

DB 2

10 – Matching Queries and Indexes

Summer 2020

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?



2 | Indexes on Expressions

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)
```

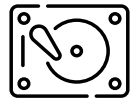


Indexes on Expressions

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 $i.c = \text{█}^{-1}(v)$:
 - █ may be complex and/or user-defined and the inverse might be hard to find for the system.
 - █ 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 ( $e$ )
```

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 $e \theta v$.
- The sequence set of index I is ordered by e .
- **CREATE UNIQUE INDEX ...**: can protect complex constraints.



Indexes on Expressions

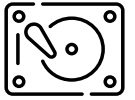
Consider expression-based index `people_age` on the user-defined SQL function (UDF) `get_age()`:

```
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?




3 | Composite (or: Concatenated) Indexes

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

```
CREATE INDEX  $I$  ON  $T$  USING btree ( $c_1, \dots, c_n$ )
```

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

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

-  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) ?



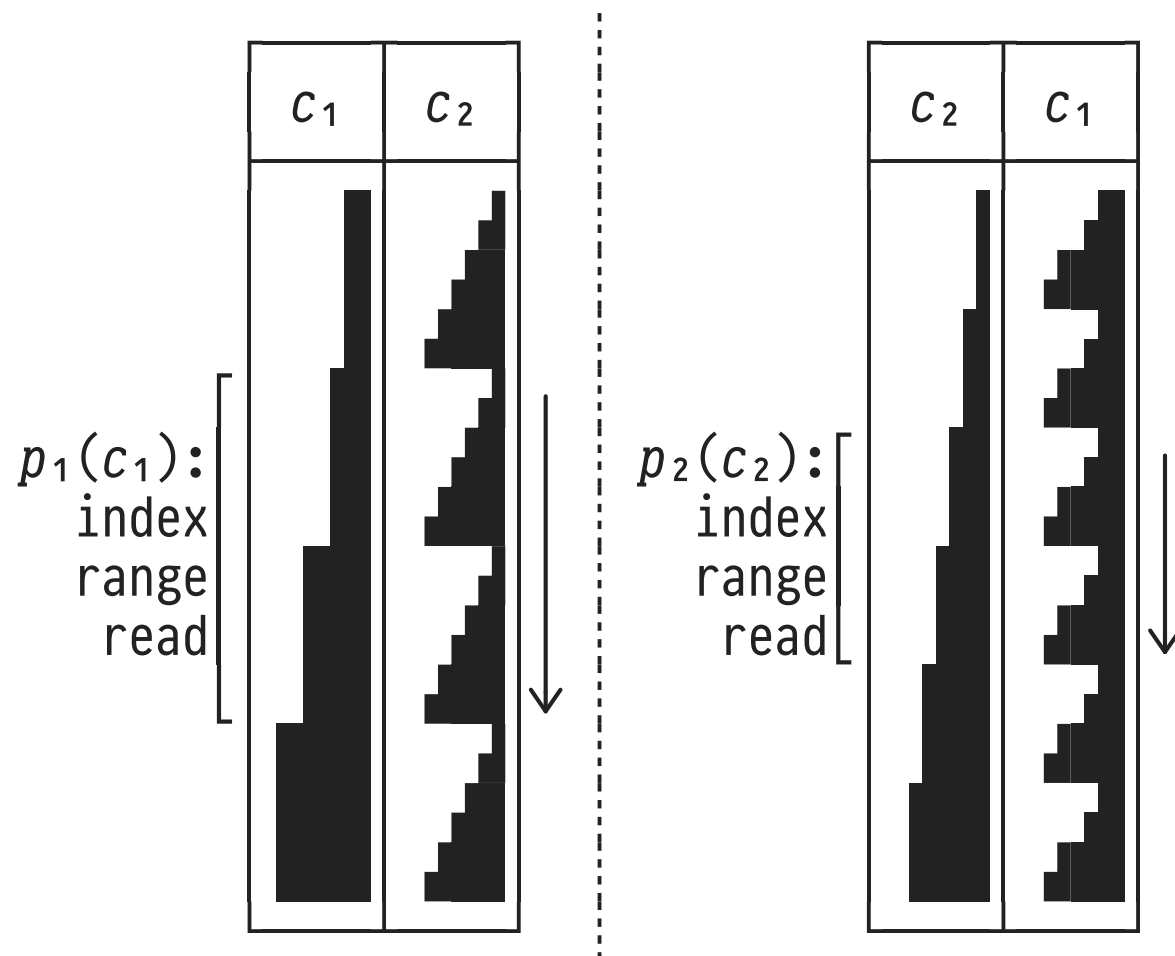
Multi-Dimensional Queries and Composite Indexes

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

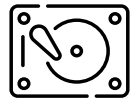
```
SELECT e(t)
FROM   T AS t
WHERE  p1(t.c1)  -- two dimensions:
      AND p2(t.c2)  -- c1, c2
```

- 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?

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!”



Partitioned B+Trees: SQL Code

Simple implementation of bulk appends and delayed merging:

```
-- ❶ 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);

-- ❷+❸ Fast bulk inserts (simply appends to B+Tree I)
INSERT INTO T(p,...) SELECT 1, ... FROM ...;
INSERT INTO T(p,...) SELECT 2, ... FROM ...;

-- ❹ 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
```

5 : Multi-Dimensional Predicates and Index Combinations



Consider a SQL query with a *disjunctive* predicate:

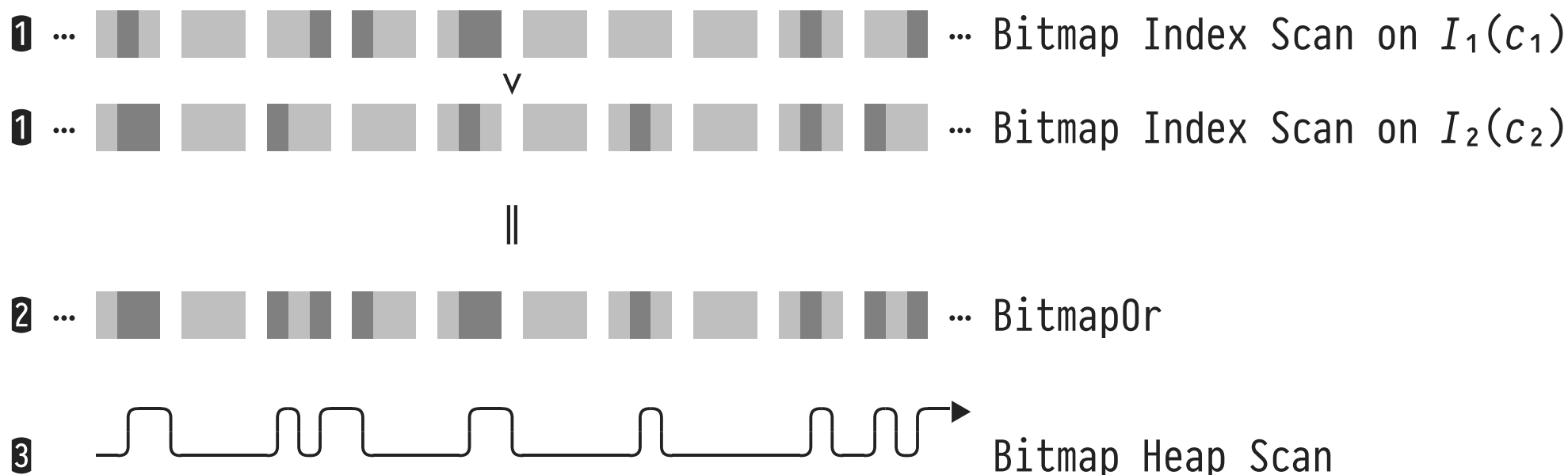
```
SELECT e(t)
FROM   T AS t
WHERE  p1(t.c1)  -- } disjunctive
      OR p2(t.c2)  -- } predicate
```

- 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**.

Combining Indexes via Bitmap Heap Scan and BitmapOr/And



- ❶ Perform individual **Bitmap Index Scans**, possibly in //, possibly multiple times on the same index.
- ❷ Combine resulting row-/page-level bitmaps using \vee or \wedge .
- ❸ Perform **Bitmap Heap Scan** with combined bitmap.





6 | String Pattern Matching (**LIKE**) and Indexes

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

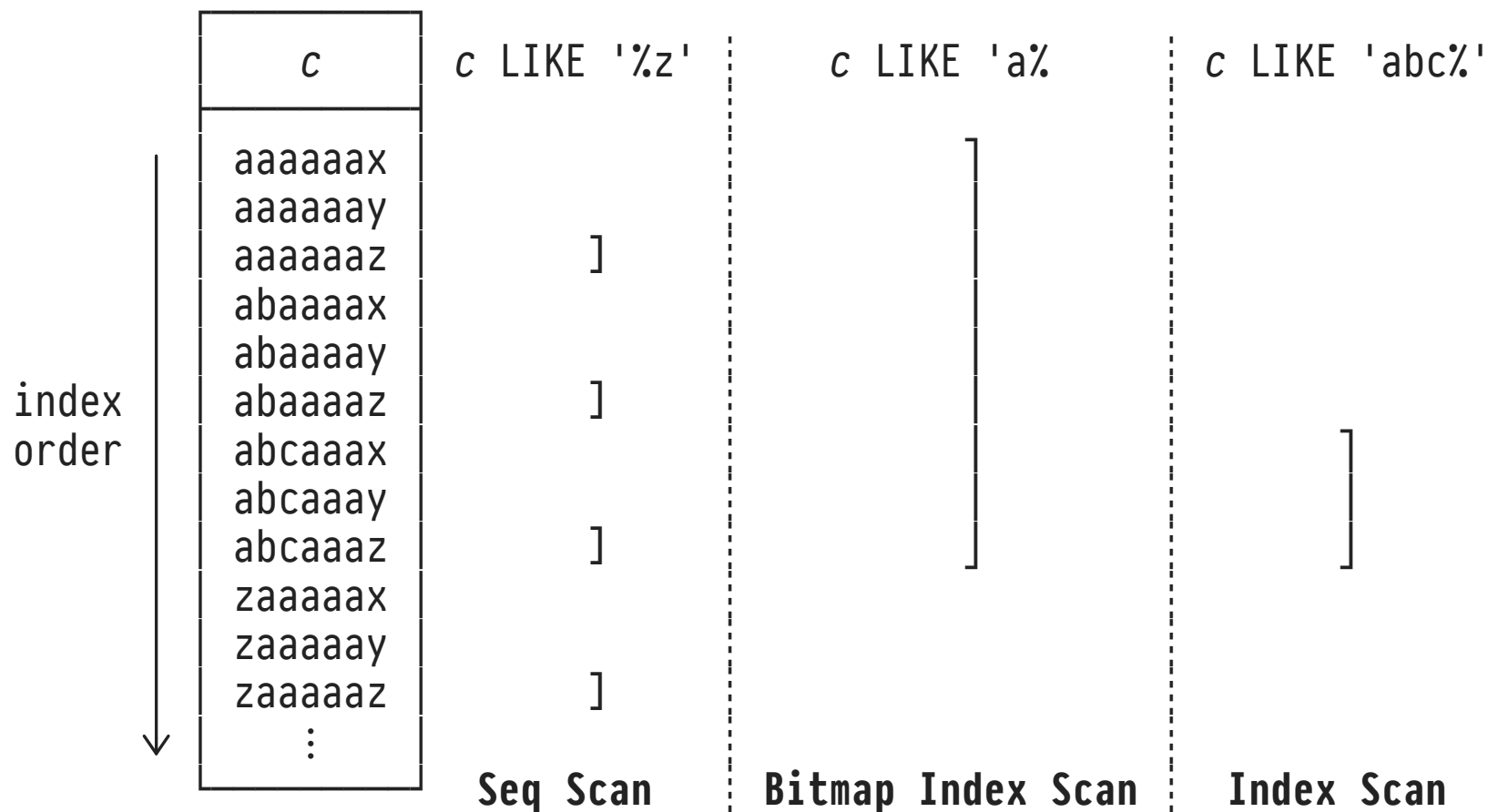
- SQL pattern matching: $e \text{ LIKE } 's_1\%s_2'$ holds iff string e contains substrings $s_{1,2}$ separated by zero or more arbitrary characters (regular expressions: $'s_1.*s_2'$).
- PostgreSQL: B+Tree index on column $c :: \text{text}$ of table T :

```
-- 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:





7 : Partial Indexes: Hot vs. Cold Rows

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

- Table **orders**

id	...	fulfilled
42		□
41		□
39		□
40	2020-06-03	
38	2020-05-27	
⋮	⋮	
2	2019-08-26	
1	2019-04-10	

“hot” open orders
 closed orders only used in reporting

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

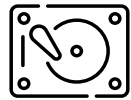
💡 Build **partial index** that covers the hot row subset only:

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

- *I* will be small: only rows of *T* satisfying *p* are 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.



8 | Index-Only Query Evaluation

For some queries, **all columns** C_1, \dots, 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 indexes¹ with keys $C_1, \dots, C_k, C_{k+1}, \dots, C_n$, in which
 - prefix C_1, \dots, C_k is used to guide index search (*i.e.*, to evaluate predicates),
 - suffix C_{k+1}, \dots, C_n is used to evaluate other expressions.

¹ PostgreSQL \geq v11: `CREATE INDEX I on T USING btree (C1, ..., Ck) INCLUDE (Ck+1, ..., Cn)`, builds a B+Tree in which keys (C_1, \dots, C_k) are narrow and only the leaves carry all columns C_1, \dots, C_n .



Index-Only Queries?

Assume B+Tree index (a,c) on table `indexed`. Q: Can ❶...❷ 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;



Index-Only Scans and Row Visibility

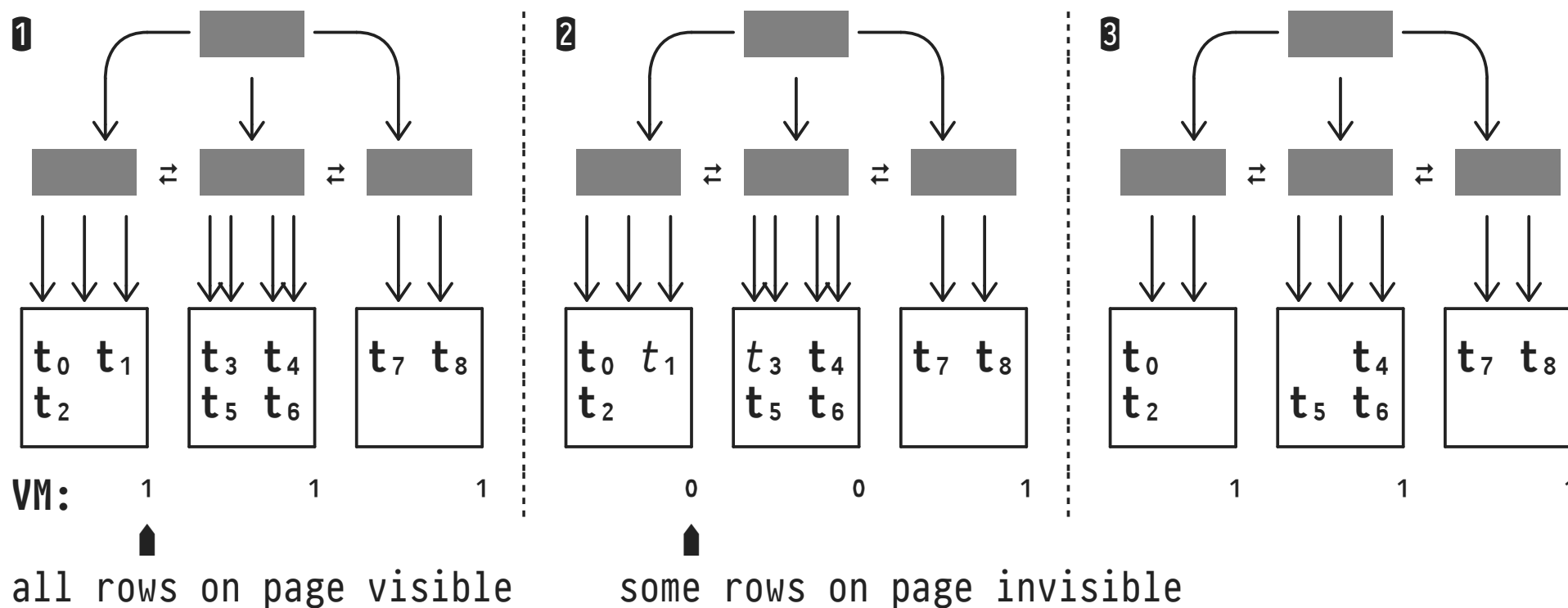
Index-only query evaluation—implemented by plan operator **Index Only Scan**—in PostgreSQL faces a challenge:

- 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.



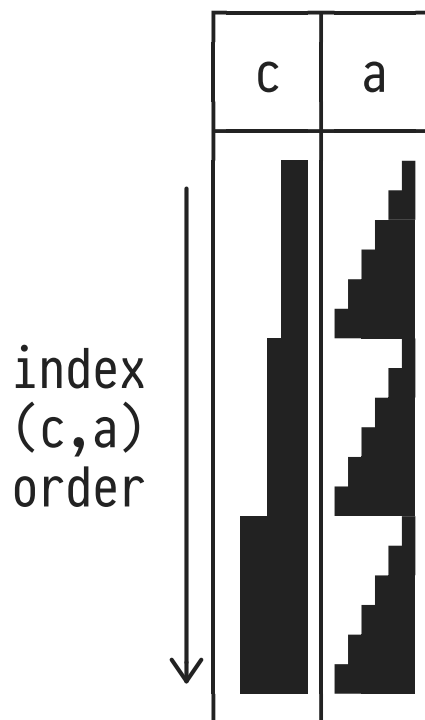
Index-Only Scans and the Visibility Map (VM)

- ❶ Original table state (t_i : visible row).
- ❷ After deletion of t_1 and t_3 (t_i : invisible row).
- ❸ After **VACUUM**: dead rows removed, index updated.





9 : Supporting More Query Types With B+Trees



- 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.
 - During an index scan, we will encounter rows *as if* they had been sorted by `ORDER BY c ASC, a ASC` (see left).



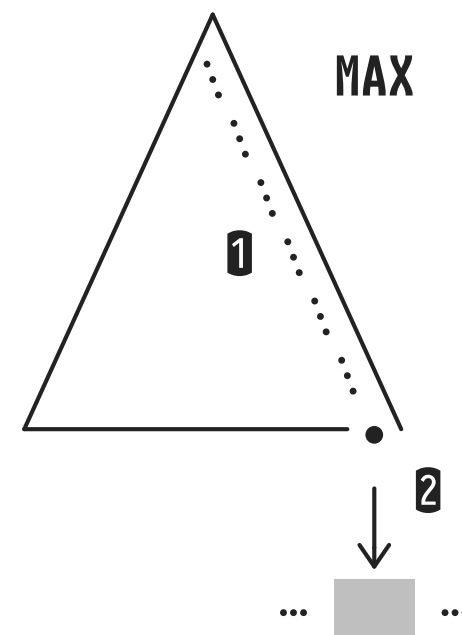
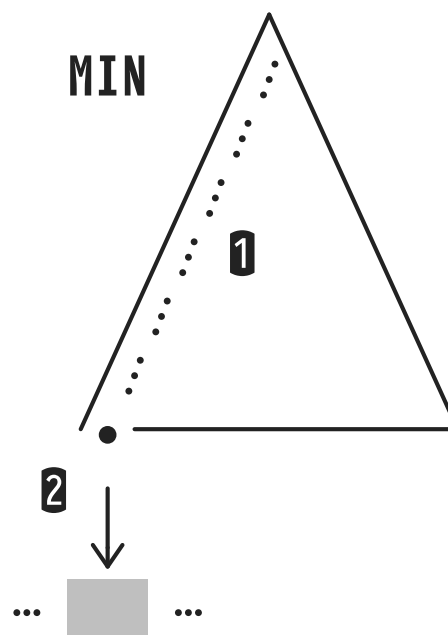
Supporting **MIN/MAX** With B+Trees

```
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?





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

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

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

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

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

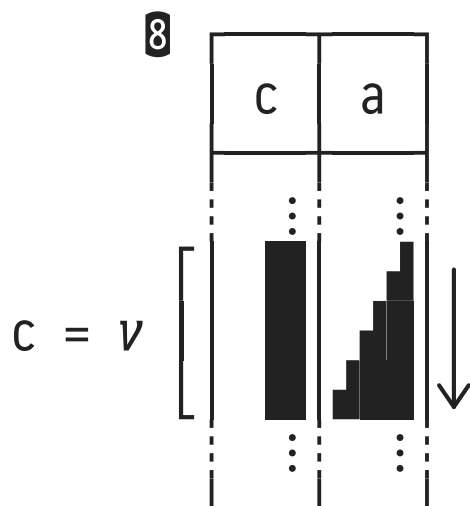
```
❻ SELECT i.*  
   FROM indexed AS i  
  ORDER BY i.c  
 LIMIT 42 -- first 42 rows only
```



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
```



- **NB.** A range predicate on **c** (e.g., $c \leq v$) rules out index support again.
- In 8, 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.



10 : Use Case: Paging Through Table Contents

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

⊗ Your connections...

When	Destination
09:51	Hoth ▲
10:04	Alderaan
10:27	Dagobah ▼

EARLIER*

LATER

- 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.



Option 1: Using **OFFSET** and **LIMIT**

Parameters: **:page** $\in \{0, \dots\}$ 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.

