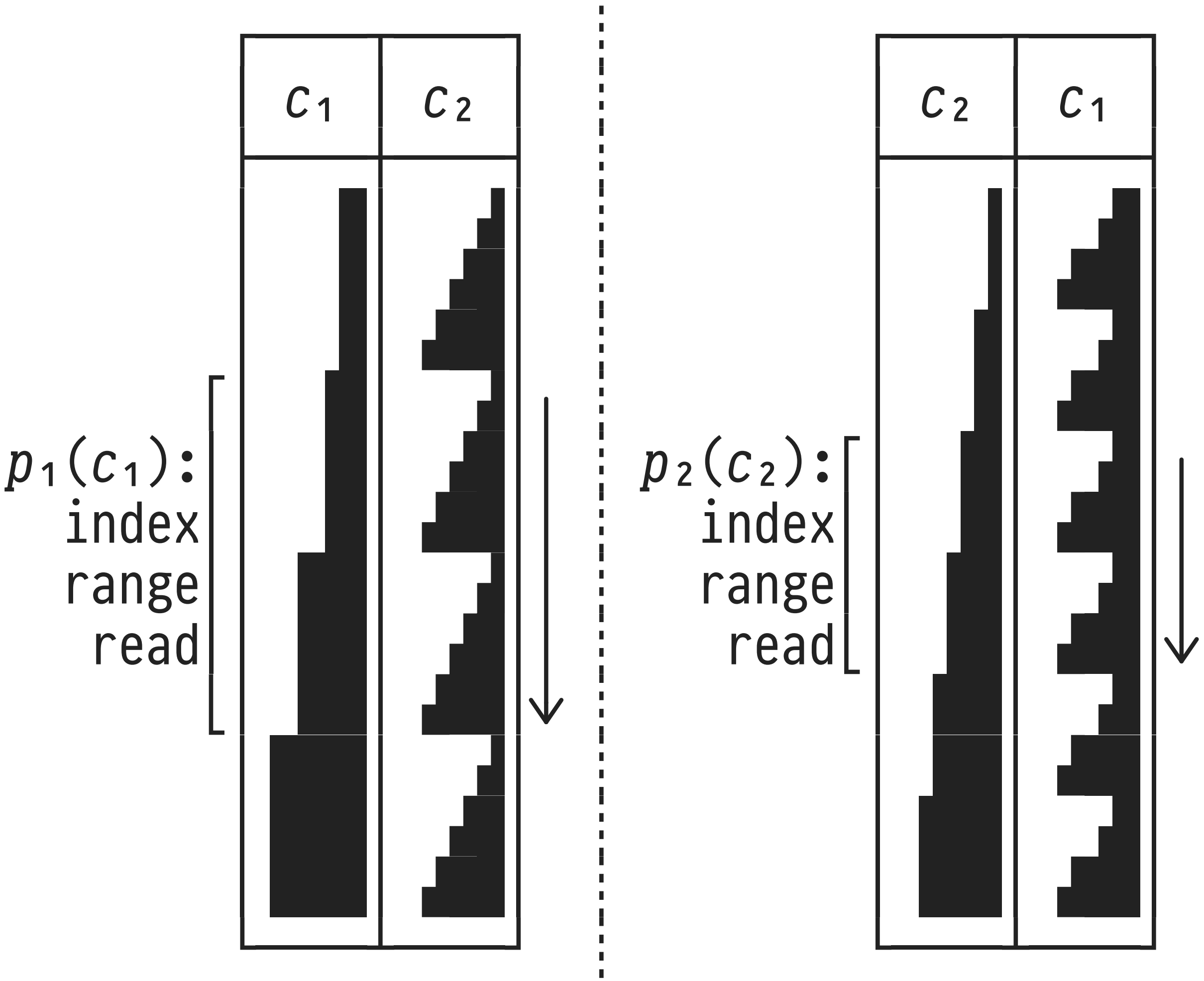
```
-- m = 5000: Bitmap Index Scan on indexed_c_a
-- m = 2500: Index Scan on indexed_a_c
-- m = 3750: Index Scan on indexed_a_c
-- m = 4385: Index Scan on indexed_c_a

-- ⇒ Switch point at m ≈ 3915  ⚠️ This may vary slightly from ANALYZE to ANALYZE

EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT i.b
  FROM   indexed AS i
  WHERE  i.c BETWEEN 0.00 AND 0.01
    AND  i.a BETWEEN 0 AND 3914;    -- p₂ with m = 3914 more selective
```

```
                                                QUERY PLAN
Index Scan using indexed_a_c on public.indexed i  (cost=0.42..151.85 rows=13 width=33) (actual time=0.303..2.393 rows=21 loops=1)
  Output: b
  Index Cond: ((i.a >= 0) AND (i.a <= 3914) AND (i.c >= 0.00) AND (i.c <= 0.01))
  Buffers: shared hit=31
Planning time: 0.358 ms
Execution time: 2.452 ms
```

Composite Indexes: Index for Selective Dimension First



- Leading column c_i and predicate p_i define index scan range: ↓

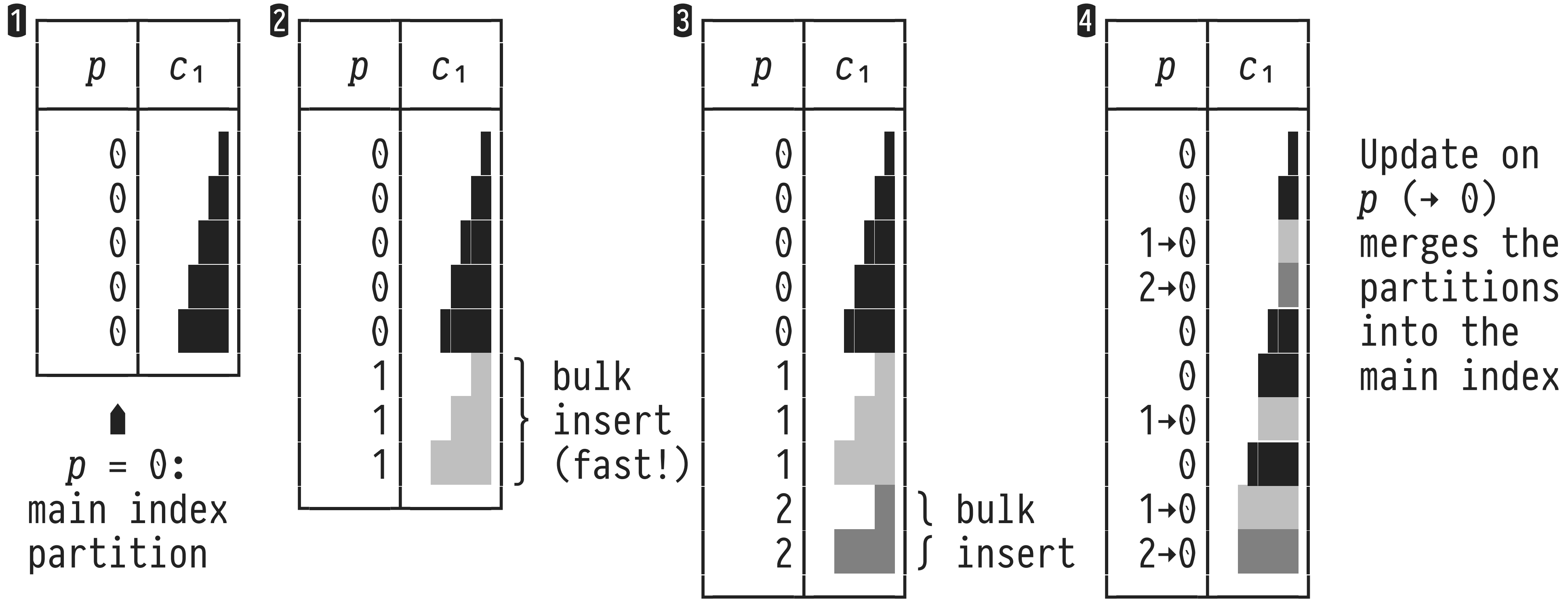
- Aim to minimize work, i.e., index scan range: the **more selective predicate** determines choice of index (c_2,c_1)

- If you can afford one of the two indexes only, build (c_2,c_1)

⇒ Rule of thumb: “Index for ‘=’ first!”

4 ⁝ Partitioned B⁺Trees: Non-Selective Key Prefixes

Indexes in which an artificial partitioning column *p* of **low selectivity** is prepended to the index key can be useful:



This follows Goetz Graefe's "Partitioned B-trees – a user's guide", also see
https://pdfs.semanticscholar.org/78ce/cd5f738c26ddefb3633f8a50bd6397ebc8dc.pdf

Advantages of the scheme:

- Addition of artificial column *p* can be easily hidden from user/applications through views.
- Bulk inserts with increasing value for *p* (= 1, 2, ...) effectively leads to efficient B⁺Tree appends (see chapter on Ordered Indexes), pages of main index not affected and do not need to be locked while the bulk insert happens.
- Can merge new partitions into main partition *p* = 0 when convenient. Can control how many partitions to merge at any one time. Graefe paper: can even iteratively merge a single partition in small chunks of rows (see PL/SQL function reorganize() below).

```sql
-- 1
DROP TABLE IF EXISTS parts;
CREATE TABLE parts (a int, b text, c numeric(3,2));

ALTER TABLE parts
  ADD COLUMN p int NOT NULL CHECK (p >= 0) DEFAULT 0;

INSERT INTO parts(a,b,c)
  SELECT i, md5(i::text), sin(i)
  FROM   generate_series(1,1000000) AS i;

CREATE INDEX parts_p_a ON parts USING btree (p, a);
CLUSTER parts USING parts_p_a;
ANALYZE parts;

-- 2
INSERT INTO parts(p,a,b,c)
  --
  SELECT 1, random() * 1000000, md5(i::text), sin(i)
  FROM   generate_series(1,100000) AS i;

-- 3
INSERT INTO parts(p,a,b,c)
  --
  SELECT 2, random() * 1000000, md5(i::text), sin(i)
  FROM   generate_series(1,100000) AS i;

-- 4
UPDATE parts AS p
SET    p = 0 -- ⬅ merge partition 1 into main partition 0
WHERE  p.p = 1;

UPDATE parts AS p
SET    p = 0 -- ⬅ merge partition 2 into main partition 0
WHERE  p.p = 2;
```

Simple implementation of bulk appends and delayed merging:

```
-- 1 Prepend partition column p, build partitioned B+Tree I
ALTER TABLE T
  ADD COLUMN p int NOT NULL CHECK (p >= 0) DEFAULT 0;
CREATE INDEX I ON T USING btree (p,c1);

-- 2+3 Fast⚡ bulk inserts (simply appends to B+Tree I)
INSERT INTO T(p,…) SELECT 1, …  FROM …;
INSERT INTO T(p,…) SELECT 2, …  FROM …;

-- 4 Merge partition(s) into main partition when convenient
UPDATE T AS t
SET    p = 0
WHERE  t.p = 1;  -- or: t.p IN (‹partitions›) ¦ t.p <> 0
```

Consider a SQL query with a *disjunctive* predicate:

```
SELECT e(t)
FROM   T AS t
WHERE  p₁(t.c₁)  -- ⎫ disjunctive
   OR  p₂(t.c₂)  -- ⎭ predicate
```

- Neither a (c_1, c_2) nor a (c_2, c_1) index can support the disjunction: we would need to scan the *entire* index 👎 (thus: rather access T's heap file directly).

- 💡 Try to **combine separate indexes** on columns I_1 on (c_1) and I_2 on (c_2). Idea builds on Bitmap Index Scan.

Demonstrate the use of bitmap index ORing to implement index support for a disjunctive predicate:

```
-- ❶ Prepare separate indexes on columns a and c:

  DROP INDEX indexed_a_c;
  DROP INDEX indexed_c_a;

  CREATE INDEX indexed_a ON indexed USING btree (a);
  CREATE INDEX indexed_c ON indexed USING btree (c);
  ANALYZE indexed;

  \d indexed
               Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text			
c	numeric(3,2)			

```
Indexes:
    "indexed_a" btree (a)
    "indexed_c" btree (c)


-- ❷ Perform query featuring a disjunctive predicate:

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.b
    FROM   indexed AS i
    WHERE  i.c BETWEEN 0.00 AND 0.01
       OR  i.a BETWEEN 0 AND 4000;
```

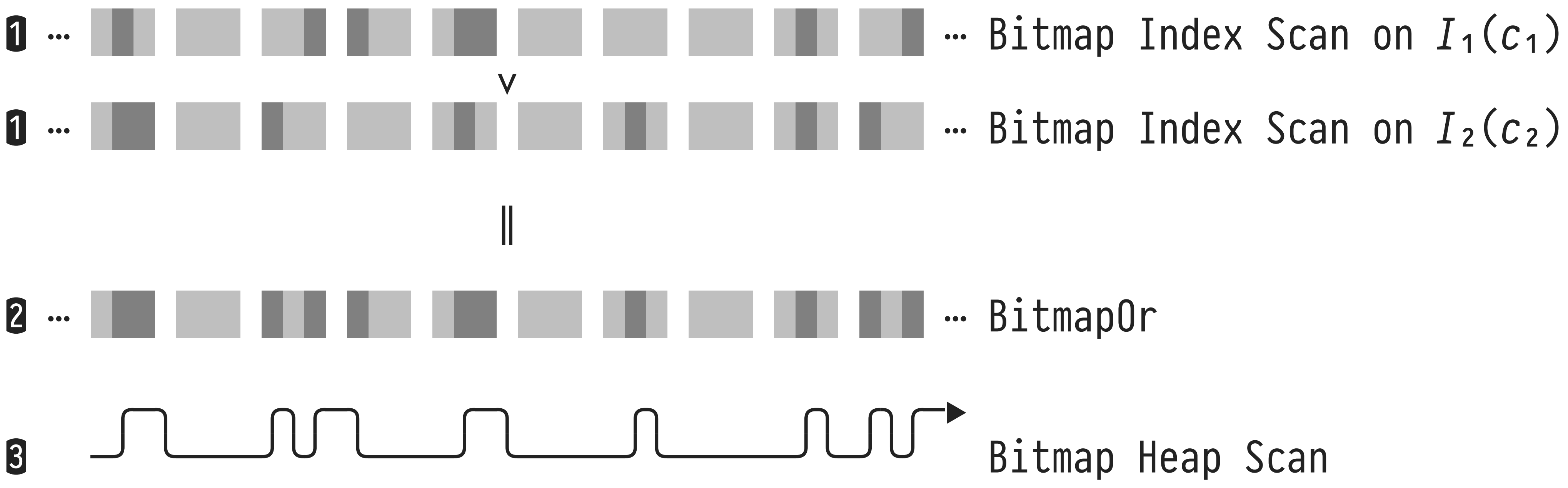
QUERY PLAN

Bitmap Heap Scan on public.indexed i (cost=160.85..9481.56 rows=7227 width=33) (actual time=6.849..15.713 rows=10361 loops=1)
  Output: b
  Recheck Cond: (((i.c >= 0.00) AND (i.c <= 0.01)) OR ((i.a >= 0) AND (i.a <= 4000)))
  Heap Blocks: exact=3647
  Buffers: shared hit=3681
  -> BitmapOr (cost=160.85..160.85 rows=7240 width=0) (actual time=5.861..5.861 rows=0 loops=1)
      ⬆ Buffers: shared hit=34 ⬅ ORing of bitmaps reads all 21 + 13 = 34 pages holding the individual bitmaps
        -> Bitmap Index Scan on indexed_c (cost=0.00..68.67 rows=3224 width=0) (actual time=4.511..4.511 rows=6383 loops=1)
            ⬆ Index Cond: ((i.c >= 0.00) AND (i.c <= 0.01))
              Buffers: shared hit=21 ⬅
        -> Bitmap Index Scan on indexed_a (cost=0.00..88.58 rows=4015 width=0) (actual time=1.347..1.347 rows=4000 loops=1)

```
            Index Cond: ((i.a >= 0) AND (i.a <= 4000))
            Buffers: shared hit=13
Planning time: 0.520 ms
Execution time: 16.990 ms
```

Combining Indexes via Bitmap Heap Scan and BitmapOr/And

1. Perform individual **Bitmap Index Scans**, possibly in //, possibly multiple times on the same index.
2. Combine resulting row-/page-level bitmaps using ∨ or ∧.
3. Perform **Bitmap Heap Scan** with combined bitmap.



Demonstrate that combined the Bitmap Index Scans indeed pays off, despite the scan of *two* indexes, bitmap ORing, and Bitmap Heap Scan:

```
set enable_bitmapscan = off;

EXPLAIN (VERBOSE, ANALYZE)
  SELECT i.b
  FROM   indexed AS i
  WHERE  i.c BETWEEN 0.00 AND 0.01
     OR  i.a BETWEEN 0 AND 4000;
```

QUERY PLAN
Seq Scan on public.indexed i (cost=0.00..29346.00 rows=7227 width=33) (actual time=0.051..388.609 rows=10361 loops=1)
Output: b
Filter: (((i.c >= 0.00) AND (i.c <= 0.01)) OR ((i.a >= 0) AND (i.a <= 4000)))
Rows Removed by Filter: 989639
Planning time: 0.363 ms
Execution time: 389.674 ms ⬅ with bitmap ORing above: 16.9 ms

```
set enable_bitmapscan = on;
```

Q: Can indexes support the evaluation of SQL **string pattern matches** LIKE '%this'? **A:** Yes, but it depends on the pattern.

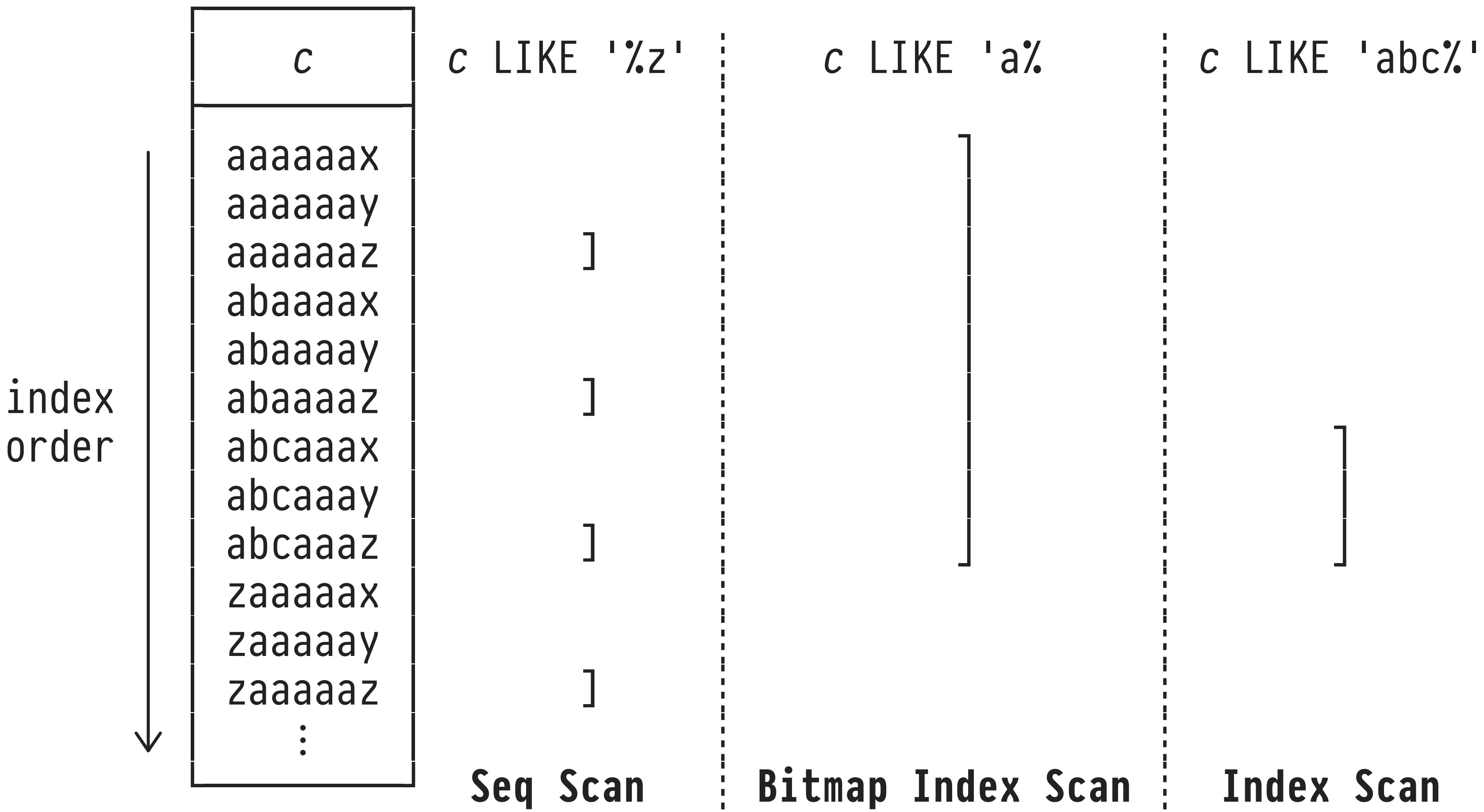
- SQL pattern matching: e LIKE 's_1%s_2' holds iff string e constains substrings $s_{1,2}$ separated by zero or more arbitrary characters (regular expressions: 's_1.*s_2').

- PostgreSQL: B^+ tree index on column c :: text of table T:

```
-- I₁ supports LIKE
CREATE INDEX I₁ on T USING btree (c text_pattern_ops)
-- I₂ supports =, <, >, …
CREATE INDEX I₂ on T USING btree (c)
```

Patterns, Selectivity, and Index Ranges

- Placement of wildcard % influences predicate selectivity:



Demonstrate how the string patterns (LIKE) influence predicate selectivity and the resulting (index) scans chosen by PostgreSQL:

```
CREATE INDEX indexed_b ON indexed USING btree (b text_pattern_ops);
ANALYZE indexed;

\d indexed
                Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_a" btree (a)
    "indexed_b" btree (b text_pattern_ops)
    "indexed_c" btree (c)

-- 1 Leading % wildcard: low selectivity
  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE  i.b LIKE '%42';
```

QUERY PLAN

Seq Scan on public.indexed i (cost=0.00..21846.00 rows=100 width=37) (actual time=0.205..366.285 rows=3939 loops=1)
  Output: a, b
  Filter: (i.b ~~ '%42'::text)
  Rows Removed by Filter: 996061
  Buffers: shared hit=32 read=9314
Planning time: 3.253 ms
Execution time: 366.842 ms

```
-- 2 Leading character: medium selectivity
  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE  i.b LIKE 'a%42';
```

QUERY PLAN
Bitmap Heap Scan on public.indexed i (cost=2720.45..12941.45 rows=100 width=37) (actual time=36.944..130.989 rows=239 loops=1)

```
   Output: a, b
   Filter: (i.b ~~ 'a%42'::text) ◀ post-processing: match trailing 42
   Rows Removed by Filter: 62058
   Heap Blocks: exact=9340 ◀ Heap Scan
   Buffers: shared read=9788
   ->  Bitmap Index Scan on indexed_b  (cost=0.00..2720.43 rows=70000 width=0) (actual time=34.028..34.028 rows=62297 loops=1)
         Index Cond: ((i.b ~>=~ 'a'::text) AND (i.b ~<~ 'b'::text)) ◀ pattern match rewritten into range query
         Buffers: shared read=448
 Planning time: 2.383 ms
 Execution time: 131.089 ms
```

```
-- ❸ Leading characters: selectivity increases with length of
--    character sequence
  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE  i.b LIKE 'abc%42';
```

```
                                    QUERY PLAN
 Index Scan using indexed_b on public.indexed i  (cost=0.42..8.45 rows=100 width=37) (actual time=2.101..2.432 rows=2 loops=1)
   Output: a, b
   Index Cond: ((i.b ~>=~ 'abc'::text) AND (i.b ~<~ 'abd'::text)) ◀ pattern match rewritten into selective range query
   Filter: (i.b ~~ 'abc%42'::text) ◀ post-processing: match trailing 42
   Rows Removed by Filter: 243
   Buffers: shared hit=170 read=80
 Planning time: 0.297 ms
 Execution time: 2.558 ms
```

Sometimes, small parts of a table contain 🔥 "hot" rows while most of the table only has archival value:

Table **orders**

id	…	fulfilled	
42		☐	"hot" open orders
41		☐	
39		☐	
40		2018-06-03	closed orders only used in reporting
38		2018-05-27	
⋮		⋮	
2		2017-08-26	
1		2017-04-10	

- Lion share of rows is cold, read infrequently (*e.g.*, to create a monthly report).
- Hot row subset queried regularly, would benefit from index support.
- But: Hot rows would be distributed all over a regular index. 👎
- Predicate *p* discerns hot rows (*e.g.*, `fulfilled IS NULL`).

Partial Indexes

💡 Build **partial index** that covers the hot row subset only:

```
CREATE INDEX I on T USING btree (c₁,c₂,…) WHERE p(cₚ)
```

- I will be small: only rows of T satisfying p are be present in the index.
- Updates on column(s) c_p may move rows into/out of I.
- I matches a query if its filter predicate q **implies** p:

```
SELECT e(t)
FROM   T AS t
WHERE  q(t)    -- q ⇒ p?
```

  - RDBMSs typically recognize trivial implications only.

Demonstrate construction and matching of partial index on table indexed:

```
-- ❶ Create partial index: a row is "hot" if its c values exceeds 0.5

  CREATE INDEX indexed_partial_a ON indexed USING btree (a)
    WHERE c >= 0.5;
  ANALYZE indexed;

  \d indexed;
                Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_a" btree (a)
    "indexed_b" btree (b text_pattern_ops)
    "indexed_c" btree (c)
    "indexed_partial_a" btree (a) WHERE c >= 0.5


-- ❷ Check: the partial index is indeed smaller than the regular/full indexes

  SELECT relname, relkind, relpages
  FROM   pg_class
  WHERE  relname LIKE 'indexed%';
```

relname	relkind	relpages
indexed indexed_a indexed_b indexed_c indexed_partial_a	r i i i i	9346 2745 7210 2745 922 ⬅

```
-- ❸ Do these queries match the partial index?
--    QUIZ: How till `Index Cond' and `Filter` (if any) look like?

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.a
    FROM   indexed AS i
    WHERE  c >= 0.6 AND a < 1000;   -- ⬅ c >= 0.6 ⇒ c >= 0.5 ✓ match
```

```
                                                    QUERY PLAN
Index Scan using indexed_partial_a on public.indexed i  (cost=0.42..18.00 rows=291 width=4) (actual time=0.050..0.477 rows=295)
  Output: a
  Index Cond: (i.a < 1000)
  Filter: (i.c >= 0.6)
  Rows Removed by Filter: 40
  Buffers: shared hit=13
Planning time: 0.351 ms
Execution time: 0.610 ms
```

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT i.a
  FROM   indexed AS i
  WHERE  c >= 0.5 AND a < 1000;  -- ⬅ perfect match for partial index, no Filter required
```

```
                                                    QUERY PLAN
Index Only Scan using indexed_partial_a on public.indexed i  (cost=0.42..17.18 rows=329 width=4) (actual time=0.051..0.333 rows=335)
  Output: a
  Index Cond: (i.a < 1000)
  Heap Fetches: 335
  Buffers: shared hit=13
Planning time: 0.331 ms
Execution time: 0.435 ms ⬅ delivers more rows but executes faster
```

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT i.a
  FROM   indexed AS i
  WHERE  c >= 0.4 AND a < 1000;  -- ⬅ c >= 0.4 ⇏ c >= 0.5  no match, use full index instead
```

```
                                                    QUERY PLAN
Index Scan using indexed_a on public.indexed i  (cost=0.42..45.12 rows=368 width=4) (actual time=0.050..1.416 rows=371 loops=1)
  Output: a
  Index Cond: (i.a < 1000)
  Filter: (i.c >= 0.4)
  Rows Removed by Filter: 628
  Buffers: shared hit=13 read=2
Planning time: 0.302 ms
Execution time: 1.530 ms
```

For some queries, **all columns** $c_1,\ldots,c_n$ needed for evaluation may be present as key values in an index.

- 💡 Perform **index-only** query evaluation, do not access the tables' heap files at all. 🚀
- May even try to design wide multi-column indexes[1] with keys $c_1,\ldots,c_k,c_{k+1},\ldots,c_n$, in which
  - prefix $c_1,\ldots,c_k$ is used to guide index search (*i.e.*, to evaluate predicates),
  - suffix $c_{k+1},\ldots,c_n$ is used to evaluate other expressions.

[1] PostgreSQL v11: CREATE INDEX *I* on *T* USING btree ($c_1,\ldots,c_k$) INCLUDE ($c_{k+1},\ldots,c_n$), builds a B⁺Tree in which keys ($c_1,\ldots,c_k$) are narrow and only the leaves carry all columns $c_1,\ldots,c_n$.

Index-Only Queries?

Assume B$^+$Tree index (a,c) on table indexed. **Q:** Can ❶…❻ be evaluated using the index only?

❶
```
SELECT i.c
FROM   indexed AS i
WHERE  i.a < v
```

❷
```
SELECT i.a
FROM   indexed AS i
WHERE  i.c < v
```

❸
```
SELECT i.a / i.c AS div
FROM   indexed AS i
WHERE  i.a < v AND i.c <> 0
```

❹
```
SELECT MAX(i.c) AS m
FROM   indexed AS i
WHERE  i.a < v;
```

❺
```
SELECT i.a, SUM(i.c) AS s
FROM   indexed AS i
GROUP BY i.a;
```

❻
```
SELECT MIN(i.b) AS m
FROM   indexed AS i
WHERE  i.a < v;
```

```
❶ yes ❷ (yes, can enforce Index-Only Scan with enable_seqscan = off) ❸ yes ❹ yes ❺ yes ❻ no (reference to i.b)
```

Index-Only Scans and Row Visibility

Index-only query evaluation — implemented by plan operator Index Only Scan — in PostgreSQL faces a challenge:

- Row visibility (recall timestamps xmin, xmax) is recorded in the heap file *only*.
    - **Huh?** Index Only Scan needs to check the heap file whether an index entry may occur in the query result…

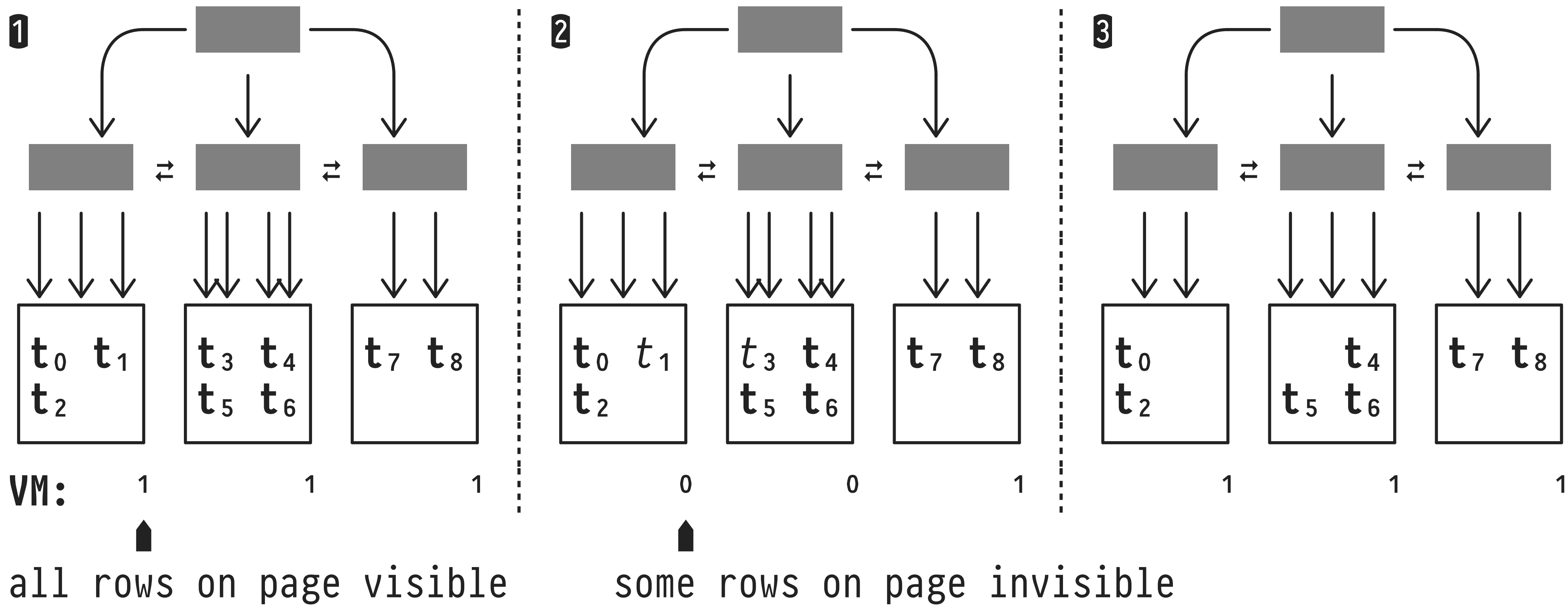
- Instead check the table's/heap file's **visibility map** to efficiently check that all rows of a page are visible.
    - Use Index Only Scan when no/few row visibility checks require actual heap file accesses.

Index-Only Scans and the Visibility Map (VM)

❶ Original table state ($\mathbf{t}_i$: visible row).

❷ After deletion of $\mathbf{t}_1$ and $\mathbf{t}_3$ (t_i: invisible row).

❸ After VACUUM: dead rows removed, index updated.



Demonstrate index-only query evaluation over table indexed and its interplay with the table's visibility map:

```
-- ❶ Prepare (a,c) index.  Make sure that all rows on all
--   pages are indeed visible (`VACCUM`)

  CREATE INDEX indexed_a_c ON indexed USING btree (a,c);
  ANALYZE indexed;
  VACUUM indexed;

  \d indexed
                Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text			
c	numeric(3,2)			

```
Indexes:
    "indexed_a" PRIMARY KEY, btree (a)
    "indexed_a_c" btree (a, c)


-- ❷ Use extension pg_visibility to check the visibility map
--   (table indexed has 9346 pages)

  CREATE EXTENSION IF NOT EXISTS pg_visibility;

  SELECT blkno, all_visible
  FROM   pg_visibility('indexed');
```

blkno	all_visible
0	t
1	t
2	t
[...]	
9344	t
9345	t

```
  SELECT all_visible
  FROM   pg_visibility_map_summary('indexed');
```

all_visible

```
      9346
```

```
-- ❸ Perform sample index-only query

 EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
   SELECT SUM(i.c) AS s
   FROM   indexed AS i
   WHERE  i.a < 10000;
```

```
                                                    QUERY PLAN
Aggregate  (cost=382.37..382.38 rows=1 width=32) (actual time=8.406..8.407 rows=1 loops=1)
  Output: sum(c)
  Buffers: shared hit=42          ▼
  ->  Index Only Scan using indexed_a_c on public.indexed i  (cost=0.42..355.62 rows=10697 width=4) (actual time=0.050..3.731 rows=9999)
      ▲ Output: a, c
        Index Cond: (i.a < 10000)
        Heap Fetches: 0 ⬅ visibility map: no heap file acccess required :-)
        Buffers: shared hit=42 ⬅ touch few index-only pages
Planning time: 0.168 ms
Execution time: 8.468 ms
```

```
  set enable_indexonlyscan = off;
  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT SUM(i.c) AS s
    FROM   indexed AS i
    WHERE  i.a < 10000;
```

```
                                                    QUERY PLAN
Aggregate  (cost=437.37..437.38 rows=1 width=32) (actual time=10.419..10.419 rows=1 loops=1)
  Output: sum(c)
  Buffers: shared hit=124     ▼
  ->  Index Scan using indexed_a on public.indexed i  (cost=0.42..410.62 rows=10697 width=4) (actual time=0.058..4.927 rows=9999)
      ▲ Output: a, b, c
        Index Cond: (i.a < 10000)
        Buffers: shared hit=124 ⬅ touches more index + heap file pages
Planning time: 0.160 ms
Execution time: 10.482 ms
```

```
  set enable_indexonlyscan = on;
```

```
-- ❹ Table updates create old row version that are invisible
--   and may not be produced by an index-only scan

  UPDATE indexed AS i
  SET    b = '!'
  WHERE  i.a % 150 = 0;  -- updates 6666 rows

  SELECT all_visible
  FROM   pg_visibility_map_summary('indexed');
```

all_visible
2679 ⬅ on 9346 - 2679 = 6667 pages some rows may be invisible ⇒ need heap file visibility checks

```
  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT SUM(i.c) AS s
    FROM   indexed AS i
    WHERE  i.a < 10000;
```

QUERY PLAN
Aggregate (cost=418.45..418.46 rows=1 width=32) (actual time=9.066..9.067 rows=1 loops=1)
Output: sum(c)
Buffers: shared hit=198
-> Index Only Scan using indexed_a_c on public.indexed i (cost=0.42..391.63 rows=10726 width=4) (actual time=0.026..5.134 rows=9999)
☗ Output: a, c
Index Cond: (i.a < 10000)
Heap Fetches: 7155 ⬅ # of rows for which visibility check on heap was needed
Buffers: shared hit=198
Planning time: 0.090 ms
Execution time: 9.107 ms

```
-- ❺ Touch even more rows, requiring even more heap-based visibility checks
--   ⇒ index-only scan becomes unattractive

  UPDATE indexed AS i
  SET    b = '!'
  WHERE  i.a % 10 = 0;  -- updates 100000 rows, EVERY page is affected

  SELECT all_visible
  FROM   pg_visibility_map_summary('indexed');
```

all_visible

```
               0 ◀ invisible rows are found on EVERY page
```

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT SUM(i.c) AS s
  FROM   indexed AS i
  WHERE  i.a < 10000;
```

QUERY PLAN
Aggregate (cost=8312.71..8312.72 rows=1 width=32) (actual time=11.919..11.919 rows=1 loops=1)
Output: sum(c)
-> Index Scan using indexed_a on public.indexed i (cost=0.42..8287.32 rows=10155 width=4) (actual time=0.057..6.520 rows=9999)
Output: a, b, c
Index Cond: (i.a < 10000)
Planning time: 0.162 ms
Execution time: 11.984 ms

```
-- 6 Perform VACUUM to identify invisible rows and mark their
--   space ready for re-use (does not reclaim space and return it
--   to the OS yet), all remaining rows are visible

  VACUUM indexed;

  SELECT all_visible
  FROM   pg_visibility_map_summary('indexed');
```

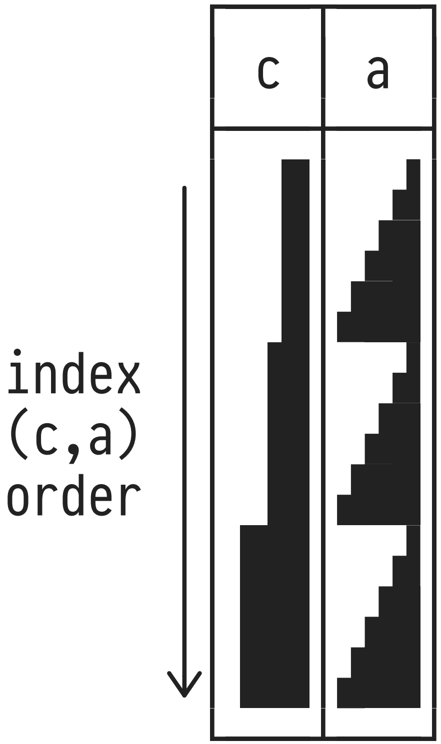
all_visible
9846

◀ rows spread over more pages now, but all rows on all pages are visible

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT SUM(i.c) AS s
  FROM   indexed AS i
  WHERE  i.a < 10000;
```

QUERY PLAN
Aggregate (cost=363.53..363.54 rows=1 width=32) (actual time=8.931..8.931 rows=1 loops=1)
Output: sum(c)
-> Index Only Scan using indexed_a_c on public.indexed i (cost=0.42..338.14 rows=10155 width=4) (actual time=0.093..4.258 rows=9999)
Output: a, c

```
        Index Cond: (i.a < 10000)
        Heap Fetches: 0
Planning time: 0.198 ms
Execution time: 8.996 ms
```



B$^+$Trees provide **ordered access** to rows. Query operations other than predicate filters should be able to benefit.

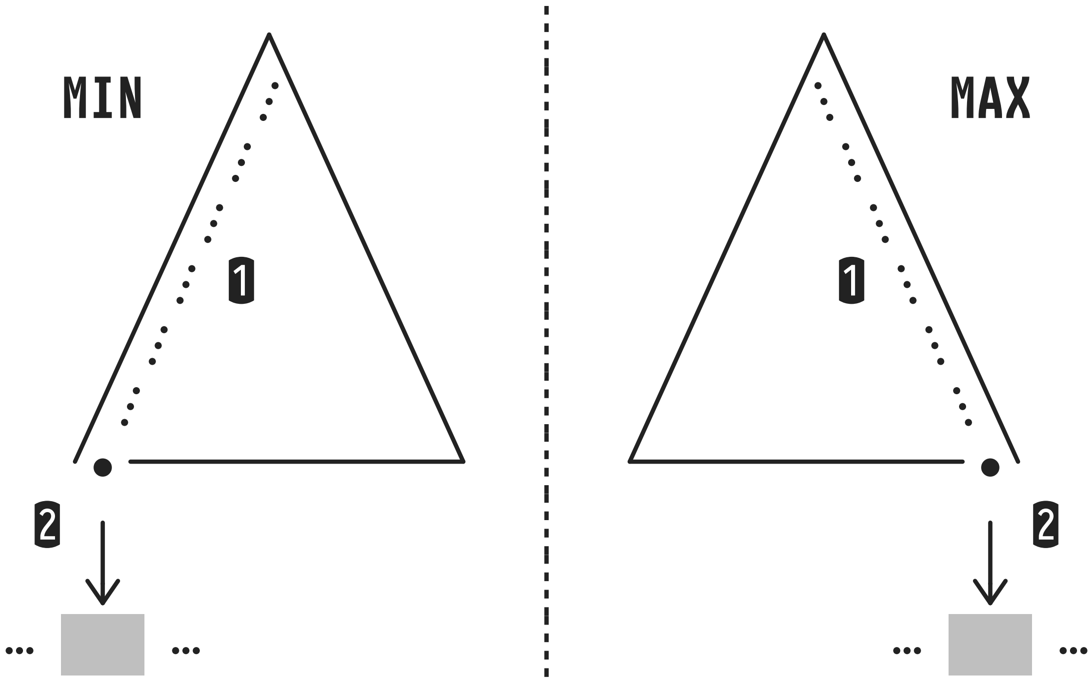
- For the following, assume that table `indexed` features two-column index `indexed_c_a` on `(c,a)` only.
- In an index scan, we will encounter rows *as if* they had been sorted by `ORDER BY c ASC, a ASC` (see left).

Supporting MIN/MAX With B+Trees

```
SELECT MIN(i.c) AS m  -- or: MAX(i.c)
FROM   indexed AS i
```

1. Descend on left/rightmost path
2. Initiate **Index Only Scan [Backward]**

Q: Which **Index Cond** will the scans use?



Demonstrate the index-only evaluation of MIN(i.c)/MAX(i.c) and the enforcement of the SQL NULL semantics:

```
-- ❶ Prepare table and the index
  DROP TABLE IF EXISTS indexed;
  CREATE TABLE indexed (a int PRIMARY KEY,
                        b text,
                        c numeric(3,2));

  INSERT INTO indexed(a,b,c)
          SELECT i, md5(i::text), sin(i)
          FROM   generate_series(1,1000000) AS i;

  ALTER TABLE indexed DROP CONSTRAINT indexed_pkey;
  CREATE INDEX indexed_c_a ON indexed USING btree (c,a);
  ANALYZE indexed;

  \d indexed
                Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_c_a" btree (c, a)

-- ❷ Index-only evaluation of MIN(i.c)/MAX(i.c)

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT MIN(i.c) AS m
    FROM   indexed AS i;
```

QUERY PLAN

Result (cost=0.46..0.47 rows=1 width=32) (actual time=0.049..0.049 rows=1 loops=1)
  Output: $0
  Buffers: shared hit=4
  InitPlan 1 (returns $0)
    -> Limit (cost=0.42..0.46 rows=1 width=4) (actual time=0.042..0.043 rows=1 loops=1)
          Output: i.c
          Buffers: shared hit=4
          -> Index Only Scan using indexed_c_a on indexed i (cost=0.42..32896.43 rows=1000000 width=4) (actual time=0.040..0.040 rows=1)
              ☝ Output: i.c
                Index Cond: (i.c IS NOT NULL) ⬅ enforce SQL aggregate semantics (NULLs ignored)   scan delivers single row only (Limit)

```
                Heap Fetches: 0
                Buffers: shared hit=4 ◀ descend B-tree of 3 levels (2,1,0) + 1 heap file access
Planning time: 0.200 ms
Execution time: 0.129 ms
```

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT MAX(i.c) AS m
  FROM   indexed AS i;
```

```
QUERY PLAN
Result  (cost=0.46..0.47 rows=1 width=32) (actual time=0.045..0.046 rows=1 loops=1)
  Output: $0
  Buffers: shared hit=4
  InitPlan 1 (returns $0)
    ->  Limit  (cost=0.42..0.46 rows=1 width=4) (actual time=0.040..0.041 rows=1 loops=1)
          Output: i.c
          Buffers: shared hit=4
          ->  Index Only Scan Backward using indexed_c_a on indexed i  (cost=... rows=1000000) (actual time=0.037..0.037 rows=1)
                Output: i.c
                Index Cond: (i.c IS NOT NULL)
                Heap Fetches: 0
                Buffers: shared hit=4
Planning time: 0.228 ms
Execution time: 0.093 ms
```

Supporting ORDER BY With B+Trees?

ORDER BY criteria need to match the row visit order of a (c,a) index forward/backward scan:

❶
```sql
SELECT i.*
FROM   indexed AS i
ORDER BY i.c
```

❷
```sql
SELECT i.*
FROM   indexed AS i
ORDER BY i.c DESC
```

❸
```sql
SELECT i.*
FROM   indexed AS i
ORDER BY i.c, i.a
```

❹
```sql
SELECT i.*
FROM   indexed AS i
ORDER BY i.c DESC, i.a DESC
```

❺
```sql
SELECT i.*
FROM   indexed AS i
ORDER BY i.c ASC, i.a DESC
```

❻
```sql
SELECT i.*
FROM   indexed AS i
ORDER BY i.c
LIMIT 42 -- first 42 rows only
```

Demonstrate (non-)support of ORDER BY by Index Scan [Backward]:

```
-- ❶ supported (also show the value of pipelined "sort")

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c;
```

QUERY PLAN
Index Scan using indexed_c_a on public.indexed i (cost=0.42..67780.42 rows=1000000 width=41) (actual time=0.034..861.362 rows=1000000)
Output: a, b, c
Planning time: 0.427 ms   first row produced immediately
Execution time: 921.608 ms   ⇒ non-blocking/pipelined "sort"

```
  set enable_indexscan = off;

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c;
```

QUERY PLAN
Sort (cost=180533.84..183033.84 rows=1000000 width=41) (actual time=1482.843..1943.239 rows=1000000 loops=1)
Output: a, b, c
Sort Key: i.c   blocking "sort"
Sort Method: external merge Disk: 50976kB ⬅ on-disk sort :-(
-> Seq Scan on public.indexed i (cost=0.00..19346.00 rows=1000000 width=41) (actual time=0.023..110.028 rows=1000000 loops=1)
Output: a, b, c
Planning time: 0.150 ms
Execution time: 2031.111 ms

```
  set enable_indexscan = on;

-- ❷ supported
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c DESC;
```

QUERY PLAN

```
Index Scan Backward using indexed_c_a on public.indexed i  (cost=0.42..67780.42 rows=1000000) (actual time=0.028..971.297 rows=1000000)
  Output: a, b, c
Planning time: 0.147 ms
Execution time: 1043.549 ms
```

```
-- ❸ supported
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c, i.a;
```

QUERY PLAN
Index Scan using indexed_c_a on public.indexed i (cost=0.42..67780.42 rows=1000000 width=41) (actual time=0.029..813.745 rows=1000000)   Output: a, b, c Planning time: 0.119 ms Execution time: 870.899 ms

```
-- ❹ supported
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c DESC, i.a DESC;
```

QUERY PLAN
Index Scan Backward using indexed_c_a on public.indexed i (cost=0.42..67780.42 rows=1000000) (actual time=0.017..761.151 rows=1000000)   Output: a, b, c Planning time: 0.069 ms Execution time: 813.052 ms

```
-- ❺ not supported
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c ASC, i.a DESC;  -- ⬅ does not match row visit order in scan
```

QUERY PLAN

Sort (cost=180533.84..183033.84 rows=1000000 width=41) (actual time=3178.794..3657.175 rows=1000000 loops=1)
  Output: a, b, c
  Sort Key: i.c, i.a DESC

```
   Sort Method: external merge  Disk: 50976kB
   ->  Seq Scan on public.indexed i  (cost=0.00..19346.00 rows=1000000 width=41) (actual time=0.017..105.695 rows=1000000 loops=1)
         Output: a, b, c
 Planning time: 0.114 ms
 Execution time: 3732.085 ms
```

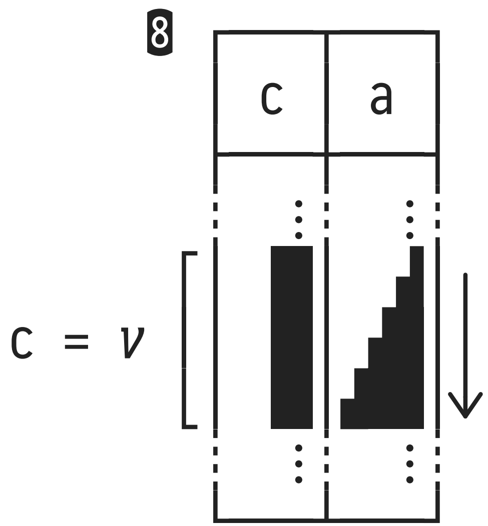
```
-- ❻ supported (also shows how Limit cuts off the Index Scan early → Volcano-style pipelining)
 EXPLAIN (VERBOSE, ANALYZE)
   SELECT i.*
   FROM   indexed AS i
   ORDER BY i.c
   LIMIT 42;
```

```
                                              QUERY PLAN
 Limit  (cost=0.42..3.27 rows=42 width=41) (actual time=0.033..0.090 rows=42 loops=1)
   Output: a, b, c
   Buffers: shared hit=16
   ->  Index Scan using indexed_c_a on public.indexed i  (cost=0.42..67780.42 rows=1000000) (actual time=0.031..0.072 rows=42)
         Output: a, b, c
         Buffers: shared hit=16                                                          cut off index scan after 42 rows
 Planning time: 0.134 ms                                                                 (deliver rows on demand)
 Execution time: 0.137 ms
```

Supporting ORDER BY With B+Trees?

```
7 SELECT i.*
  FROM   indexed AS i
  ORDER BY i.a
```

```
8 SELECT .*
  FROM   indexed AS i
  WHERE  i.c = 0.0
  ORDER BY i.a
```



- **N.B.:** A range predicate on c (*e.g.*, $c \leqslant v$) rules out index support again.
- In ❽, PostgreSQL implements filter i.c = 0.0 with a Bitmap Index Scan, then implements ORDER BY i.a using Sort.[2]

[2] Use set enable_sort = off or set enable_bitmapscan = off to see that PostgreSQL *can* be reasonable.

Demonstrate this more intricate case of query/index matching:

```
-- ❼ not supported
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.a;
```

QUERY PLAN
Sort (cost=180533.84..183033.84 rows=1000000 width=41) (actual time=1043.813..1200.235 rows=1000000 loops=1)
Output: a, b, c
Sort Key: i.a
Sort Method: external sort Disk: 50880kB
-> Seq Scan on public.indexed i (cost=0.00..19346.00 rows=1000000 width=41) (actual time=0.017..116.170 rows=1000000 loops=1)
Output: a, b, c
Planning time: 0.102 ms
Execution time: 1261.513 ms

```
-- ❽ not really supported but could be supported just fine (supports predicate but not the ORDER BY clause)
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    WHERE  i.c = 0.0
    ORDER BY i.a;
```

QUERY PLAN
Sort (cost=7466.93..7476.38 rows=3779 width=41) (actual time=9.745..10.008 rows=3198 loops=1)
Output: a, b, c
Sort Key: i.a
Sort Method: quicksort Memory: 346kB
-> Bitmap Heap Scan on public.indexed i (cost=89.71..7242.38 rows=3779 width=41) (actual time=3.948..8.877 rows=3198 loops=1)
Output: a, b, c
⁂ Recheck Cond: (i.c = 0.0)
Heap Blocks: exact=2896
-> Bitmap Index Scan on indexed_c_a (cost=0.00..88.77 rows=3779 width=0) (actual time=2.642..2.642 rows=3198 loops=1)
Index Cond: (i.c = 0.0)
Planning time: 0.167 ms   ⁂ choice of Bitmap Index Scan will let rows appear out of (any) order :-(
Execution time: 10.293 ms

```
set enable_bitmapscan = off;  -- ◀ force the system into using Index Scan (to produce rows in a-sorted order)
```

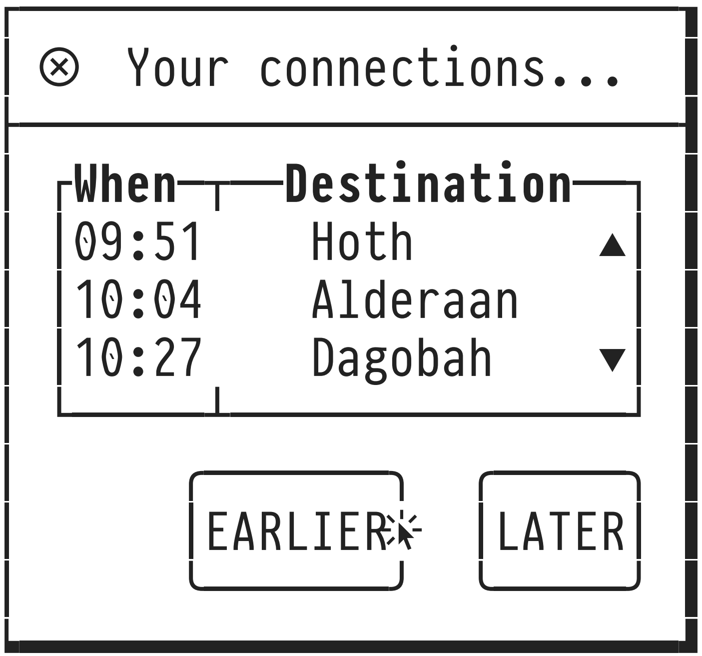
```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT i.*
  FROM   indexed AS i
  WHERE  i.c = 0.0
  ORDER BY i.a;
```

QUERY PLAN
Index Scan using indexed_c_a on public.indexed i (cost=0.42..12702.56 rows=3779 width=41) (actual time=0.071..7.214 rows=3198)
Output: a, b, c
Index Cond: (i.c = 0.0) ... is wrongly estimated to be more expensive
Planning time: 0.164 ms (see cost=7476.38) in Sort plan above)
Execution time: 7.627 ms ← evaluates faster, BUT ...

```
set enable_bitmapscan = on;
```

10 ¦ Use Case: Paging Through Table Contents

id	when	destination
1	09:51	Tatooine
2	09:51	Hoth
3	10:04	Alderaan
4	10:27	Dagobah



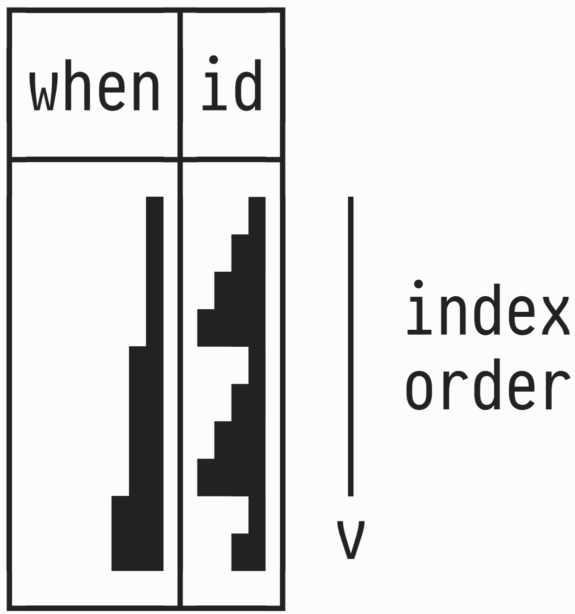
- Efficiently **page** through a large table or query result. Show *n* rows at a time.

- Do not cache large table in UI (think Web browser), instead request required window of *n* rows from the DB server on demand.

Indexing for Efficient “when”-Based Paging

id	when	destination
1	09:51	Tatooine
2	09:51	Hoth
3	10:04	Alderaan
4	10:27	Dagobah

```
CREATE TABLE connections (
  id          int PRIMARY KEY,
  "when"      timestamp,
  destination text
);
CREATE INDEX connections_when_id
  ON connections("when", id);
```

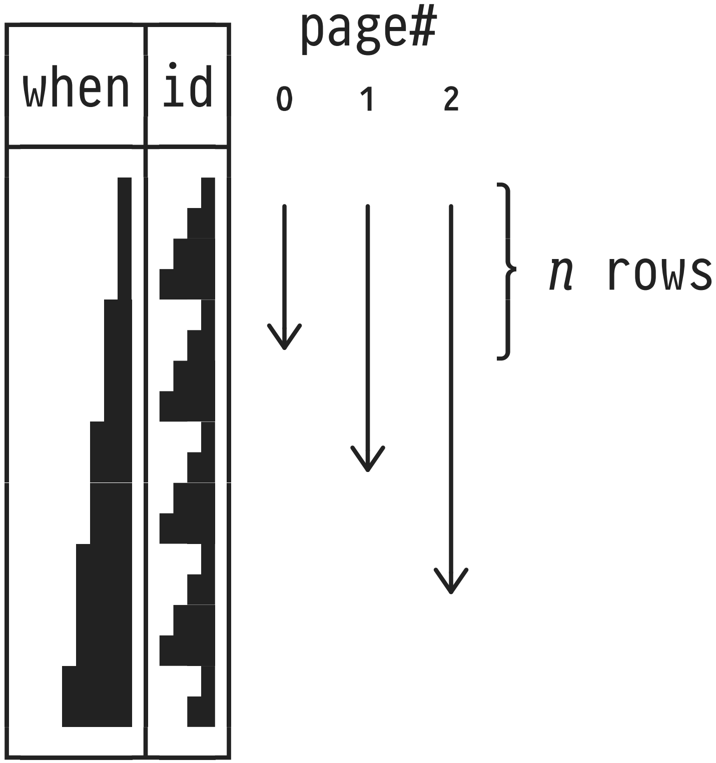


```
Index on when first, since this will be the primary paging criterion.
```

Option 1: Using OFFSET and LIMIT

Parameters: *:page* ∈ {0,…} current page, *:n* rows per page.

```
SELECT c.*
FROM   connections AS c
ORDER BY c."when"
OFFSET :page * :n
LIMIT  :n
```



- The further we page, the wider becomes the index scan range.

⇒ Paging gets slower and slower.

Demonstrate the slowdown of paging when using the OFFSET/LIMIT method:

```
-- 1 Set up connections table and its index

  \set rows_per_page  10

  DROP TABLE IF EXISTS connections;
  CREATE TABLE connections (
    id          int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    "when"      timestamp,
    destination text
  );

  INSERT INTO connections ("when", destination)
    SELECT now() + make_interval(mins => i) AS "when",
           md5(i :: text) AS destination
    FROM   generate_series(1, 10000) AS i;

  CREATE INDEX connections_when_id
    ON connections USING btree ("when", id);
  ANALYZE connections;


-- 2 Browse pages, starting from 0

  \set page 0

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT c.*
    FROM   connections AS c
    ORDER BY c."when"
    OFFSET :page * :rows_per_page
    LIMIT  :rows_per_page;
```

QUERY PLAN
Limit (cost=0.29..1.01 rows=10 width=45) (actual time=0.018..0.033 rows=10 loops=1)
Output: id, "when", destination
-> Index Scan using connections_when_id on connections c (cost=... rows=10000) (actual time=0.017..0.031 rows=10)
Output: id, "when", destination index scan produces 10 rows only
Planning time: 0.117 ms
Execution time: 0.058 ms ⬅ fast browsing of page 0

```
  \set page 900
```

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT c.*
  FROM   connections AS c
  ORDER BY c."when"
  OFFSET :page * :rows_per_page
  LIMIT  :rows_per_page;
```

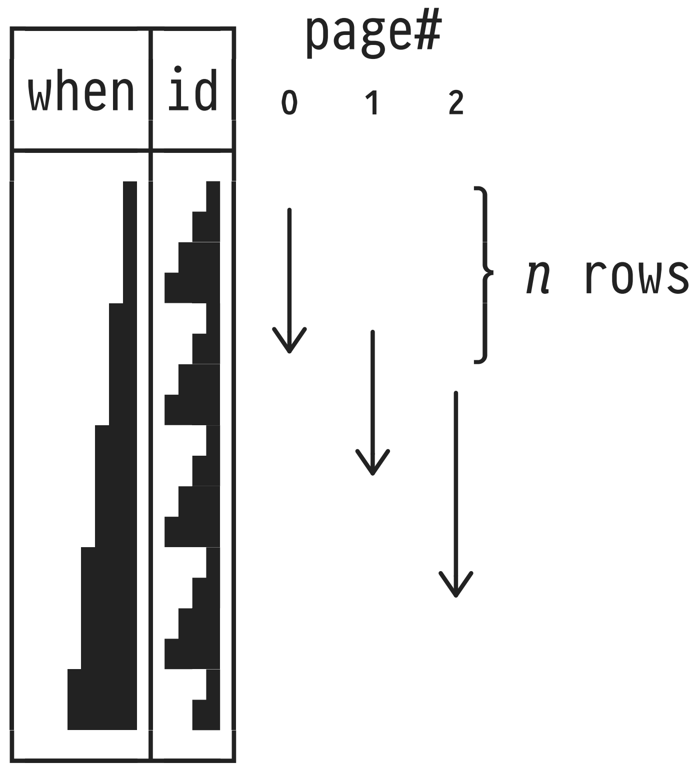
QUERY PLAN
Limit (cost=656.57..657.30 rows=10 width=45) (actual time=10.444..10.452 rows=10 loops=1)
Output: id, "when", destination
-> Index Scan using connections_when_id on connections c (cost=... rows=10000) (actual time=0.021..9.750 rows=9010)
Output: id, "when", destination
Planning time: 0.255 ms   wide index scan to skip previous pages
Execution time: 10.482 ms ⬅ slow browsing of page 900

Option 2: Using WHERE and LIMIT

```
SELECT c.*
FROM   connections AS c
WHERE (c."when",c.id) < (:last_when,:last_id)
ORDER BY c."when", c.id
LIMIT  :n
```

- Save index keys *:last_when*, *:last_id* of last entry displayed. Pass these to RDBMS when we request next page (continue "interrupted" index scan).

⇒ Paging speed independent of page #.



Demonstrate constant-speed paging when using the WHERE/LIMIT method:

```
  \set rows_per_page  10

-- 1 determine last connection (we start paging from here)

  -- start browsing at the last connection (≡ page #0)
  SELECT c."when", c.id
  FROM   connections AS c
  ORDER BY c."when", c.id
  LIMIT 1;

  --  sets :last_when, :last_id
  \gset last_


-- 2 produce one page of connections

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT c.*
    FROM   connections AS c
    WHERE  (c."when", c.id) <= (:'last_when', :last_id)
    ORDER BY c."when", c.id  --  ⬅ ORDER BY spec matches index scan order
    LIMIT  :rows_per_page;
```

QUERY PLAN

Limit (cost=0.29..1.04 rows=10 width=45) (actual time=0.010..0.031 rows=10 loops=1)
  Output: id, "when", destination
  -> **Index Scan using** connections_when_id **on** connections c (**cost**=... **rows**=10000) (actual time=0.009..0.028 **rows**=10)
    ⬆ **Output: id,** "when", destination ⬆
    Index Cond: (ROW(c."when", c.id) <= ROW('23:59:12.597454'::time **without** time zone, 10000)) **scan** 10 **rows only**
Planning time: 0.144 ms
Execution time: 0.058 ms ⬅ **fast** browsing **of** page 0

```
-- equivalent WHERE clause (no row comparison):
--   WHERE  c."when" <= :'last_when' OR (c."when" = :'last_when' AND c.id <= :last_id)


-- 3 produce next page of connections (can repeat this query to continue paging)

  EXPLAIN (VERBOSE, ANALYZE)
    WITH connections_page(id, "when", destination) AS (
      SELECT c.*                                           -- original paging query
```

```
    FROM   connections AS c                              --
    WHERE  (c."when", c.id) <= (:'last_when', :last_id)  --
    ORDER BY c."when", c.id                              --
    LIMIT  :rows_per_page                                --
  )
  SELECT c."when", c.id                                  -- extract last row displayed on
  FROM   connections_page AS c                           -- last page to continue paging
  ORDER BY c."when" DESC, c.id DESC                      -- from that row later on
  LIMIT 1;                                               --

-- update :last_when, :last_id based on largest row (latest connection)
-- displayed in last page, can now re-invoke WITH CTE to continue paging
-- (⚠ to make \gset work, do not use EXPLAIN in WITH above)
\gset last_
```

```
                                QUERY PLAN

Limit  (cost=1.12..1.12 rows=1 width=12) (actual time=0.039..0.039 rows=1 loops=1)
  Output: c."when", c.id
  CTE connections_page

  RELEVANT SUB-PLAN BELOW (execution time remains the same regardless of number of pages browsed):

->  Limit  (cost=0.29..0.87 rows=10 width=45) (actual time=0.013..0.015 rows=10 loops=1)
      Output: c_1.id, c_1."when", c_1.destination
      ->  Index Scan using connections_when_id on connections c_1  (cost=... rows=9982) (actual time=0.012..0.012 rows=10)
          Output: c_1.id, c_1."when", c_1.destination
          Index Cond: (ROW(c_1."when", c_1.id) <= ROW('2018-06-13 14:08:16.484694'::timestamp, 9982))    10 rows scanned


  ->  Sort  (cost=0.25..0.28 rows=10 width=12) (actual time=0.038..0.038 rows=1 loops=1)
        Output: c."when", c.id
        Sort Key: c."when" DESC, c.id DESC
        Sort Method: top-N heapsort  Memory: 25kB
        ->  CTE Scan on connections_page c  (cost=0.00..0.20 rows=10 width=12) (actual time=0.015..0.024 rows=10 loops=1)
              Output: c."when", c.id
Planning time: 0.188 ms
Execution time: 0.078 ms ⬅ browsing remains fast
```