DB 2

09 – Ordered Indexes (B⁺Trees)

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 ¦ Q_8 — Filtering an Indexed Table

Sequential scan (**Seq Scan**) and interpreted predicate evaluation go a long way. Large input tables call for significantly more **efficient support for value-based row access:**

```
SELECT i.b, i.c
FROM   indexed AS i
WHERE  i.a = 42 [i.c = 0.42] -- either filter on i.a or i.c
```

Assume column a is **primary key** in table indexed: expect query workload that frequently identifies rows via predicates a = k. **Indexes** can support such queries.

Primary Key and Indexes

```
CREATE TABLE indexed (a int PRIMARY KEY, -- ⇒ NOT NULL
                      b text,
                      c numeric(3,2));   -- ± d.dd
```

DBMS expects predicates a = *k* and creates an **index on column a**—a data structure associated with and maintained in addition to table indexed—to speed up evaluation:

```
CREATE INDEX indexed_a ON indexed USING btree (a);
```

1. Whenever possible/promising, index indexed_a[1] is (also) consulted when table indexed is queried. 🚀
2. When indexed is updated, indexed_a is maintained. ⚙

[1] PostgreSQL chooses index name indexed_pkey but let's follow a *‹table›*_*‹column›* naming scheme here.

Create table indexed, check for presence of index (automatically created), rename primary key index to indexed_a to follow lecture's naming scheme:

```
DROP TABLE IF EXISTS indexed;
CREATE TABLE indexed (a int PRIMARY KEY, b text, c numeric(3,2));
                            ▲
INSERT INTO indexed(a,b,c)               -- updates table AND ALSO updates any index
  SELECT i, md5(i::text), sin(i)
  FROM   generate_series(1,1000000) AS i;

\d indexed
```

Table "public.indexed"

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

Indexes:

```
    "indexed_pkey" PRIMARY KEY, btree (a) ◀

ALTER INDEX indexed_pkey RENAME TO indexed_a;
```

This index is an additional data structure maintained by the DBMS. Persistently resides on secondary storage. Typically *much smaller* than the original source data it indexes:

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

relname	relfilenode	relpages	reltuples	relkind
indexed indexed_a	113411 113417	9346 2745	1e+06 1e+06	r i ◀

```
                          ▲              ▲
               data >> index      1M rows in the table, 1M entries in the index:
                                  dense index
```

Using EXPLAIN on Q_8

```
EXPLAIN VERBOSE
  SELECT i.b, i.c
  FROM   indexed AS i    -- 10^6 rows
  WHERE  i.a = 42;       -- selection on key column a ⇒ ⩽ 1 row will qualify
```

```
                              QUERY PLAN
─────────────────────────────────────────────────────────────────────────
 Index Scan using indexed_a on indexed i  (cost=0.42..8.44 rows=1 …)
   Output: b, c                                                  ▲
   Index Cond: (i.a = 42) ◀
```

- DBMS uses **Index Scan** (instead of **Seq Scan**), index scan will evaluate predicate i.a = *k*.
- System expects small result of a single row (rows=1), i.e., the predicate is assumed to be *very selective*.

The performance impact of **Index Scan** is significant. To demonstrate this, temporarily disable index scan (and bitmap scan). Measure query performance before/after using EXPLAIN ANALYZE.

```
-- ❶ Q8 with index support enabled

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

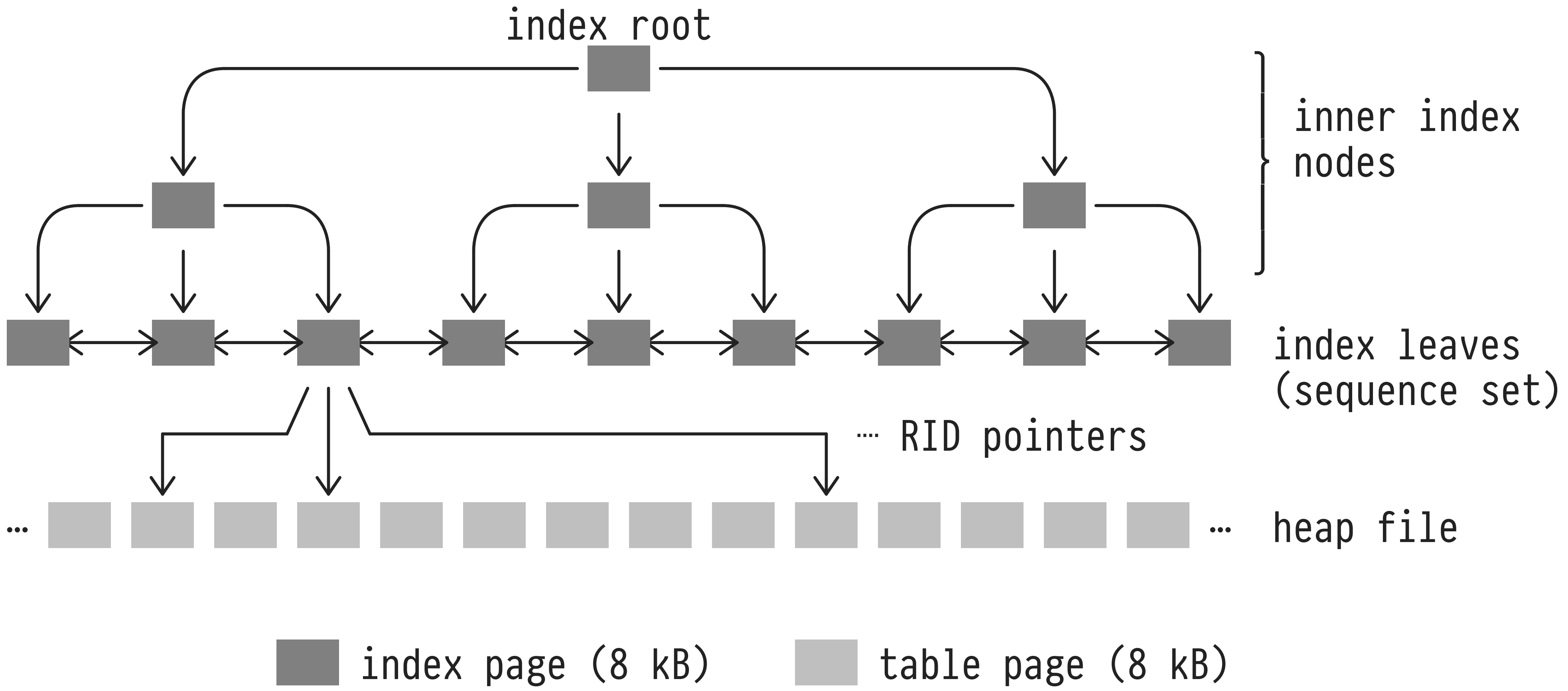
```
                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
Index Scan using indexed_a on public.indexed i  (cost=0.42..8.44 rows=1 width=37) (actual time=0.039..0.041 rows=1 loops=1)
  Output: b, c
  Index Cond: (i.a = 42)
Planning time: 0.129 ms
Execution time: 0.091 ms ⬅
```

```
-- ❷ Temporarily disable index support
  set enable_indexscan = off;
  set enable_bitmapscan = off;


-- ❸ Reevaluate query Q8 without index support

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

```
                                        QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
Seq Scan on public.indexed i  (cost=0.00..21846.00 rows=1 width=37) (actual time=0.045..170.111 rows=1 loops=1)
  Output: b, c
  Filter: (i.a = 42)
  Rows Removed by Filter: 999999
Planning time: 0.124 ms
Execution time: 170.145 ms ⬅
```

```
-- ❹ Re-enable index support
  set enable_indexscan = on;
  set enable_bitmapscan = on;
```

2 ¦ B$^+$Trees: Ordered Indexes



Anatomy of a B$^+$Tree

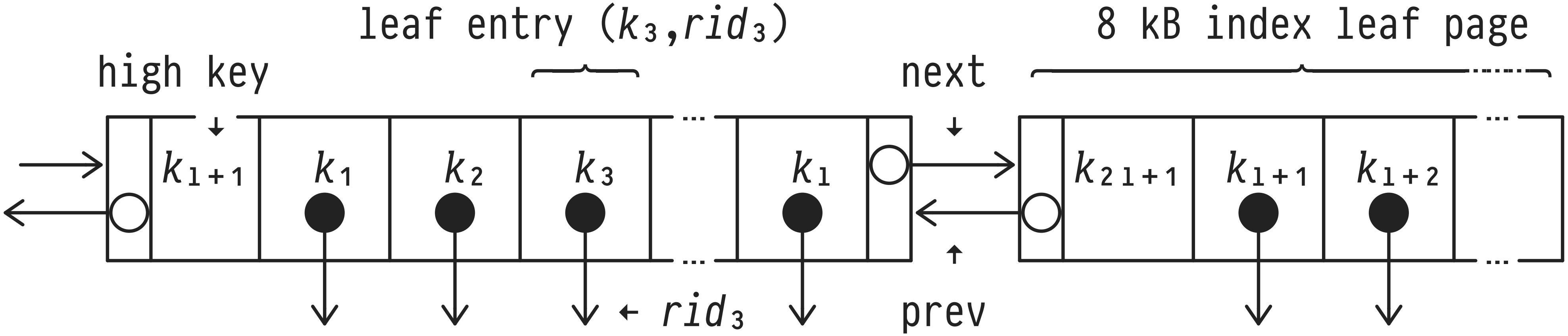
B$^+$Trees: Ordered Indexes

Notes on B$^+$Tree anatomy:

- A **B$^+$Tree[2] index** I **on column** $T(a)$ is an *ordered*, *n-ary* (n » 2), *balanced*, *block-oriented*, *dynamic* search tree.
- Inner nodes and leaves are formed by 8 kB **index pages.**
- Each inner node holds $n-1$ values of column a (**separators**) that allow to navigate the search tree structure.
- Leaves form a bidirectional chain, the **sequence set.**
- **Leaves use RIDs to point to rows** in the heap file of table T: besides a column values, I holds no data of T.

[2] Invented by Bayer and McCreight (1969) at Boeing Labs. The “B” in “B$^+$Tree” does *not* stand for Bayer, binary, balanced, block, or Boeing. (We tried to find out, but Rudolf Bayer wouldn't say.)

B⁺Trees: Inside a Leaf Node



- Uses pointers prev/next to form the chained **sequence set.**
- **Leaf entries are ordered** by index keys k_i: $k_i \leqslant k_{i+1}$.
- RID rid_i points to a row t of T with $t.a = k_i$.
- The **high key** holds smallest key of *next* leaf (if any).

Use PostgreSQL extension pageinspect to dump the contents of the leaf nodes of index indexed_a on column indexed(a):

- bt_page_stats(‹index›, ‹p›) – details for index page ‹p›

```
-- ❶ How many pages are there in `indexed_a` overall?

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

relname	relfilenode	relpages	reltuples	relkind
indexed indexed_a	113411 113417	9346 2745 ◀	1e+06 1e+06	r i

- The index has pages #0...#2744. Page #0 is the index' special page carrying meta data.

```
-- ❷ Visit all index pages, dump only the leave node pages

-- Switch to expanded row display
  \x

  SELECT node.*
  FROM   generate_series(1, 2744) AS p,
         LATERAL bt_page_stats('indexed_a', p) AS node
  WHERE  node.type = 'l'  -- l ≡ leaf, i ≡ inner, r ≡ root
  ORDER BY node.blkno;
┌[ RECORD 1 ]───┬──────┐
│ blkno         │ 1    ◀  this is page #1
│ type          │ l    ◀  leaf node
│ live_items    │ 367  ◀  # of leaf entries (actually 366: includes high key)
│ dead_items    │ 0    │
│ avg_item_size │ 16   ◀  size of leaf entry in bytes
│ page_size     │ 8192 ◀  index page size
│ free_size     │ 808  │
│ btpo_prev     │ 0    ◀  leftmost leaf (no previous node)
│ btpo_next     │ 2    ◀  page #2 next in sequence set
│ btpo          │ 0    │          │
│ btpo_flags    │ 1    │          │
├[ RECORD 2 ]───┼──────┤          │
│ blkno         │ 2    │ ◀────────┘
│ type          │ l    │
│ live_items    │ 367  │
│ dead_items    │ 0    │
│ avg_item_size │ 16   │
│ page_size     │ 8192 │
│ free_size     │ 808  │
│ btpo_prev     │ 1    │
│ btpo_next     │ 4    │
│ btpo          │ 0    ◀  tree level of node (0 ≡ leaf)
│ btpo_flags    │ 1    │
├─[ RECORD 3 ]──┼──────┤
│ blkno         │ 4    │
│ type          │ l    │
│ live_items    │ 367  │
│ dead_items    │ 0    │
│ avg_item_size │ 16   │
│ page_size     │ 8192 │
│ free_size     │ 808  │
│ btpo_prev     │ 2    │
│ btpo_next     │ 5    │
│ btpo          │ 0    │
│ btpo_flags    │ 1    │
├─[ RECORD 4 ]──┼──────┤
[...]
(2733 rows)
```

- Index pages are of size 8192 bytes. Approximately 366 * 16 bytes = 5856 bytes (or ≈ 71%) of pages are occupied.
- All but the rightmost pages on each tree level contain one high key entry (included in live_items).

```
-- ❸ Recursively walk the sequence set chain and extract the
      number of index entries found in each leaf (subtract 1
      from live_items for all pages but the rightmost page)

  WITH RECURSIVE sequence_set(leaf, next, entries) AS (
    SELECT node.blkno          AS leaf,
           node.btpo_next      AS next,
           node.live_items - (node.btpo_next <> 0)::int AS entries  -- node.btpo_next <> 0 ≡ node is not rightmost on tree level
    FROM   pg_class AS c,
           LATERAL generate_series(1, c.relpages-1) AS p,
           LATERAL bt_page_stats('indexed_a', p) AS node
    WHERE  c.relname = 'indexed_a' AND c.relkind = 'i'
    AND    node.type = 'l' AND node.btpo_prev = 0

     UNION ALL

    SELECT node.blkno          AS leaf,
           node.btpo_next      AS next,
           node.live_items - (node.btpo_next <> 0)::int AS entries
    FROM   sequence_set AS s,
           LATERAL bt_page_stats('indexed_a', s.next) AS node
    WHERE  s.next <> 0
  )
  -- TABLE sequence_set
  SELECT SUM(s.entries) AS entries
  FROM   sequence_set AS s;
```

entries
1000000

$\equiv 10^6 = |\text{indexed}|$

Now list the leaf entries on page #1 of index indexed_a:

- bt_page_items(‹index›, ‹p›) — leaf entries for index page ‹p›

We expect the following:

1. For two leaf entries, (k_1, rid_2) and (k_2, rid_2) we expect $k_1 \leqslant k_2$ if the first entry precedes the second in the sequence set.
2. For a leaf entry (k, rid), we expect the record pointed to by RID rid to have an a-value of k.

```
-- ❶ Access leaf entries on page #1 (a leaf page, see above) of indexed_a:

  SELECT *
  FROM   bt_page_items('indexed_a', 1);
```

```
| itemoffset | ctid      | itemlen | nulls | vars | data                    |
|------------|-----------|---------|-------|------|-------------------------|
|          1 | (3,46) ◀★ |      16 | f     | f    | 6f 01 00 00 00 00 00 00 | ◀ 0x016f = 367: high key ≡ lowest a-value on next leaf page
|          2 | (0,1)     |      16 | f     | f    | 01 00 00 00 00 00 00 00 | ◀ 0x0001 = 1: key-value k₁
|          3 | (0,2)     |      16 | f     | f    | 02 00 00 00 00 00 00 00 | ◀ 0x0002 = 2: key-value k₂
|          4 | (0,3)     |      16 | f     | f    | 03 00 00 00 00 00 00 00 |
|          5 | (0,4)     |      16 | f     | f    | 04 00 00 00 00 00 00 00 |
|          6 | (0,5)     |      16 | f     | f    | 05 00 00 00 00 00 00 00 |
[...]
|        366 | (3,44)    |      16 | f     | f    | 6d 01 00 00 00 00 00 00 | ◀ 0x016d = 365: key-value k₃₆₅
|        367 | (3,45)    |      16 | f     | f    | 6e 01 00 00 00 00 00 00 | ◀ 0x016e = 366: key-value k₃₆₆
                    ▲                                          ▲
                RIDs ridᵢ                               key values kᵢ

★: ignore RID here (only the high key is relevant)
```

1. Key values k_i are ascending as expected.
2. Follow RID (3,44) to check whether the row in table indexed carries the expected key value k_{365} = 365:

```
  SELECT *
  FROM   indexed AS i WHERE i.ctid = '(3,44)';
```

```
| a   | b                                | c    |
|-----|----------------------------------|------|
| 365 | 9be40cee5b0eee1462c82c6964087ff9 | 0.54 |
   ▲
  \o/
```

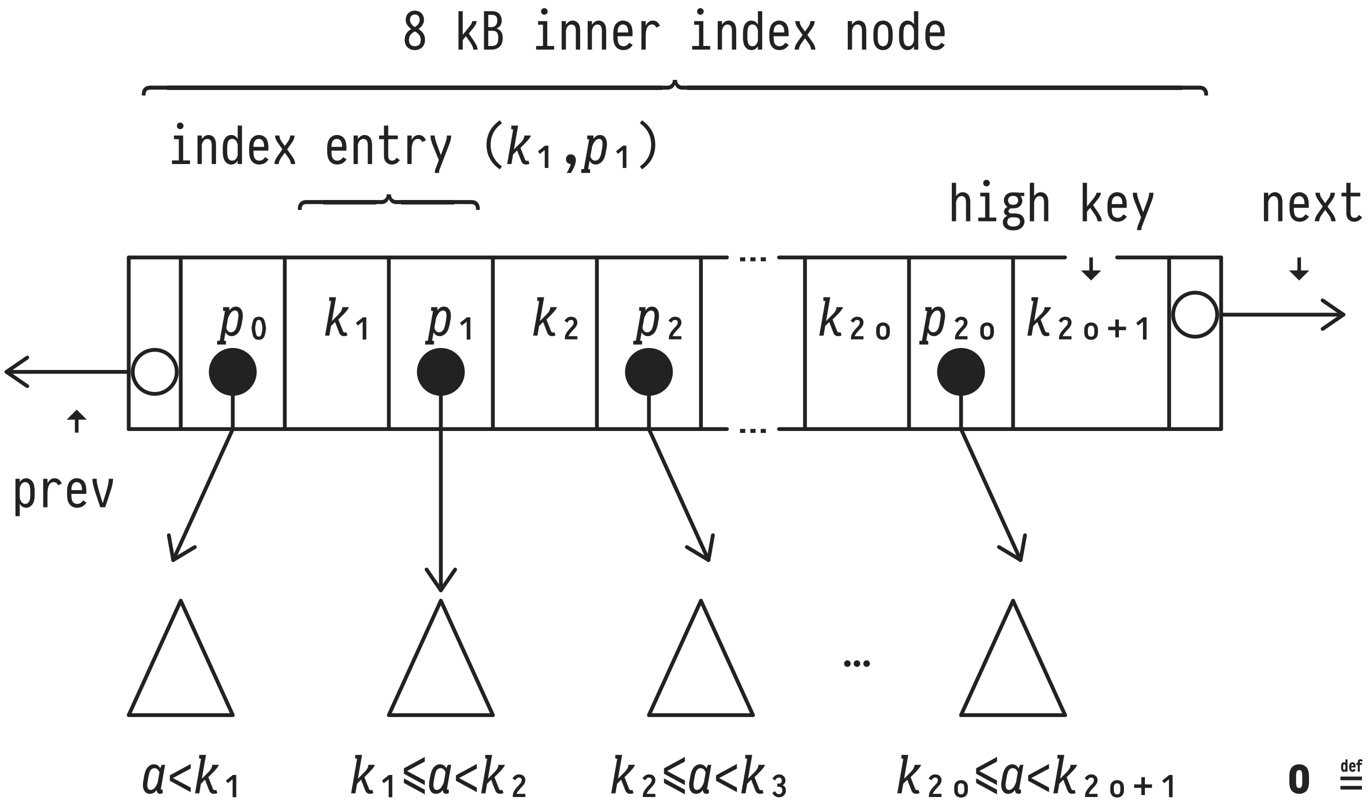
B$^+$Trees: How to Find Rows t With $t.a = k$?

As described, a B$^+$Tree is a **dense** index structure: every row t of T is represented by one leaf entry.

- The sequence set is ordered by keys k_i ⇒ a *binary search* for a key $k = k_i$ may sound viable, **BUT** the search would
  1. need to inspect $\log_2(|T|)$ keys in the sequence set and access just as many pages 👎, and
  2. “jump around” the sequence set in an unpredictable fashion, thus leading to random I/O. 👎

B$^+$Trees exploit the sequence set ordering and erect an ***n*-ary search tree structure** (n large!) atop the leaf entries.

B⁺Trees: Inside an Inner Node



$o \overset{\text{def}}{=}$ B⁺Tree **order**

- The **separator** keys k_i are ordered: $k_i \leqslant k_{i+1}$.
- Page pointers p_j point to index (leaf or inner) nodes.

B⁺Trees: Notes on Inner Nodes

- Space in inner nodes is used economically: in a B⁺Tree of **order** o, any inner node—but the root node—is guaranteed to hold between o and $2 \times o$ ($\overset{\text{def}}{=}$ **fan-out** F) index entries.
- Given predicate $t.a = k$, perform **binary search inside node** to find B⁺Tree subtree with $k_i \leqslant k < k_{i+1}$.
- B⁺Tree is **balanced:** subtrees △ are of identical height.
- **Path length** s from B⁺Tree root to leaf node **predictable:**

$$|T| \times \underbrace{1/F \times \cdots \times 1/F}_{s \text{ times}} = 1 \Leftrightarrow s = \log_F(|T|)$$

Typical B^+Tree parameters (8 kB pages):

- Index entry size ≈ 16 bytes (key + page pointer)
- Typical fan-out $F \approx 500$ ($500 \times 16 + |\text{meta data}| = 8192$)
- $|T| = 10^6$, $F = 500$: $s = \log_{500}(10^6) \approx$ **2.2**
- $|T| = 10^9$, $F = 500$: $s = \log_{500}(10^9) \approx$ **3.3**

Explore root node and inner nodes of index indexed_a for table indexed. Uses

- bt_metap(‹index›)

to access B⁺Tree meta information (including page # of root node).

```
-- ❶ Access B⁺Tree meta information

db2=# SELECT root, level
      FROM bt_metap('indexed_a');
┌──────┬───────┐
│ root │ level │
├──────┼───────┤
│  412 │     2 │
└───▲──┴────▲──┘
 root page   root is at height 2 (leaves are at level 0, see field btpo above)


-- ❷ Access B⁺Tree root node

db2=# SELECT *
      FROM bt_page_stats('indexed_a', 412);
┌─[ RECORD 1 ]───┬──────┐
│ blkno          │ 412  │
│ type           │ r    ◀ this is the _r_oot node
│ live_items     │ 10   ◀ fan-out of 10 (10 subtrees of height 1)
│ dead_items     │ 0    │
│ avg_item_size  │ 15   │
│ page_size      │ 8192 │
│ free_size      │ 7956 │
│ btpo_prev      │ 0    ◀ ⎫  this is the root node, thus
│ btpo_next      │ 0    ◀ ⎭  no siblings on this tree level
│ btpo           │ 2    │
│ btpo_flags     │ 2    │
└────────────────┴──────┘


-- ❸ Access index entries in root node (the root is rightmost on its level 2 and
--   thus has no high key)

db2=# \x
db2=# SELECT itemoffset, itemlen, ctid, data
      FROM bt_page_items('indexed_a', 412)
      ORDER BY itemoffset;
┌────────────┬─────────┬──────────┬──────────────────────────┐
│ itemoffset │ itemlen │   ctid   │           data           │
├────────────┼─────────┼──────────┼──────────────────────────┤
```

```
│          1 │        8 │ (3,1)     │                          ◀ $p_0$ = 3 (there is no key $k_0$)
│          2 │       16 │ (411,1)   │ 77 97 01 00 00 00 00 00  ◀ index entry 1: ($k_1$ = 104311, $p_1$ = 411)
│          3 │       16 │ (698,1)   │ ed 2e 03 00 00 00 00 00  ◀ index entry 2: ($k_2$ = 208621, $p_2$ = 698)
│          4 │       16 │ (984,1)   │ 63 c6 04 00 00 00 00 00 │
│          5 │       16 │ (1270,1)  │ d9 5d 06 00 00 00 00 00 │  use  SELECT x'019777'::int  to convert hex in SQL
│          6 │       16 │ (1556,1)  │ 4f f5 07 00 00 00 00 00 │
│          7 │       16 │ (1842,1)  │ c5 8c 09 00 00 00 00 00 │
│          8 │       16 │ (2128,1)  │ 3b 24 0b 00 00 00 00 00 │   $k_i \leqslant k_{i+1}$
│          9 │       16 │ (2414,1)  │ b1 bb 0c 00 00 00 00 00 │
│         10 │       16 │ (2700,1)  │ 27 53 0e 00 00 00 00 00 │
└────────────┴────────▲─┴───────▲───┴─────────────────────────┘
         key (8 bytes) +    page pointers: ($p$,1)
page pointer (8 bytes)      points to page $p$ (ignore 1)
```

```
-- ❹ Explore B+Tree subtree with root page 411 (hosts index entries for values 104311 ⩽ $a$ < 208621)

db2=# SELECT *
      FROM bt_page_stats('indexed_a', 411);
┌─[ RECORD 1 ]───┬──────┐
│ blkno          │ 411  │
│ type           │ i    ◀ this is an _i_nner node
│ live_items     │ 286  ◀ fan-out (286 subtrees of height 0 ≡ index leaves)
│ dead_items     │ 0    │
│ avg_item_size  │ 15   │
│ page_size      │ 8192 │
│ free_size      │ 2436 │
│ btpo_prev      │ 3    ◀ previous node on level 1
│ btpo_next      │ 698  ◀ next node on level 1
│ btpo           │ 1    ◀ level 1
│ btpo_flags     │ 0    │
└────────────────┴──────┘
```

```
-- ❺ Explore index entries in B+Tree inner node on page 411 (hosts index entries for values 104311 ⩽ $a$ < 208621),
--   411 is non-rightmost on its level 1 and thus has a high key (≡ smallest key value on next subtree on level 1)

db2=# \x
db2=# SELECT itemoffset, itemlen, ctid, data
      FROM bt_page_items('indexed_a', 411)
      ORDER BY itemoffset
      LIMIT 10;
┌────────────┬─────────┬─────────┬─────────────────────────┐
│ itemoffset │ itemlen │  ctid   │          data           │
├────────────┼─────────┼─────────┼─────────────────────────┤
│          1 │      16 │ (574,1) │ ed 2e 03 00 00 00 00 00 ◀ high key = 208621 (smallest key in next subtree), ignore 574
│          2 │       8 │ (287,1) │                         ◀ $p_0$ = 287 (there is no key $k_0$)
│            │         │         │
│          3 │        16 │ (288,1) │ e5 98 01 00 00 00 00 00 ◀ index entry 1: ($k_1$ = 104677, $p_1$ = 288)
│          4 │        16 │ (289,1) │ 53 9a 01 00 00 00 00 00 ◀ index entry 2: ($k_1$ = 105043, $p_1$ = 289)
│          5 │        16 │ (290,1) │ c1 9b 01 00 00 00 00 00 │
│          6 │        16 │ (291,1) │ 2f 9d 01 00 00 00 00 00 │
│          7 │        16 │ (292,1) │ 9d 9e 01 00 00 00 00 00 │
│          8 │        16 │ (293,1) │ 0b a0 01 00 00 00 00 00 │
│          9 │        16 │ (294,1) │ 79 a1 01 00 00 00 00 00 │
│         10 │        16 │ (295,1) │ e7 a2 01 00 00 00 00 00 │
└────────────┴───────────┴─────────┴─────────────────────────┘


-- ❻ Explore leaf entries on B+Tree leaf node on page 288 (hosts index entries for values 104677 ⩽ $a$ < 105043)


db2=# SELECT itemoffset, itemlen, ctid, data
      FROM bt_page_items('indexed_a', 288)
      ORDER BY itemoffset;
┌────────────┬───────────┬──────────┬─────────────────────────┐
│ itemoffset │  itemlen  │   ctid   │          data           │
├────────────┼───────────┼──────────┼─────────────────────────┤
│          1 │        16 │ (981,76) │ 53 9a 01 00 00 00 00 00 ◀ high key = 105043
│          2 │        16 │ (978,31) │ e5 98 01 00 00 00 00 00 ◀ lowest key $k_1$ = 104677
│          3 │        16 │ (978,32) │ e6 98 01 00 00 00 00 00 │
│          4 │        16 │ (978,33) │ e7 98 01 00 00 00 00 00 │
│          5 │        16 │ (978,34) │ e8 98 01 00 00 00 00 00 │
[...]
│        365 │        16 │ (981,73) │ 50 9a 01 00 00 00 00 00 │
│        366 │        16 │ (981,74) │ 51 9a 01 00 00 00 00 00 │
│        367 │        16 │ (981,75) │ 52 9a 01 00 00 00 00 00 ◀ highest key $k_{366}$ = 105042
└────────────┴───────────┴──────────┴─────────────────────────┘
```

3 ¦ Index Scan

A B^+Tree is *the* index structure to support the evaluation of these kinds of conditions:[3]

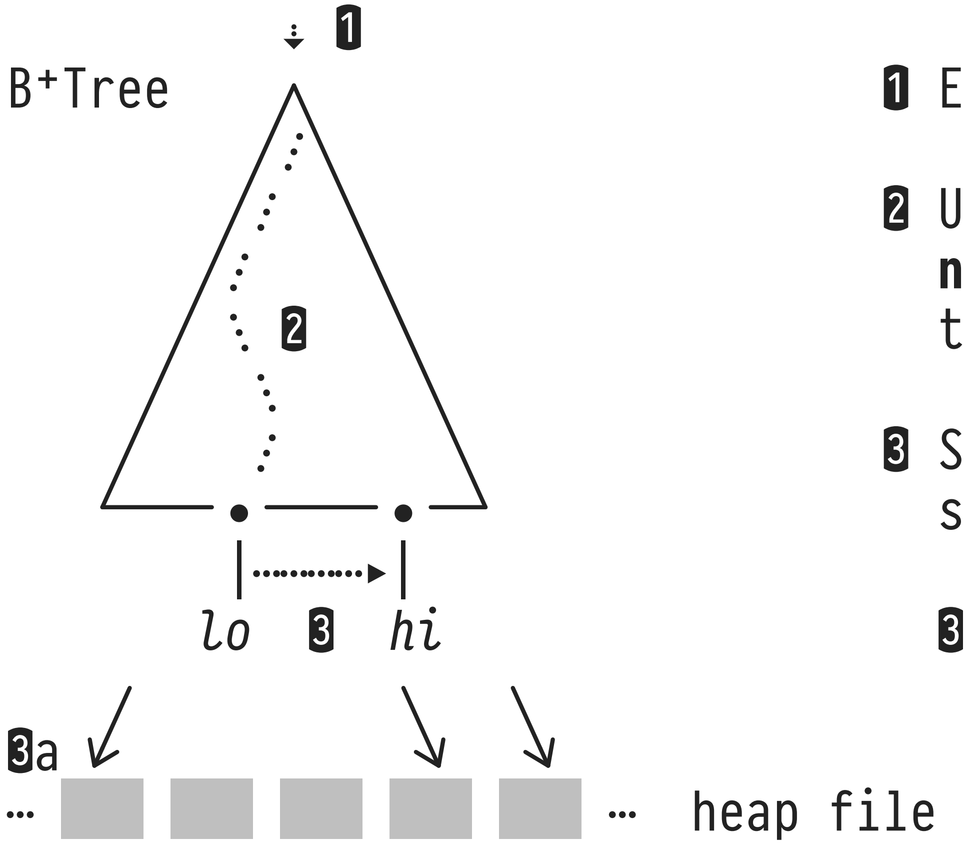
1. **Range predicates:** $lo \leqslant a \leqslant hi$
2. **Half-open ranges:** $lo \leqslant a$ or $a \leqslant hi$
3. **Equality predicates:** $a = hi$

- An **Index Scan** on index I for column $T(a)$ is parameterized by such a condition (PostgreSQL EXPLAIN: Index Cond).
- **Index Scan** uses lo to navigate the search tree structure and locate the start of relevant sequence set section.

[3] Half-open ranges are special range predicates where $hi = \infty$ ($lo = \infty$). Equality predicates are special range predicates where $lo = hi$.

Index Scan for Condition $lo \leqslant a \leqslant hi$

An **index scan** accesses the B⁺Tree index *and* the heap file:



1 Enter at B⁺Tree root page

2 Use key *lo* to **navigate the inner nodes** (search tree) until we reach the leaf level

3 Scan leaf entries in the sequence set section $lo \leqslant a \leqslant hi$, **extract RIDs**

3a For each RID, **access heap file for table** *T* and return matching row

Navigating the Inner Nodes

Phase ❷ runs a vanilla traversal of a 2×o-way search tree:

```
Search(lo):
  return TreeSearch(lo, root);
```

returns entry point for scan of sequence set

```
TreeSearch(lo, node):
  if (node is a leaf)
    return node;
  switch lo
    case lo < $k_1$
      return TreeSearch(lo, $p_0$);
    case $k_i \leqslant$ lo < $k_{i+1}$
      return TreeSearch(lo, $p_i$);
    case $k_{2o} \leqslant$ lo
      return TreeSearch(lo, $p_{2o}$);
```

use binary search to implement subtree choice

Demonstrate: PostgreSQL prefers **index scan** when the index condition is selective and the cost of accessing the index AND the heap file appears sufficiently low:

```
db2=# EXPLAIN ANALYZE
        SELECT i.a, i.b
        FROM   indexed AS i
        WHERE  i.a < 1000;
```

```
                                            QUERY PLAN
Index Scan using indexed_a on indexed i  (cost=0.42..45.36 rows=1082 width=37) (actual time=0.047..3.139 rows=999 loops=1)
  Index Cond: (a < 1000)
Planning time: 0.268 ms                                         quite good estimate
Execution time: 3.317 ms
```

```
db2=# EXPLAIN ANALYZE
        SELECT i.a, i.b
        FROM   indexed AS i
        WHERE  i.a < 500000;
```

```
                                            QUERY PLAN
Index Scan using indexed_a on indexed i  (cost=0.42..19068.13 rows=503926 width=37) (actual time=0.058..257.234 rows=499999 loops=1)
  Index Cond: (a < 500000)
Planning time: 0.128 ms
Execution time: 292.927 ms
```

```
db2=# EXPLAIN ANALYZE
        SELECT i.a, i.b
        FROM   indexed AS i
        WHERE  i.a < 570000;
```

```
                                            QUERY PLAN
Seq Scan on indexed i  (cost=0.00..21846.00 rows=573505 width=37) (actual time=0.025..293.761 rows=569999 loops=1)
  Filter: (a < 570000)
  Rows Removed by Filter: 430001            collect RID set will be large, need to access
Planning time: 0.122 ms                     large number of heap file pages anyway
Execution time: 337.295 ms
```

```
-- Forcing PostgreSQL to use an index scan: there is indeed no benefit:
```

```
db2=# set enable_seqscan = off;

db2=# EXPLAIN ANALYZE
        SELECT i.a, i.b
        FROM   indexed AS i
        WHERE  i.a < 570000;
```

```
                                                          QUERY PLAN
Index Scan using indexed_a on indexed i  (cost=0.42..21699.76 rows=573505 width=37) (actual time=0.111..287.229 rows=569999 loops=1)
  Index Cond: (a < 570000)
Planning time: 0.126 ms
Execution time: 332.915 ms ⬅ no advantage over Seq Scan (see above)
```

```
db2=# set enable_seqscan = on;
```

- ⇒ PostgreSQL considers *expected cost* when choosing a plan: cost-based query optimization (→ later)

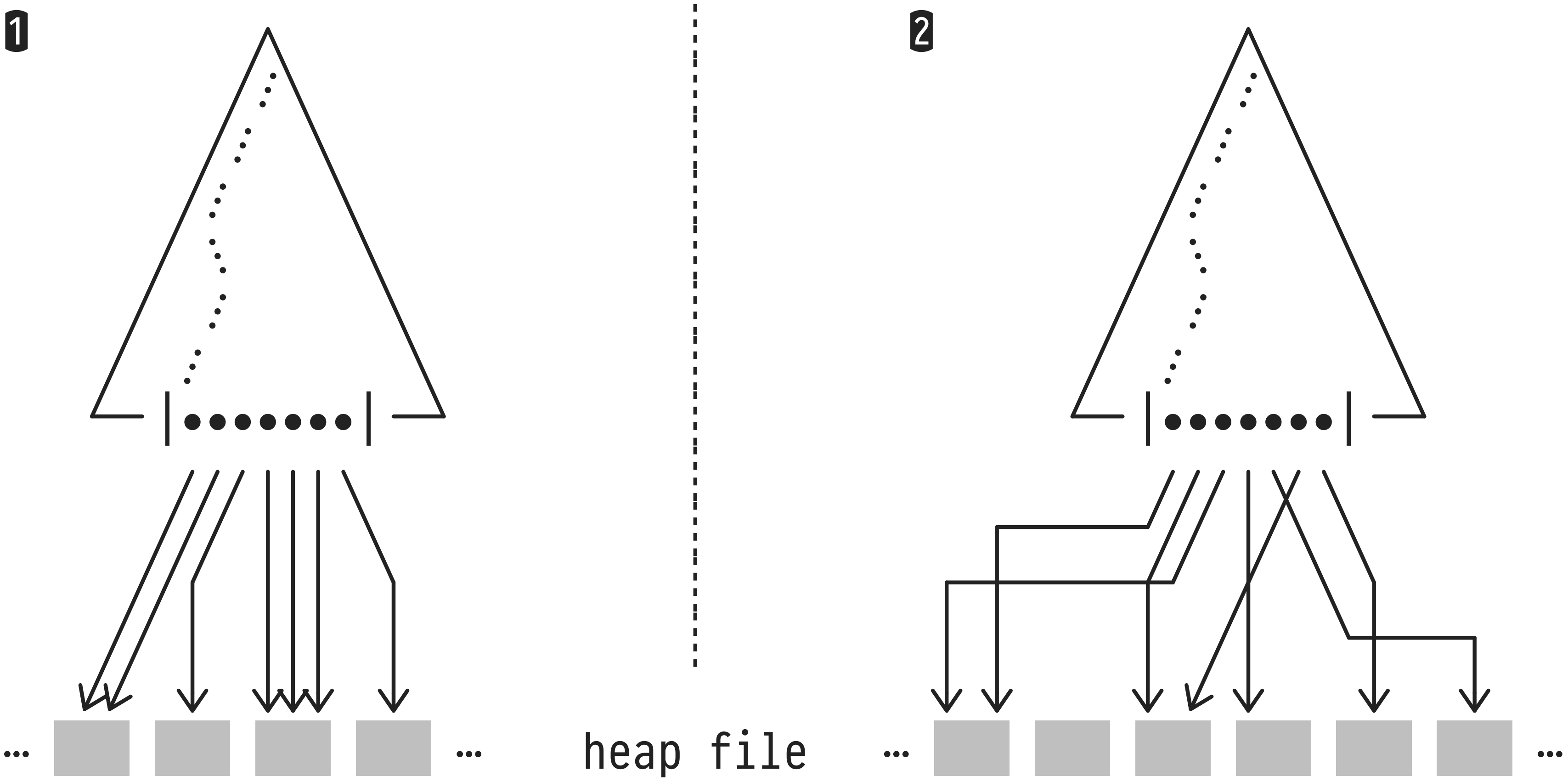
Index scans can **really pay off**! Want to establish B⁺Tree for the other columns of table indexed, too:

```
-- ❶ Create additional indexes on columns b and c:

  CREATE INDEX indexed_b ON indexed USING btree (b text_pattern_ops);
  CREATE INDEX indexed_c ON indexed USING btree (c);
  \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_b" btree (b varchar_pattern_ops)
    "indexed_c" btree (c)

  CLUSTER indexed USING indexed_a;
  ANALYZE indexed;    -- Hey PostgreSQL, reconsider indexes and statistics for table indexed!


-- ❷ Perform a selection column c (we expect an Index Scan on index indexed_c):

  EXPLAIN ANALYZE
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE  i.c = 0.42;
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                                          QUERY PLAN                                                          │
├──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┤
│➡Bitmap Heap Scan on indexed i  (cost=73.88..7247.63 rows=3801 width=37) (actual time=3.769..9.056 rows=3531 loops=1)         │
│   Recheck Cond: (c = 0.42)                                                                                                   │
│   Heap Blocks: exact=2964 ⬅                                                                                                  │
│   -> ➡Bitmap Index Scan on indexed_c  (cost=0.00..72.93 rows=3801 width=0) (actual time=2.944..2.944 rows=3531 loops=1)     │
│         Index Cond: (c = 0.42)                                                                               ⬆               │
│ Planning time: 0.535 ms                                                                   i.c = 0.42 is quite selective,     │
│ Execution time: 9.397 ms                                                                  we thus expected an Index Scan     │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

1. PostgreSQL uses the pair **Bitmap Index Scan** & **Bitmap Heap Scan**
2. The result has 3531 rows. **Bitmap Heap Scan** had to read 2964 heap file blocks to create this result (⇒ on average: 1.19 matching rows per block :-/)

What is going on here?

4 ┆ Order of Leaf Entries vs. Order of Table Rows



❶ Order of leaf entry keys k_i ≡ row order in heap file. 👍

❷ Order of k_i in sequence set and row order do *not* match.

Demonstrate the effect of clustering for the two indexes indexed_a and indexed_c for two conditions of identical selectivity:

```
-- ❶ Demonstrate two queries of identical complexity

  EXPLAIN ANALYZE
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE  i.a < 3532;
```

```
                                                                QUERY PLAN

 Index Scan using indexed_a on indexed i  (cost=0.42..134.14 rows=3355 width=37) (actual time=0.047..2.927 rows=3531 loops=1)
   Index Cond: (a < 3532)
 Planning time: 0.208 ms
 Execution time: 3.395 ms
```

```
  EXPLAIN ANALYZE
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE  i.c = 0.42;
```

```
                                                                QUERY PLAN

➡Bitmap Heap Scan on indexed i  (cost=73.88..7247.63 rows=3801 width=37) (actual time=4.183..8.776 rows=3531 loops=1)
   Recheck Cond: (c = 0.42)
   Heap Blocks: exact=2964
   -> ➡Bitmap Index Scan on indexed_c  (cost=0.00..72.93 rows=3801 width=0) (actual time=2.999..2.999 rows=3531 loops=1)
         Index Cond: (c = 0.42)
 Planning time: 0.164 ms
 Execution time: 9.081 ms
```

```
-- ❷ Show that 3531 matchings rows...
     - indexed_a: ... cluster on fewer and closer heap file pages,
     - indexed_c: ... are found on many pages spread all over the heap file.

-- Extract page p from RID (p,_)
  DROP FUNCTION IF EXISTS page_of(tid);
  CREATE FUNCTION page_of(rid tid) RETURNS bigint AS
  $$
    SELECT (rid::text::point)[0]::bigint;
  $$
  LANGUAGE SQL;
```

```
SELECT COUNT (DISTINCT page_of(i.ctid)) AS pages,
       MAX(page_of(i.ctid)) - MIN(page_of(i.ctid)) + 1 AS span
FROM   indexed AS i
WHERE  i.a < 3532;
```

pages	span
33	33

⬅ all rows found on 33 adjacent pages ⇒ indexed is clustered w.r.t. indexed_a

```
SELECT COUNT (DISTINCT page_of(i.ctid)) AS pages,
MAX(page_of(i.ctid)) - MIN(page_of(i.ctid)) + 1 AS span
FROM   indexed AS i
WHERE  i.c = 0.42;
```

pages	span
2964	9345

⬅ rows found on many distinct pages all over the place (heap file has 9346 pages in total)

Clustered Indexes

An index I for column $T(a)$ is **clustered** if the order of leaf entries coincides with T's row order (i.e., both I's sequence set and T's heap file are ordered by a):

Given entries $\langle k_i, p_i \rangle$ and $\langle k_j, p_j \rangle$, $k_i \leqslant k_j \Rightarrow p_i \leqslant p_j$.

- An **Index Scan** over a *clustered* index
  1. collects matching rows from adjacent heap file pages (⇒ sequential I/O 👍),
  2. will find many matching rows on each loaded heap file page (⇒ less page I/O 👍).

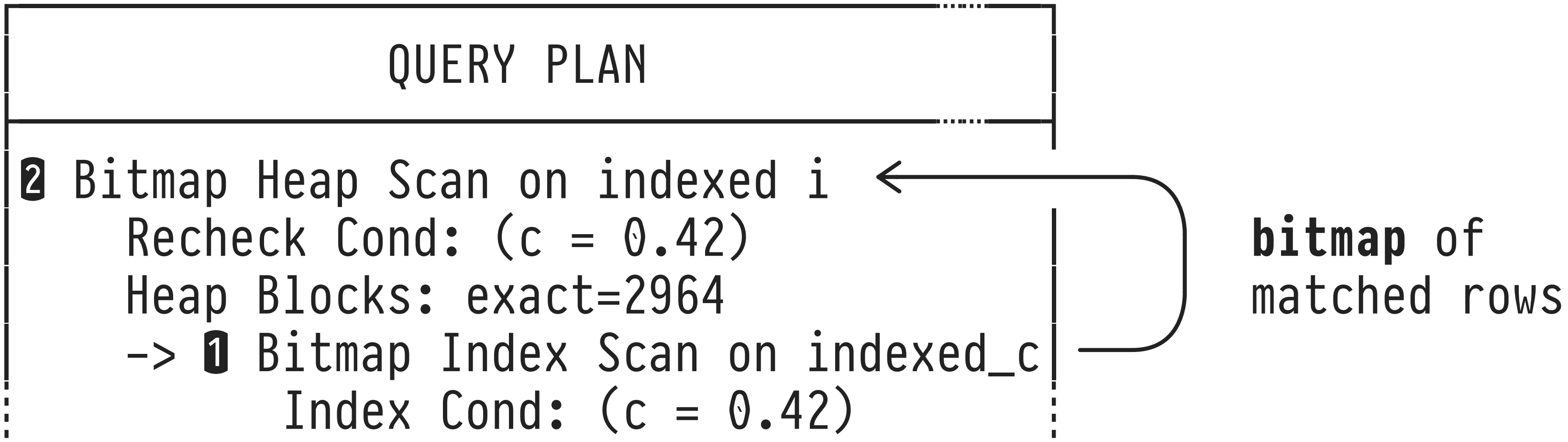
Non-Clustered Indexes

Sad fact: only *one*—among the many possible—indexes for a table may be clustered. Most indexes are non-clustered.

- An **Index Scan** over a *non-clustered* index
  1. will find matching rows potentially scattered across all heap file pages (⇒ random I/O 👎),
  2. will find few matching rows on each loaded heap file page and may access the same page more than once (⇒ as many page I/Os as matching rows 👎).

PostgreSQL addresses this challenge through **RID sorting**, implemented via **Bitmap Index Scan** & **Bitmap Heap Scan**.

Bitmap Index Scan & Bitmap Heap Scan



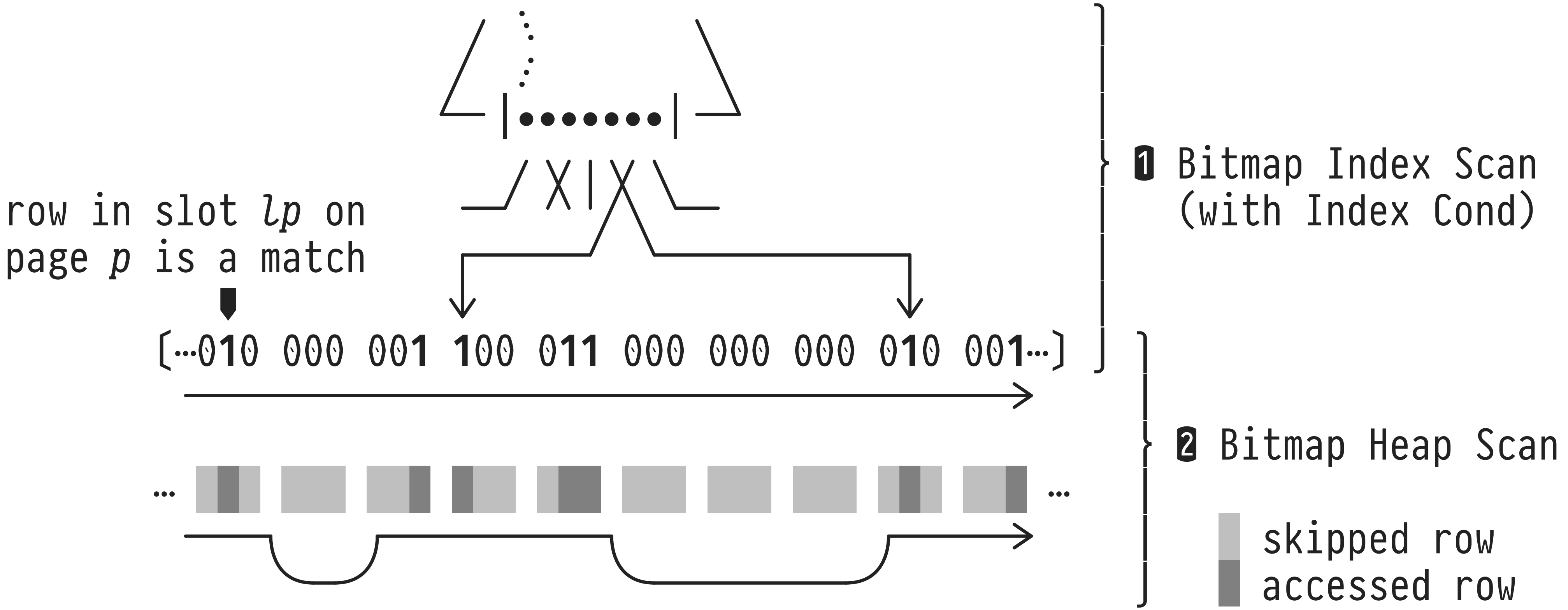
```
                 QUERY PLAN
❷ Bitmap Heap Scan on indexed i
    Recheck Cond: (c = 0.42)
    Heap Blocks: exact=2964
    -> ❶ Bitmap Index Scan on indexed_c
          Index Cond: (c = 0.42)
```

bitmap of
matched rows

❶ **Bitmap Index Scan:** perform Index Scan and create **bitmap** that encodes *heap file locations* of rows matching the Index Cond. Do *not* access rows in heap file yet.

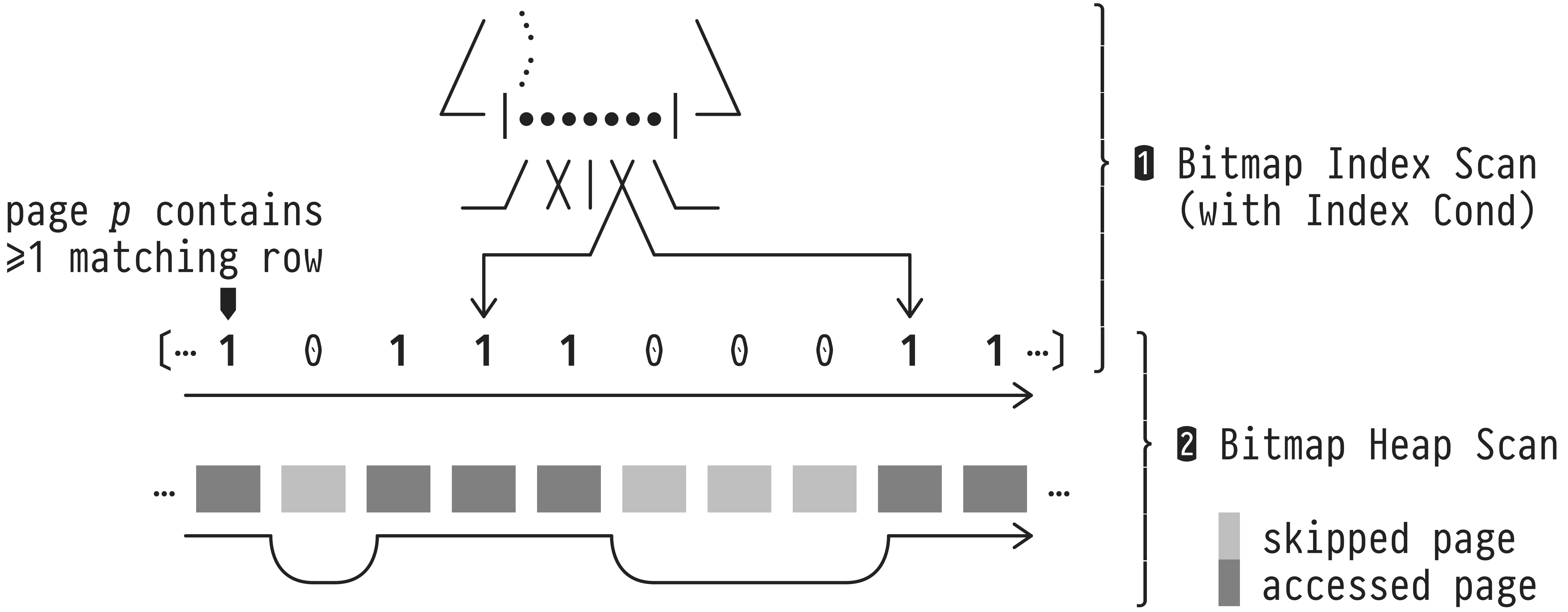
❷ **Bitmap Heap Scan:** scan heap file once, only access those rows (pages) that have been marked **1** in the bitmap.

Bitmap Index Scan & Bitmap Heap Scan: Row-Level Bitmap



Bitmap Heap Scan performs one sequential scan (with skips) of the heap file, regardless of RID order in sequence set.

Bitmap Index Scan & Bitmap Heap Scan: Page-Level Bitmap



Working memory tight ⇒ build **page-level** bitmap. ⚠️ In ❷, need to **recheck condition** for all rows on accessed pages.

Demonstrate the effect of tight working memory on Bitmap Index/Heap Scan, switches from exact (row-level) to lossy (page-level) bitmap encoding of matches.

```
-- ❶ Check default working memory and perform bitmap index scan-based query

  show work_mem;
```

work_mem
4MB

```
  EXPLAIN (VERBOSE, ANALYZE)
   SELECT i.a, i.b
   FROM   indexed AS i
   WHERE i.c = 0.42;
```

```
                                         QUERY PLAN
Bitmap Heap Scan on public.indexed i  (cost=73.88..7247.63 rows=3801 width=37) (actual time=5.754..13.620 rows=3531 loops=1)
  Output: a, b
  Recheck Cond: (i.c = 0.42)  ⬅ ⚠ shown here but NOT actually performed by PostgreSQL (EXPLAIN artefact)
  Heap Blocks: exact=2964     ⬅ exact (= row-level) bitmap
  ->  Bitmap Index Scan on indexed_c  (cost=0.00..72.93 rows=3801 width=0) (actual time=4.621..4.621 rows=3531 loops=1)
        Index Cond: (i.c = 0.42)
Planning time: 0.253 ms
Execution time: 14.024 ms ⬅ fast!
```

```
-- ❷ Repeat query with severely restriced working memory

  set work_mem = '64kB';

  EXPLAIN (VERBOSE, ANALYZE)
   SELECT i.a, i.b
   FROM   indexed AS i
   WHERE i.c = 0.42;
```

```
                                         QUERY PLAN
Bitmap Heap Scan on public.indexed i  (cost=73.88..7247.63 rows=3801 width=37) (actual time=3.504..78.213 rows=3531 loops=1)
  Output: a, b
  Recheck Cond: (i.c = 0.42)              ⬅ indeed performed now
  Rows Removed by Index Recheck: 220513  ⬅ recheck of the condition discarded this many rows from the 2084 lossy pages
  Heap Blocks: exact=880 lossy=2084      ⬅ parts of the bitmap representd lossy (= pagel-level), need to recheck condition in these 2084 pages
  ->  Bitmap Index Scan on indexed_c  (cost=0.00..72.93 rows=3801 width=0) (actual time=3.080..3.080 rows=3531 loops=1)
```

```
        Index Cond: (i.c = 0.42)
Planning time: 0.161 ms
Execution time: 78.505 ms ⬅ slower...
```

```
-- ❸ Reset working memory size

  set work_mem = '4MB';   -- also: set work_mem = default;
```

5 ¦ CLUSTERing Based on an Index

If the workload depends on top performance of particular predicates supported by *non-clustered* index *I*, we may

> **physically reorder the rows of underlying table's** *T* **heap file** to coincide with the key order in *I*'s sequence set (i.e., *I* will become a *clustered* index[4]):

```
CLUSTER [VERBOSE] ‹T› USING ‹I›;
CLUSTER ‹T›;    -- re-cluster once T's rows get out of order
```

- ⚠️ Subsequent updates on *T* can destroy the perfect clustering. (May need to re-cluster *T* in intervals.)

[4] At a price, of course: formerly *clustered* indexes on *T* will turn into *non-clustered* indexes.

Demonstrate the effect of clustering on Bitmap Index Scan (find all matches on significantly fewer pages):

```
-- ❶ Perform Bitmap Index Scan on non-clustered index

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE i.c = 0.42;
```

```
                                         QUERY PLAN
Bitmap Heap Scan on public.indexed i  (cost=73.88..7247.63 rows=3801 width=37) (actual time=4.729..10.112 rows=3531 loops=1)
  Output: a, b
  Recheck Cond: (i.c = 0.42)
  Heap Blocks: exact=2964 ⬅ 3531 matches found on 2964 pages (1.19 matched rows/page :-/)
  ->  Bitmap Index Scan on indexed_c  (cost=0.00..72.93 rows=3801 width=0) (actual time=3.559..3.559 rows=3531 loops=1)
        Index Cond: (i.c = 0.42)
Planning time: 0.734 ms
Execution time: 10.434 ms ⬅ slow
```

```
-- ❷ Recluster table 'indexed' based on index 'indexed_c', writes a new heap file

  SELECT relfilenode
  FROM   pg_class
  WHERE  relname = 'indexed';
```

```
relfilenode
     113411 ⬅ old heap file
```

```
  CLUSTER VERBOSE indexed USING indexed_c;

  \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_b" btree (b varchar_pattern_ops)
    "indexed_c" btree (c) CLUSTER ◀

  SELECT relfilenode
  FROM   pg_class
  WHERE  relname = 'indexed';

┌─────────────┐
│ relfilenode │
├─────────────┤
│      129657 │◀ new heap file
└─────────────┘

-- Physical order of rows in heap file now coincides with order in column 'c'
  SELECT i.ctid, i.*
  FROM   indexed AS i
  ORDER BY i.c;

┌────────────┬─────────┬──────────────────────────────────┬───────┐
│    ctid    │    a    │                b                 │   c   │
├────────────┼─────────┼──────────────────────────────────┼───────┤
│ (0,1)      │      11 │ 6512bd43d9caa6e02c990b0a82652dca │ -1.00 │
│ (0,2)      │      55 │ b53b3a3d6ab90ce0268229151c9bde11 │ -1.00 │
│ (0,3)      │      99 │ ac627ab1ccbdb62ec96e702f07f6425b │ -1.00 │
[...]
│ (9342,58)  │  999958 │ 3a5338afb1f571d903d85b495aec7363 │  1.00 │
│ (9342,59)  │  999983 │ 4034d2f137199ef04b7544d2d333ed67 │  1.00 │
└────▲───────┴─────────┴──────────────────────────────────┴───▲───┘


-- ⚠ DO YOU WANT TO SHOW THIS?  Could be a good homework assignment:
-- Clustering factor of table 'indexed' w.r.t. columns 'c' and 'a':

  [redacted]

┌──────────────────────┐
│ clustering_factor    │
├──────────────────────┤
│ 100.0000000000000000 │
└──────────────────────┘

  [redacted]

┌─────────────────────┐
│ clustering_factor   │
├─────────────────────┤
│ 50.1717000000000000 │
└─────────────────────┘
```

```
-- ❸ Repeat query (Bitmap Index Scan will now touch less block)
--   ⚠️ DO NOT perform 'ANALYZE indexed' yet!

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.a, i.b
    FROM   indexed AS i
    WHERE  i.c = 0.42;
```

```
                                                                QUERY PLAN
 ─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
 Bitmap Heap Scan on public.indexed i  (cost=73.88..7246.75 rows=3801 width=37) (actual time=1.484..3.466 rows=3531 loops=1)
   Output: a, b
   Recheck Cond: (i.c = 0.42)
   Heap Blocks: exact=34  ⬅ matches found significantly less pages (34 << 2964), now 103.85 matched rows/page :-)
   ->  Bitmap Index Scan on indexed_c  (cost=0.00..72.93 rows=3801 width=0) (actual time=1.452..1.452 rows=3531 loops=1)
         Index Cond: (i.c = 0.42)
 Planning time: 0.197 ms
 Execution time: 4.059 ms ⬅ faster!
```

```
-- ❹ Run ANALYZE on table 'indexex', DBMS updates statistics on
--   row order, now chooses Index Scan over Bitmap Index Scan

  ANALYZE indexed;

  EXPLAIN (VERBOSE, ANALYZE)
   SELECT i.a, i.b
   FROM   indexed AS i
   WHERE  i.c = 0.42;
```

```
                                                                    QUERY PLAN
 ─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
➡Index Scan using indexed_c on public.indexed i  (cost=0.42..149.50 rows=3776 width=37) (actual time=0.097..3.963 rows=3531 loops=1)
   Output: a, b
   Index Cond: (i.c = 0.42)
 Planning time: 0.434 ms
 Execution time: 4.494 ms
```

B$^+$Trees…

1. **economically utilize space** in inner/leaf nodes (minimum node occupancy 50%, typical fill factor 67%),
2. are **balanced** trees and thus require a **predictable number of page I/Os** to traverse from root to sequence set—enables query optimizer to forecast B$^+$Tree access cost.

DBMSs maintain properties 1. and 2. when rows are **inserted** into/**deleted** from an B$^+$Tree-indexed table.[5]

[5] Some real B$^+$Tree implementations of row deletion deviate from the textbook to keep things simpler.

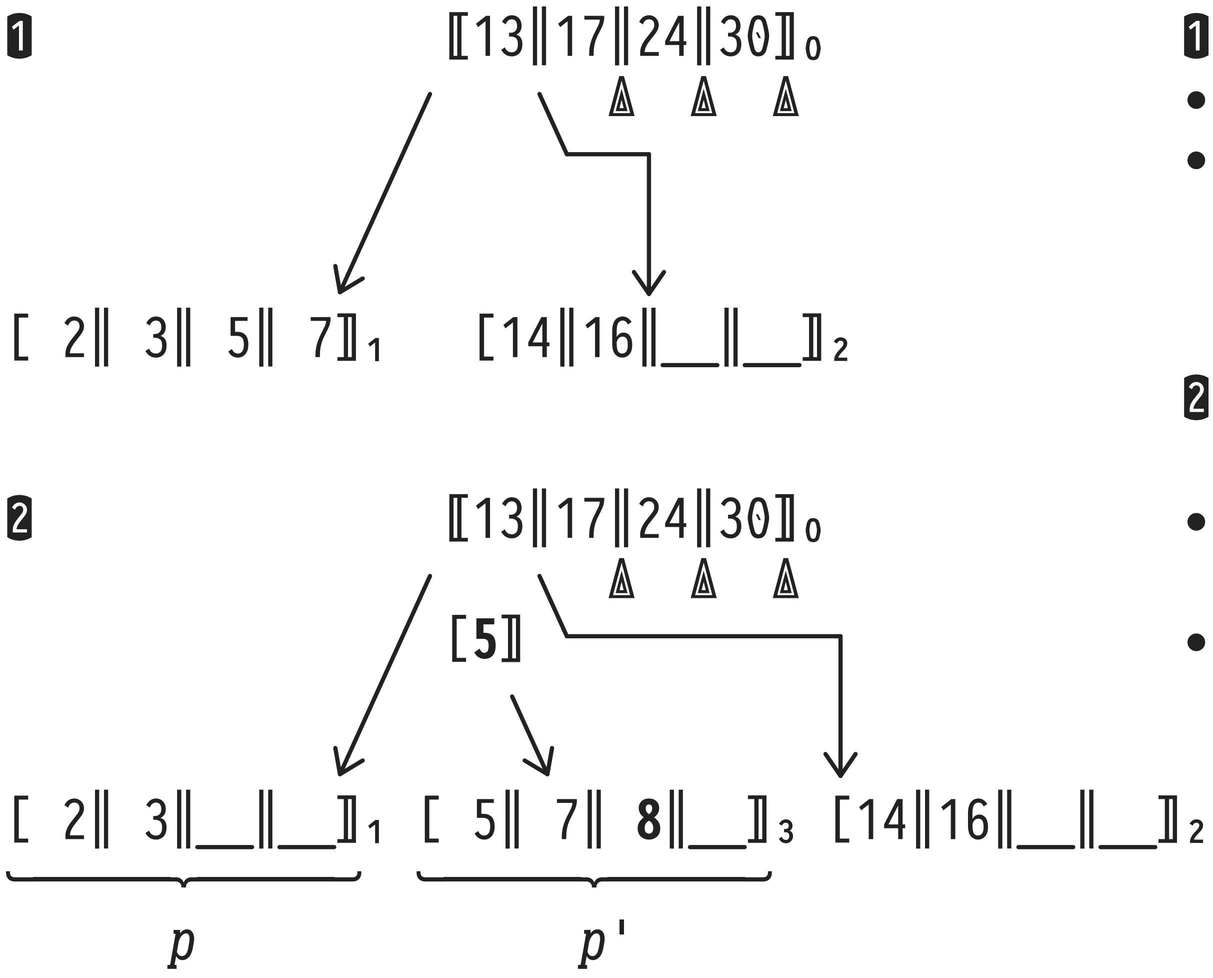
B$^+$Tree Insertion for New Entry ‹*k*,*rid*› (Sketch)

1. Use Search(*k*) to **find leaf page** *p* which should hold the entry for *k*.
2. If *p* has **enough space** to hold new entry (i.e., at most 2×o-1 entries in *p*), **simply insert** ‹*k*,*rid*› into *p*.
3. Otherwise, node *p* must be **split** into *p* and *p*' and a new **separator** has to be inserted↺ into the parent of *p*.

   Splitting happens recursively↺ and may eventually lead to a split of the root node (increasing B$^+$Tree height).

4. **Distribute** the entries of *p* and new entry ‹*k*,*rid*› onto pages *p* and *p*'.

B$^+$Tree Insertion and Leaf Node Split



❶ ⟦13‖17‖24‖30⟧$_0$

⍋ ⍋ ⍋

[2‖ 3‖ 5‖ 7⟧$_1$ [14‖16‖__‖__⟧$_2$

❷ ⟦13‖17‖24‖30⟧$_0$

⍋ ⍋ ⍋

[**5**⟧

[2‖ 3‖__‖__⟧$_1$ [5‖ 7‖ **8**‖__⟧$_3$ [14‖16‖__‖__⟧$_2$

p *p*'

❶ Insert new entry ‹**8**,*rid*›

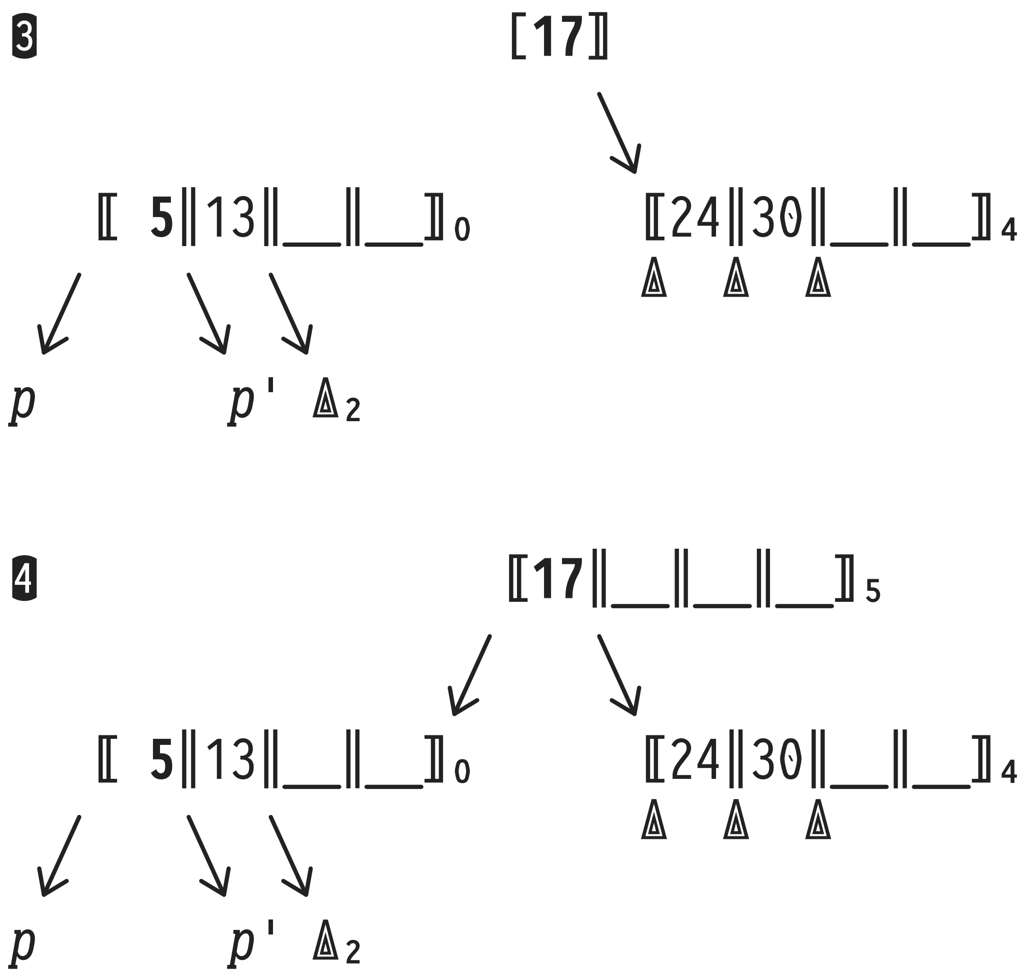
- Search(**8**) returns leaf *p* = 1
- Leaf 1 is full ⇛ split

❷ Leaf 1 split into leaves *p* = 1 and *p*' = 3

- Distribute {2,3,5,7,**8**} between leaves 1 and 3
- **Copy**↺ new separator [**5**⟧ into parent node 0

p'

B⁺Tree Insertion and Inner Node Split



❸ Inner node 0 (here: root) is full ⇒ split

- Inner node 0 splits into old node 0 and new *p*'' = 4
- Distribute {**5**,13,24,30} **𝔸** between nodes 0 and 4
- **Move**↰ new separator [**17**⟧ into parent of node 0

↘ *p*''

❹ Split node 0 has been the old root

- Create new root node 5, has ⟦**17**‖ as only entry
- B⁺Tree height has increased

B⁺Tree Insertion Notes

- Splitting starts at the leaf level and continues upward as long as inner index nodes are fully occupied (holding 2×o entries).

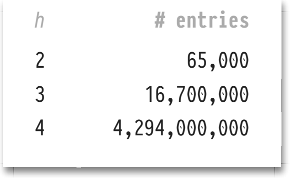
- ⚠ Unlike during a *leaf* split, an *inner* node split **moves**[6] the new separator [***sep***⟧ discriminating between p and p' upwards and recursively inserts it into the parent. **Q:** Why?

- **Q:** How often do you expect a root node split to happen?

[6] A leaf node split **copies** the new separator upwards, i.e., the entry [***sep***⟧ also remains at the leaf level.

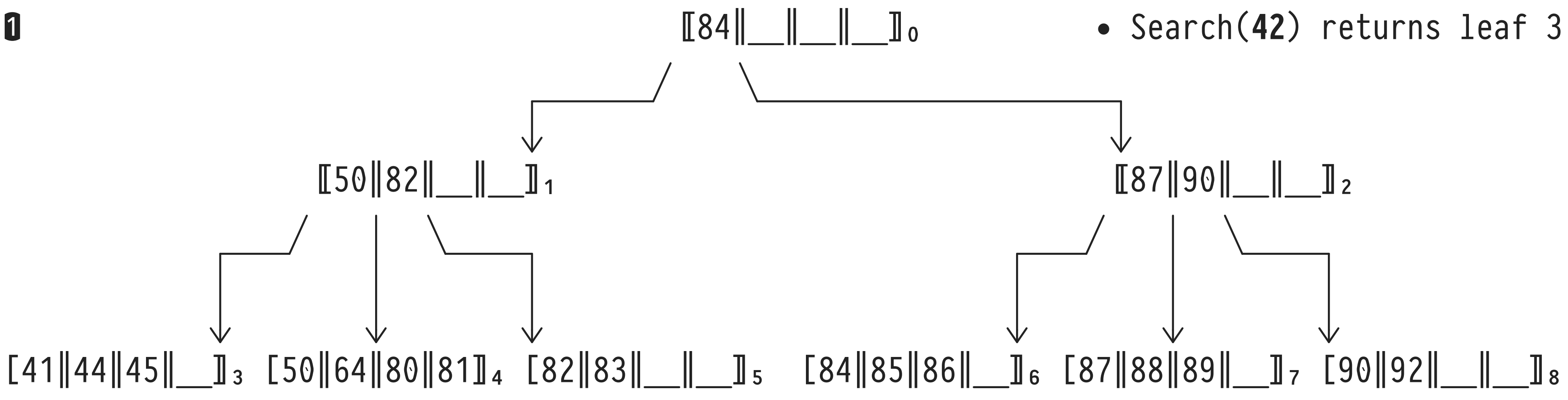
Frequency of root node splits:

- B^+Tree over 8 byte integers, pointers also occupy 8 bytes, assume 8 kB page size.
- ⇒ A node holds between 256 and 512 index entries (i.e., fan-out $F = 512$)
- A B^+Tree of height h holds *at least* 256^h entries, typically more.

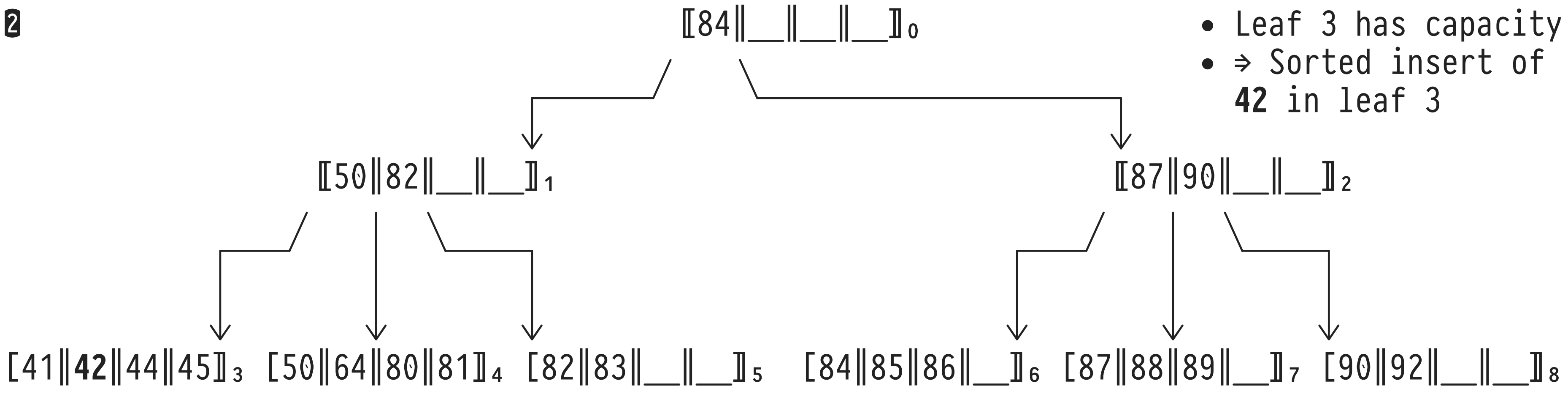


h	# entries
2	65,000
3	16,700,000
4	4,294,000,000

B^{+}Tree Insertion Example: Insert ‹42,*rid*›



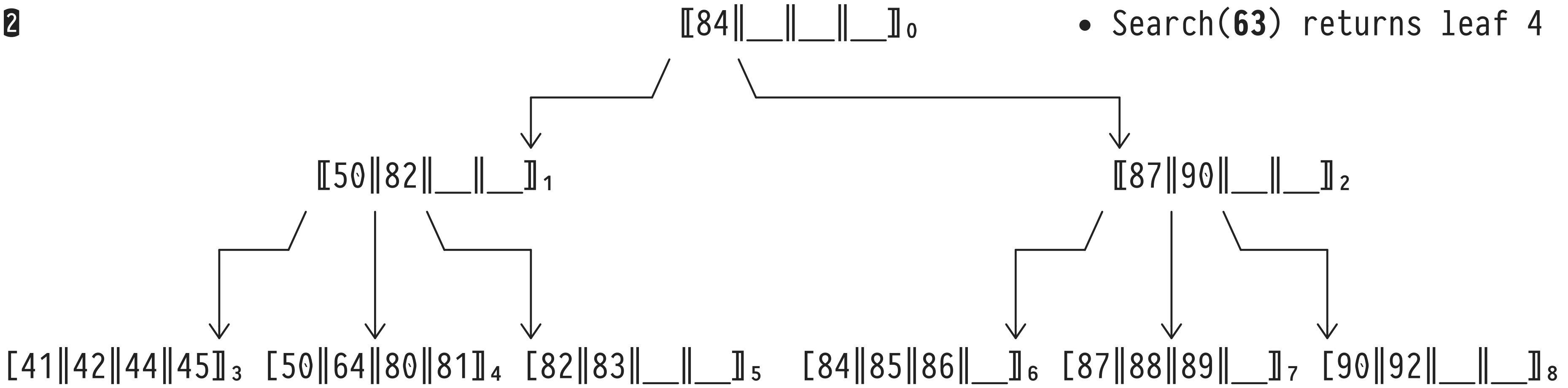
- Search(**42**) returns leaf 3



- Leaf 3 has capacity
- ⇒ Sorted insert of **42** in leaf 3

B⁺Tree Insertion Example: Insert ‹63,*rid*›

❷

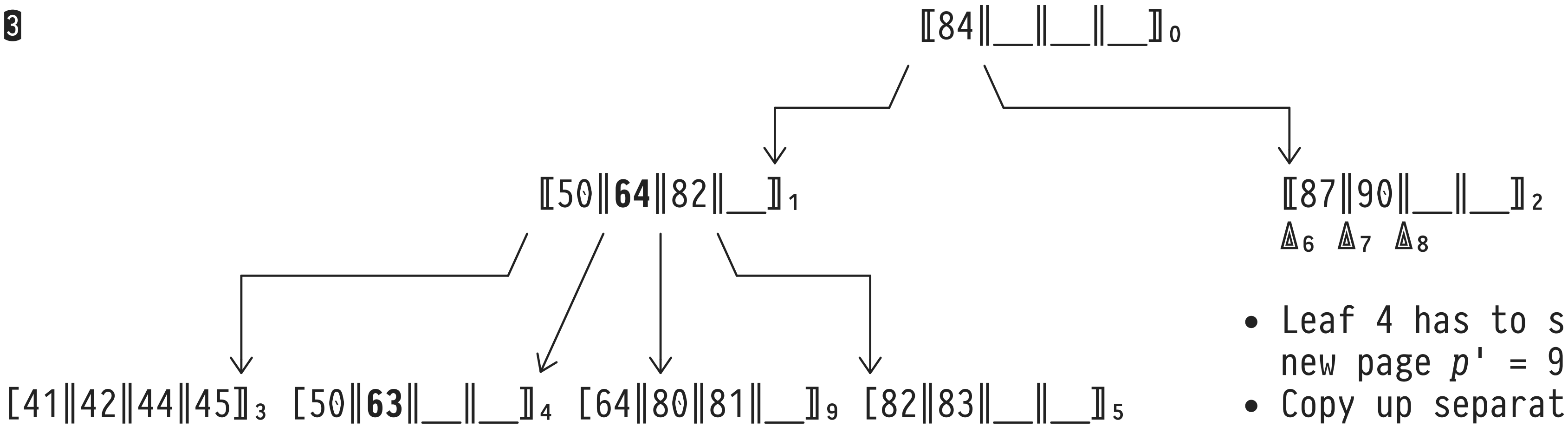

⟦84‖__‖__‖__⟧$_0$

⟦50‖82‖__‖__⟧$_1$ ⟦87‖90‖__‖__⟧$_2$

[41‖42‖44‖45⟧$_3$ [50‖64‖80‖81⟧$_4$ [82‖83‖__‖__⟧$_5$ [84‖85‖86‖__⟧$_6$ [87‖88‖89‖__⟧$_7$ [90‖92‖__‖__⟧$_8$

- Search(**63**) returns leaf 4

❸


⟦84‖__‖__‖__⟧$_0$

⟦50‖**64**‖82‖__⟧$_1$ ⟦87‖90‖__‖__⟧$_2$

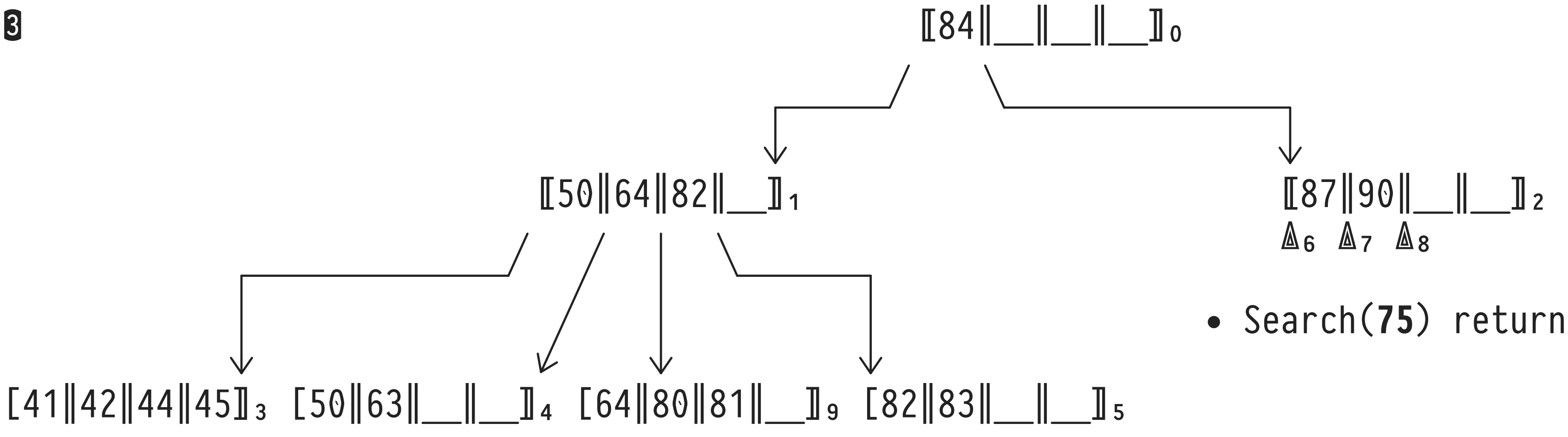
△$_6$ △$_7$ △$_8$

[41‖42‖44‖45⟧$_3$ [50‖**63**‖__‖__⟧$_4$ [64‖80‖81‖__⟧$_9$ [82‖83‖__‖__⟧$_5$

- Leaf 4 has to split, new page *p*' = 9
- Copy up separator [**64**⟧

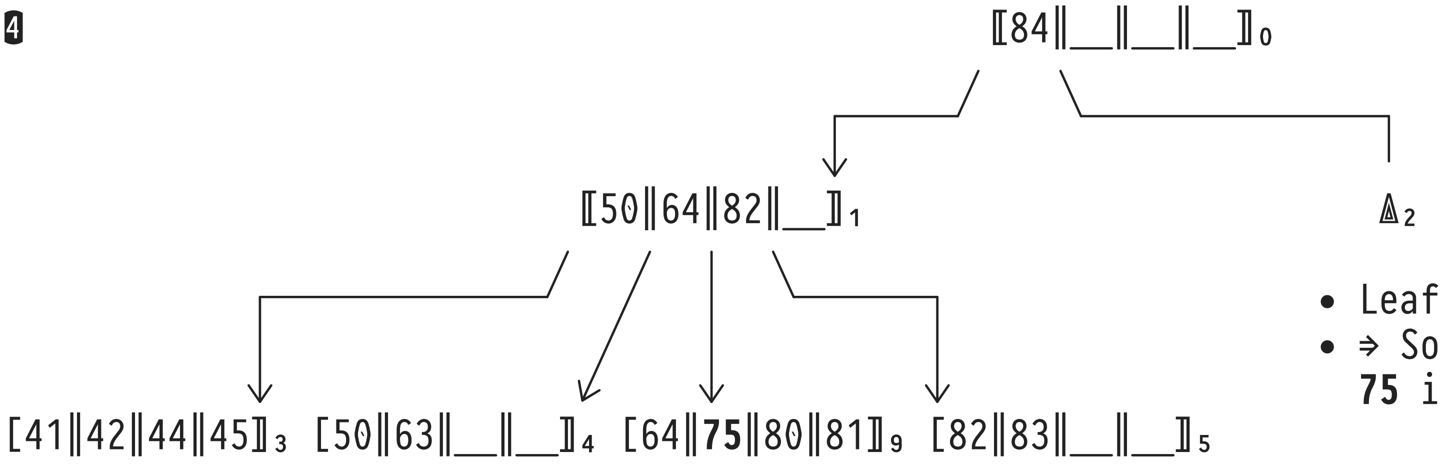
B^+Tree Insertion Example: Insert ‹75,*rid*›

❸



- Search(**75**) returns leaf 9

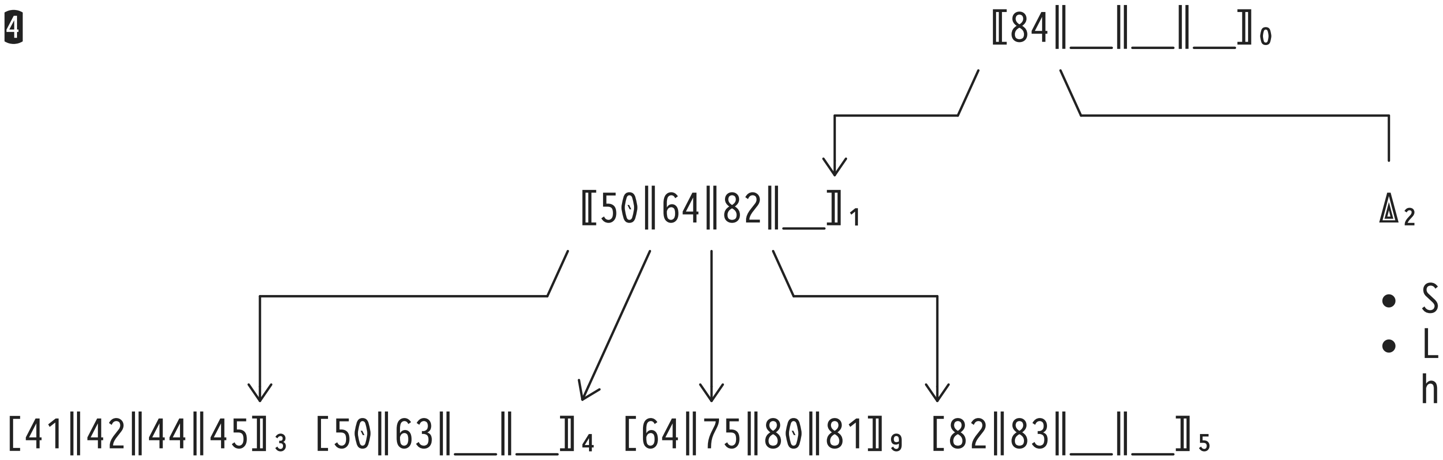
❹



- Leaf 9 has capacity
- ⇛ Sorted insert of **75** in leaf 9

B⁺Tree Insertion Example: Insert ‹77,*rid*›

4



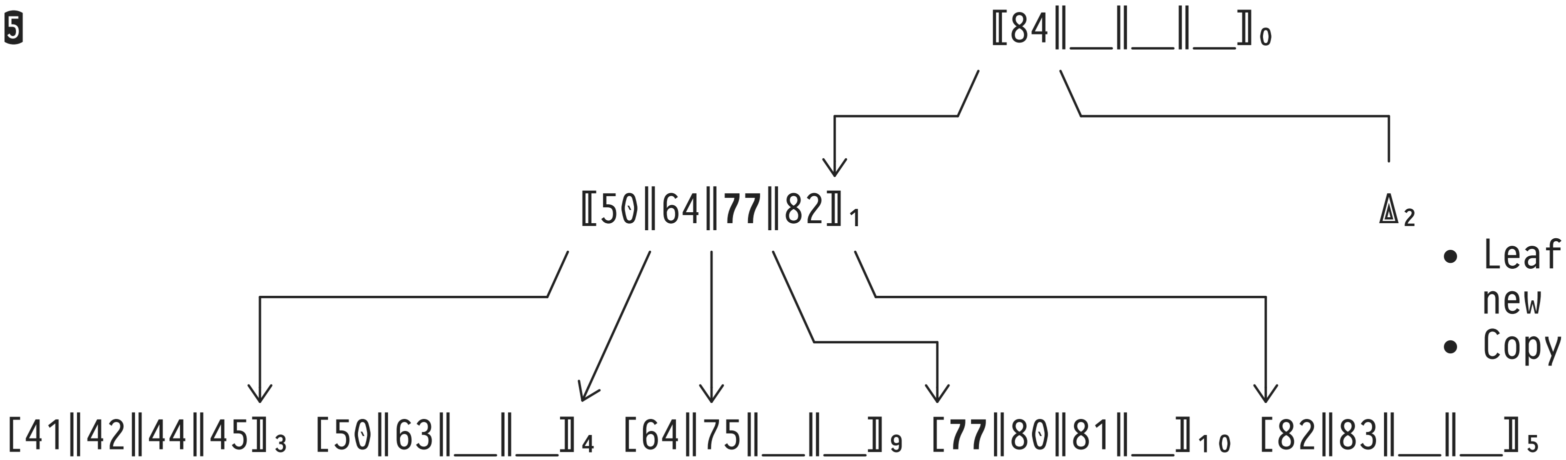
⟦84‖__‖__‖__⟧$_0$

⟦50‖64‖82‖__⟧$_1$ ◬$_2$

[41‖42‖44‖45⟧$_3$ [50‖63‖__‖__⟧$_4$ [64‖75‖80‖81⟧$_9$ [82‖83‖__‖__⟧$_5$

- Search(**77**) returns leaf 9
- Leaf 9 is full (already holds 2×o = 4 entries)

5



⟦84‖__‖__‖__⟧$_0$

⟦50‖64‖**77**‖82⟧$_1$ ◬$_2$

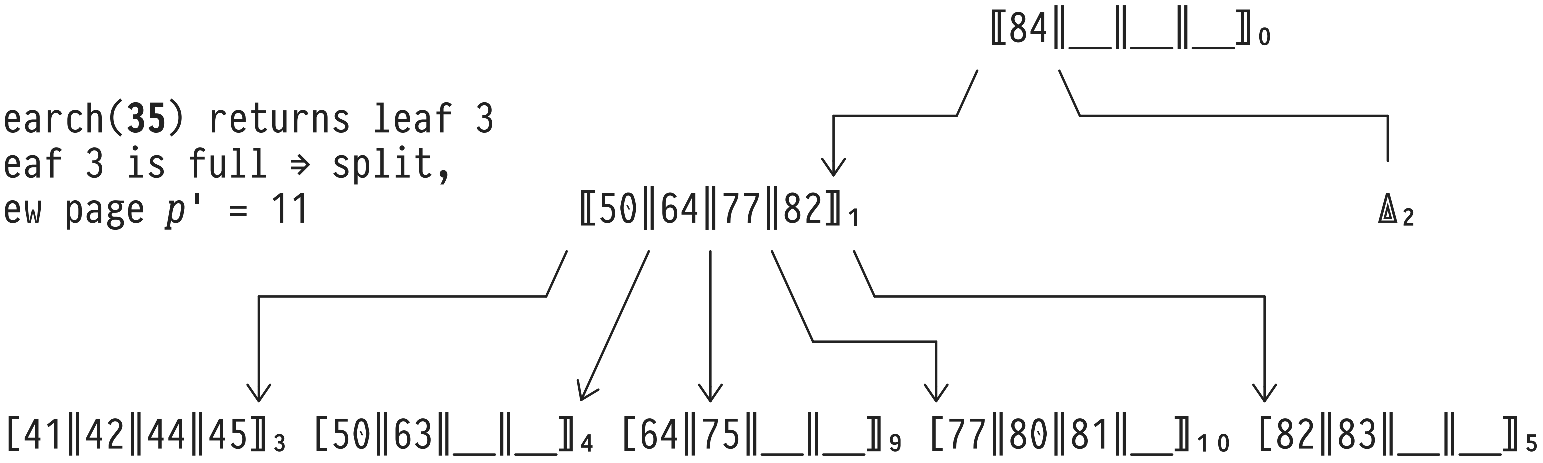
[41‖42‖44‖45⟧$_3$ [50‖63‖__‖__⟧$_4$ [64‖75‖__‖__⟧$_9$ [**77**‖80‖81‖__⟧$_{10}$ [82‖83‖__‖__⟧$_5$

- Leaf 9 has to split, new page *p*' = 10
- Copy up separator [**77**⟧

B^+Tree Insertion Example: Insert ‹35,*rid*›

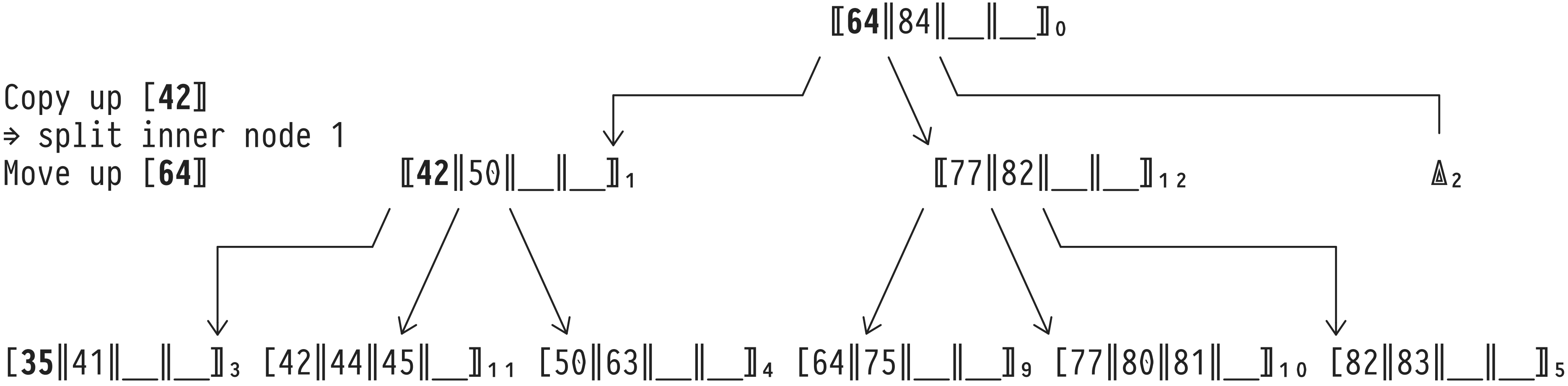
5

- Search(**35**) returns leaf 3
- Leaf 3 is full ⇒ split, new page *p*' = 11



6

- Copy up [**42**⟧ ⇒ split inner node 1
- Move up [**64**⟧



B$^+$Tree Insertion Algorithm[7] (1)

```
TreeInsert(⟨k,rid⟩,node):
  if (node is a leaf)
    return LeafInsert(⟨k,rid⟩,node);
  else
    switch k
      case $k < k_1$
        ⟨sep,ptr⟩ ← TreeInsert(⟨k,rid⟩,$p_0$);        ⎫
      case $k_i \leqslant k < k_{i+1}$                  ⎪
        ⟨sep,ptr⟩ ← TreeInsert(⟨k,rid⟩,$p_i$);        ⎬ see Search()
      case $k_{2d} \leqslant k$                         ⎪
        ⟨sep,ptr⟩ ← TreeInsert(⟨k,rid⟩,$p_{2d}$);     ⎭
    if (sep = ⊥)
      return ⟨⊥,⊥⟩;
    else
      return InnerInsert(⟨sep,ptr⟩,node);
```

[7] Note: $\langle sep,ptr\rangle \equiv [\![sep]\!]$ in our discussion above.

B⁺Tree Insertion Algorithm (2)

```
LeafInsert(⟨$k$,$rid$⟩,$node$):
  if ($node$ has < 2×o entries)
     insert ⟨$k$,$rid$⟩ into $node$;
     return ⟨⊥,⊥⟩;              } ⟨⊥,_⟩ ≡ no upwards split required
  else
     $p'$ ← allocate leaf page;
     [⟨$k_1$,$rid_1$⟩,…,⟨$k_{2o+1}$,$rid_{2o+1}$⟩] ← entries of $node$ ∪ ⟨$k$,$rid$⟩;
     $node$ ← [$k_1$|$rid_1$|…|$k_o$|$rid_o$|__‖__⟧;
     $p'$   ← [$k_{o+1}$|$rid_{o+1}$|…|$k_{2o+1}$|$rid_{2o+1}$|__‖__⟧;
     return ⟨$k_{o+1}$,$p'$⟩;
```

- **Copy** upwards: entry ⟨k_{o+1},rid_{o+1}⟩ remains in leaf p'.

B⁺Tree Insertion Algorithm (3)

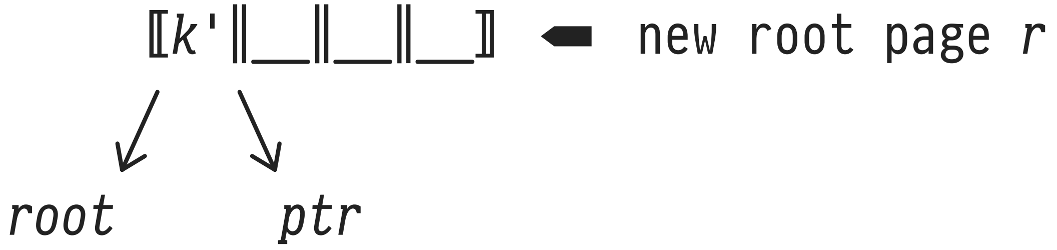
```
InnerInsert(⟨sep,ptr⟩,node):
  if (node has < 2×o entries)
  | insert ⟨sep,ptr⟩ into node;
  | return ⟨⊥,⊥⟩;                  } ⟨⊥,_⟩ ≡ no upwards split required
  else
  | p' ← allocate inner node page;
  | $[p_0,\langle k_1,p_1\rangle,\ldots,\langle k_{2o+1},p_{2o+1}\rangle]$ ← entries of node ∪ ⟨sep,ptr⟩;
  | node ← $[p_0|k_1|p_1|\ldots|k_o|p_o|\_\_\|\_\_]$;
  | p'   ← $[p_{o+1}|k_{o+2}|p_{o+2}|\ldots|k_{2o+1}|p_{2o+1}|\_\_\|\_\_]$;
  ⌊ return $\langle k_{o+1},p'\rangle$;
```

- **Move** upwards: new entry $\langle k_{o+1},p'\rangle$ returned for insertion at parent. No entry $\langle k_{o+1},\_\rangle$ remains at level of *node*/*p'*.

B⁺Tree Insertion Algorithm (Top Level)

Insert(⟨*k*,*rid*⟩) is the top-level B⁺Tree insertion routine:

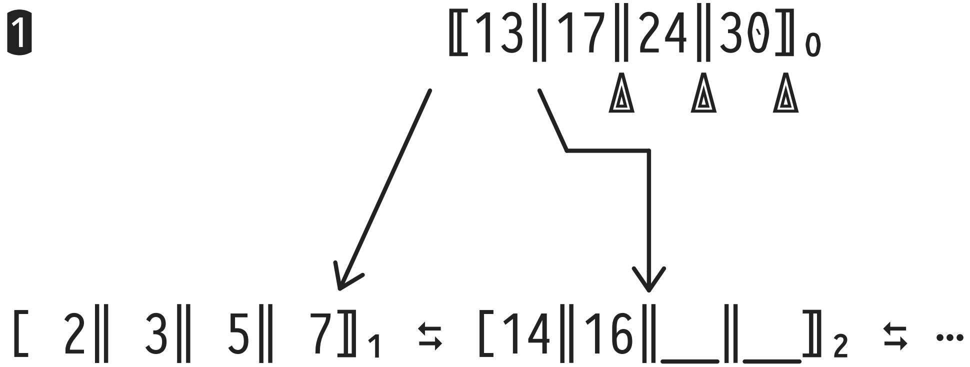
```
Insert(⟨$k$,$rid$⟩):
  ⟨$k'$,$ptr$⟩ ← TreeInsert(⟨$k$,$rid$⟩,$root$);   } $root$ ≡ old root page
  if ($k' \neq \bot$)
  | $r$ ← [$root$|$k'$|$ptr$|__‖__‖__⟧;            } $r$ ≡ new root page
  ⌊ $root$ ← $r$
```



- Note: Insert() may leave us with a new root node that violates the minimum occupancy rule. ¯\(ツ)/¯

B⁺Tree Insertion: Redistribution (1)

Can improve average occupancy and delay height increase on B⁺Tree insertion through **redistribution:**



❶

⟦13‖17‖24‖30⟧$_0$

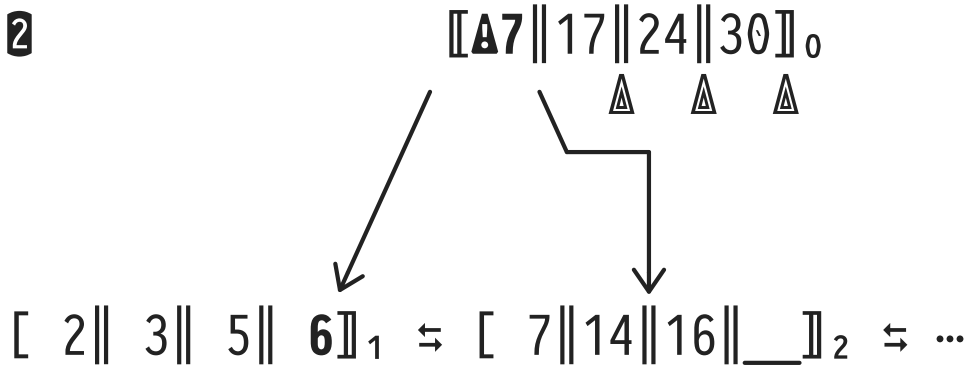
[2‖ 3‖ 5‖ 7⟧$_1$ ⇆ [14‖16‖__‖__⟧$_2$ ⇆ …

❶ Insert new entry ‹**6**,*rid*›

- Search(**6**) returns leaf 1
- Leaf 1 is full, but its right **sibling** 2 has capacity

- Use sequence set chain pointers (⇆) to inspect **sibling** nodes for spare capacity.
- **Push** entry from overflowing node to sibling and ⚠️ **update separator in parent node** to reflect this redistribution.

B⁺Tree Insertion: Redistribution (2)



❷ ⟦⚠**7**‖17‖24‖30⟧$_0$

△ △ △

[2‖ 3‖ 5‖ **6**⟧$_1$ ⇆ [7‖14‖16‖___⟧$_2$ ⇆ …

❷ Push entry ‹7,*rid*'› to leaf 2
- Place ‹**6**,*rid*› in leaf 1
- Update separator (13 → **7**) in parent node 0
- B⁺Tree remains at height 2

- Inspecting node sibling involves additional page I/O. 👎
- Actual implementations use redistribution on the index leaf level only (if at all).

PostgreSQL does not appear to implement redistribution at all.

7 ¦ B$^+$Tree Deletion of Entry With Key *k* (Sketch)

1. Use Search(*k*) to **find the leaf** *p* holding entry ‹*k*,*rid*›.
2. **Simply delete** ‹*k*,*rid*› from *p*.[8]
3. If *p* now holds < o entries, leaf *p* **underflows.** Any sibling of *p* with spare entries?
   - Yes, use **redistribution** to move an entry into *p*.
   - No, **merge** *p* and a sibling leaf *p*' of o entries. Delete↻ the now obsolete separator of *p* and *p*' in their parent node.

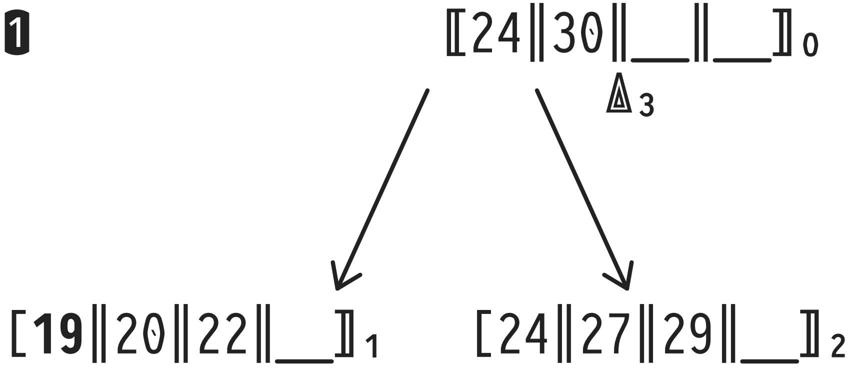
     Deletion propagates upwards and may eventually leave the root node empty (decreases B$^+$Tree height).

[8] **Q:** If ‹*k*,*rid*› is the leftmost entry in *p*, do we need to update the associated separator entry in *p*'s parent node? Why not?

Removal of $\langle k, rid \rangle$ from p does not need any parent node update. The search tree property of the existing separators in the parent remains intact after removal.

B⁺Tree Deletion (No Underflow)

❶



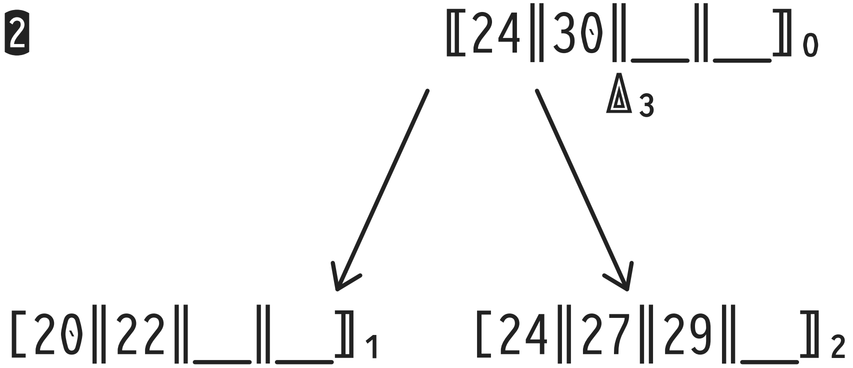
⟦24‖30‖__‖__⟧$_0$ 𝔸$_3$

[**19**‖20‖22‖__⟧$_1$ [24‖27‖29‖__⟧$_2$

❶ Delete entry with key *k* = **19**

- Search(**19**) returns leaf 1
- Leaf 1 has > o entries, node will not underflow

❷



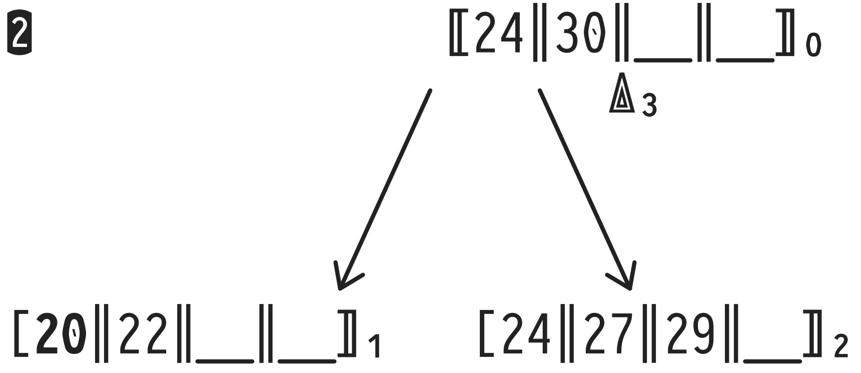
⟦24‖30‖__‖__⟧$_0$ 𝔸$_3$

[20‖22‖__‖__⟧$_1$ [24‖27‖29‖__⟧$_2$

❷ Simply delete entry ‹**19**,*rid*› from leaf 1

B⁺Tree Deletion and Redistribution

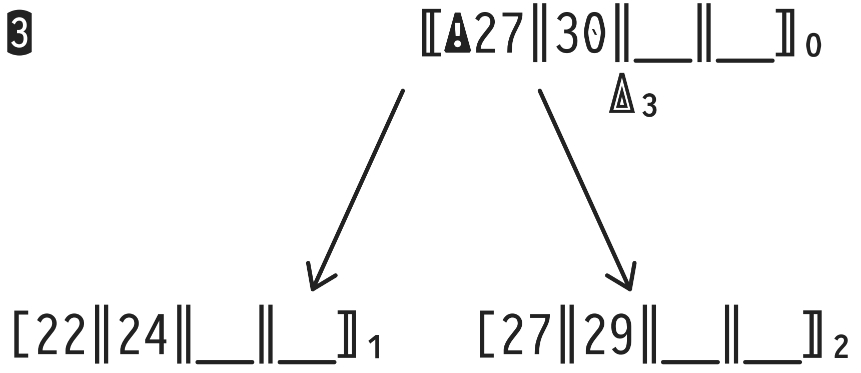
❷



❷ Delete entry with key *k* = **20**

- Search(**20**) returns leaf 1
- Leaf 1 has minimum occupancy of o entries ⇛ will underflow

❸

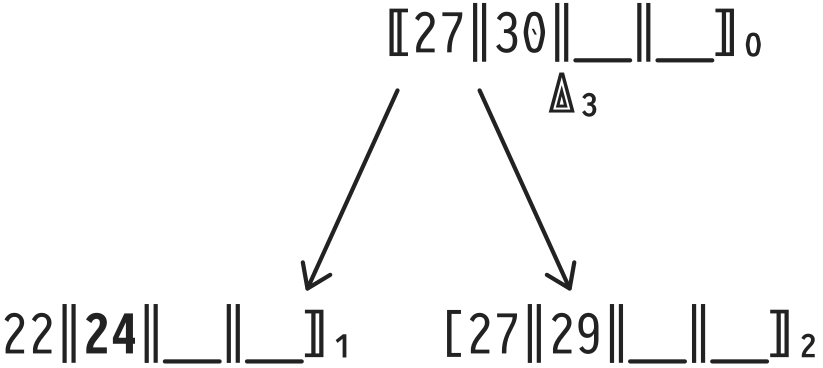


❸ Sibling *p*' = 2 has one entry to spare ⇛ redistribution

- Move entry ‹24,*rid*'› from leaf 2 to leaf 1
- Update separator (24 → 27) in parent node 0

B$^+$Tree Deletion and Leaf Node Merging

❸



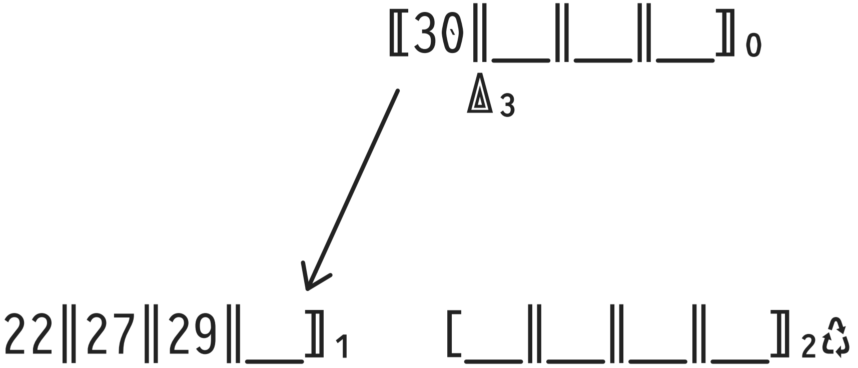
⟦27‖30‖__‖__⟧$_0$ △$_3$

[22‖**24**‖__‖__⟧$_1$ [27‖29‖__‖__⟧$_2$

❸ Delete entry with key *k* = **24**

- Search(**24**) returns leaf 1
- Leaf 1 has minimum occupancy, no sibling with spare entries

❹



⟦30‖__‖__‖__⟧$_0$ △$_3$

[22‖27‖29‖__⟧$_1$ [__‖__‖__‖__⟧$_2$♻

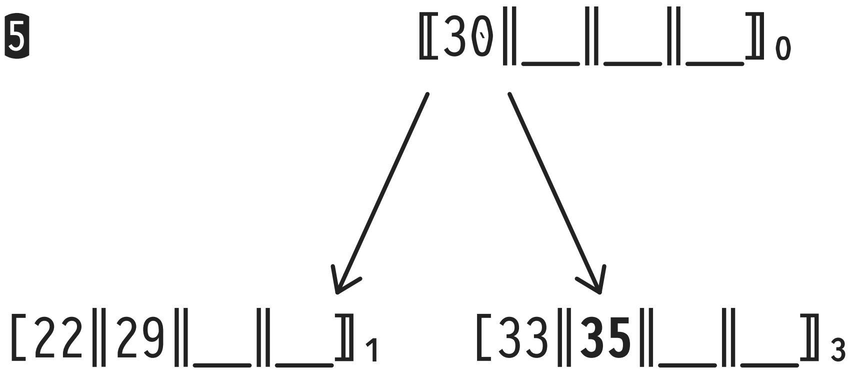
❹ Merge leaf nodes 1 and 2, mark empty page 2 as garbage

- In parent 0, delete↺ obsolete separator [27⟧

Assume step 5 (not shown in slides) that deletes entry with $k = 27$ from leaf 1. In the mean time, empty leaf page 2 has been garbage collected.

B⁺Tree Deletion and Leaf Node Merging (Empty Root)

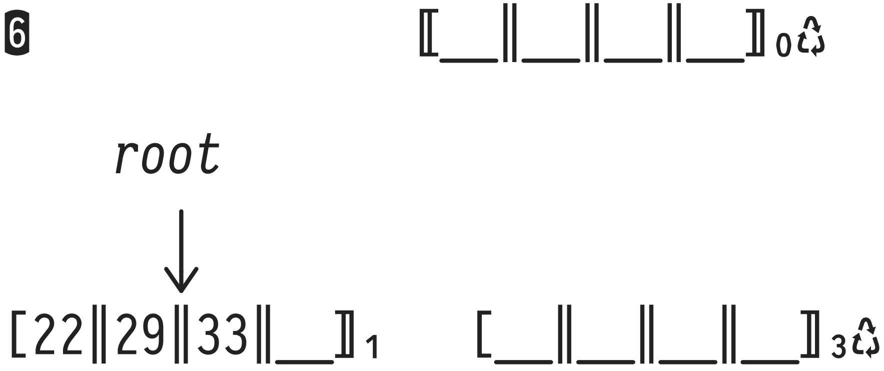
❺



❹ Delete entry with key k = **35**

- Search(**35**) returns leaf 3
- Leaf 3 has minimum occupancy, no sibling with spare entries

❻

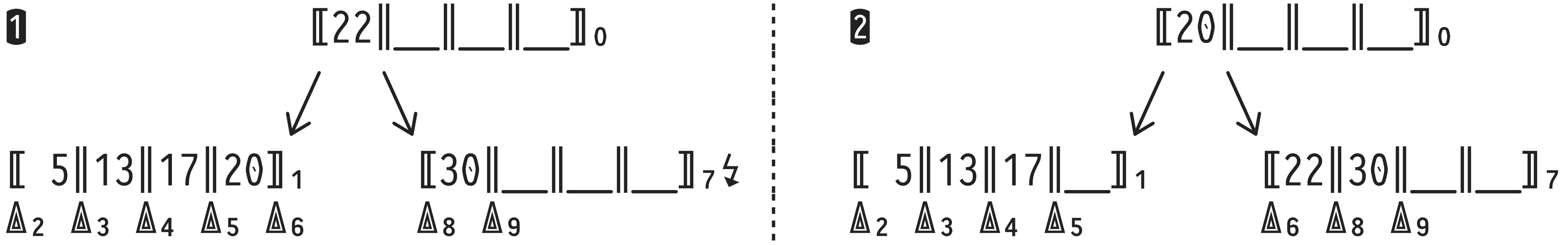


❹ Merge leaf nodes 1 and 3, mark empty page 3 as garbage

- In parent 0, deleteϑ obsolete separator [30⟧
- Old root empty (⇒ garbage), mark page 1 as the new root
- B⁺Tree height decreases

B$^+$Tree Deletion and Inner Node Redistribution

- **Redistribution** is also defined for **inner nodes.** Suppose we encounter underflow ❶ during ⟲ deletion propagation:



1

⟦22‖__‖__‖__⟧$_0$

⟦ 5‖13‖17‖20⟧$_1$ ⟦30‖__‖__‖__⟧$_7$ ↯

$\triangle_2$ $\triangle_3$ $\triangle_4$ $\triangle_5$ $\triangle_6$ $\triangle_8$ $\triangle_9$

2

⟦20‖__‖__‖__⟧$_0$

⟦ 5‖13‖17‖__⟧$_1$ ⟦22‖30‖__‖__⟧$_7$

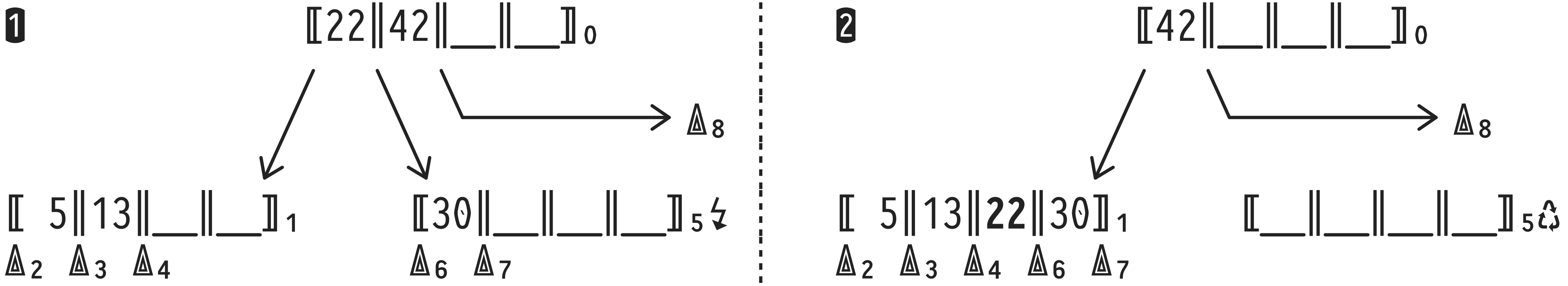
$\triangle_2$ $\triangle_3$ $\triangle_4$ $\triangle_5$ $\triangle_6$ $\triangle_8$ $\triangle_9$

- Inner node 1 has two spare entries. “Rotate entry [20] through parent” to underflowed inner node 7.

  N.B.: Semantics of subtree $\triangle_6$ (holds index entries with $k \geqslant 20 \wedge k < 22$) are preserved.

B⁺Tree Deletion and Inner Node Merging

- Likewise, **inner nodes** may also be **merged.** The underflow in ❶ cannot be handled by redistribution:



❶ ⟦22‖42‖__‖__⟧$_0$

⟦ 5‖13‖__‖__⟧$_1$ ⟦30‖__‖__‖__⟧$_5$ ϟ

Δ_2 Δ_3 Δ_4 Δ_6 Δ_7 Δ_8

❷ ⟦42‖__‖__‖__⟧$_0$

⟦ 5‖13‖**22**‖30⟧$_1$ ⟦__‖__‖__‖__⟧$_5$ ♲

Δ_2 Δ_3 Δ_4 Δ_6 Δ_7 Δ_8

- Note how the separator **22** has been **pulled down**↻ from the parent to discriminate between subtrees Δ_4 and Δ_6:
  - Δ_4: $k \geqslant 13 \land k < 22$
  - Δ_6: $k \geqslant 22 \land k < 30$

8 ¦ B$^+$Trees: Key Compression

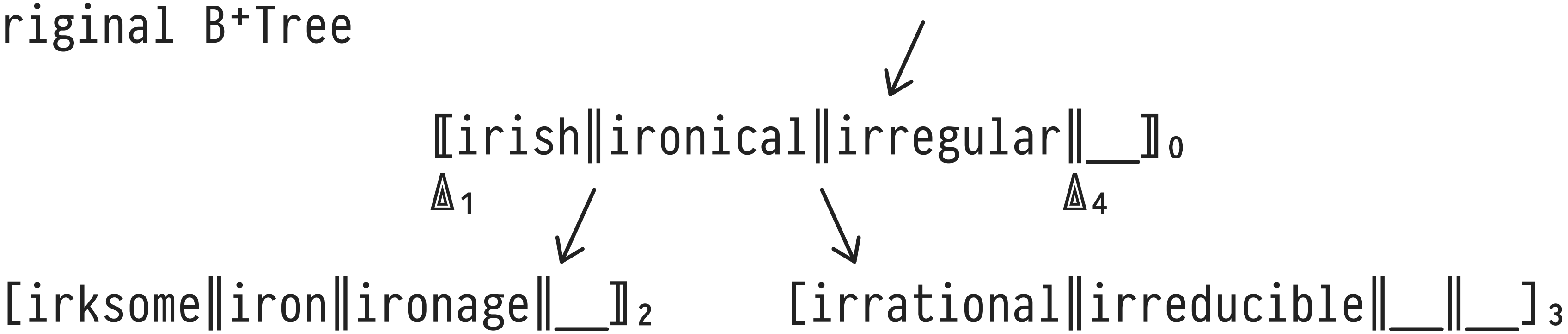
The higher the **fan-out** F, the more index entries fit in a B$^+$Tree of fixed height. How to maximize F?

- For entries $\langle k,p \rangle$ in indexes over text/char columns, we may have $|k| \gg |p|$.[9] Can we reduce the size of k?
- 💡 Search() and TreeInsert() do *not* inspect the actual key values but only use $<$/$\leqslant$ to direct tree traversals.
    - ⇒ May **shorten (truncate) string keys** as long as the ordering relation is preserved.
    - This applies to index entries inner nodes only. Leaf level keys remain as is.

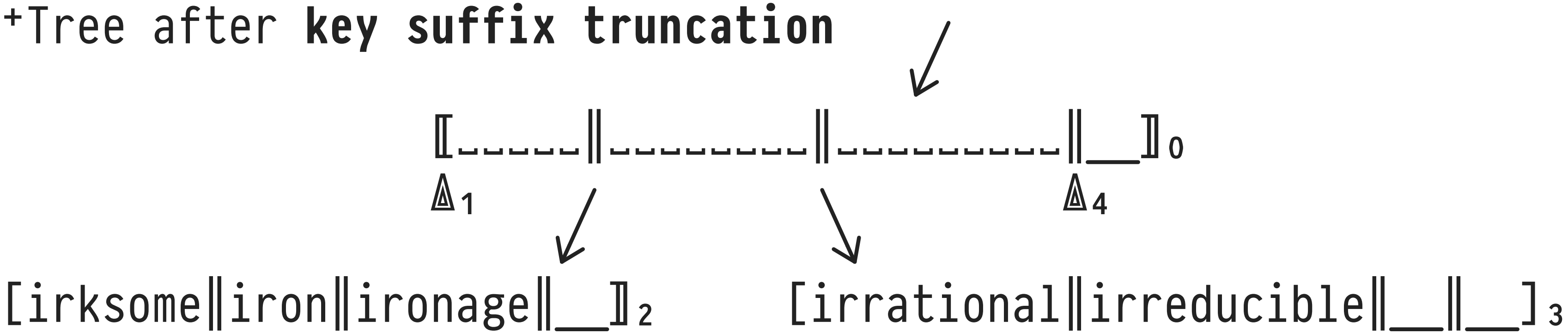
[9] The implementation (thus size) of page pointers p is prescribed by the DBMS. Nothing to win here.

B+Trees: Key Suffix Truncation

❶ Original B+Tree


⟦irish‖ironical‖irregular‖__⟧0
Δ1
Δ4
[irksome‖iron‖ironage‖__⟧2
[irrational‖irreducible‖__‖__]3

❷ B+Tree after **key suffix truncation**



⚠️ While truncating, preserve the **separator** semantics.

Must examine largest key value in subtree left of key to compress (truncation makes key value smaller lexicographically):

```
❷ B⁺Tree after key suffix truncation    /
                                       v
                    ⟦ir␣␣␣‖ironi␣␣␣‖irreg␣␣␣␣␣‖__⟧₀
                    ⍋₁    /        \          ⍋₄
                         v          v
   [irksome‖iron‖ironage‖__⟧₂      [irrational‖irreducible‖__‖__]₃
```

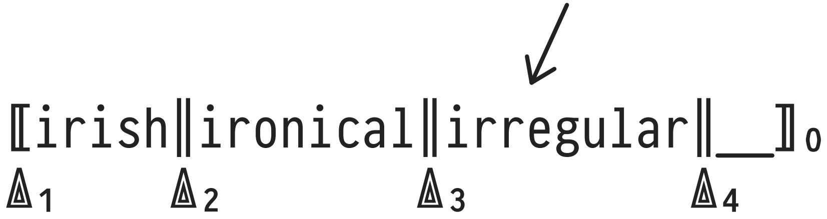
- Separators are ordered: ir < ironi < irred
- Search tree semantics of node 2: ir ⩽ {irksome, iron, ironage}
- Search tree semantics of node 2: {irksome, iron, ironage} < ironi (but **not** < iron ⚠)
- Search tree semantics of node 3: ironi ⩽ {irrational, irreducible}
- Search tree semantics of node 3: {irrational, irreducible} < irreg

B+Trees: Key Prefix Compression

Observation: string keys within a B+Tree inner node often **share a common prefix.**

- 💡 Store common prefix only once (e.g., as "k_0").
- Violating the 50% occupancy rule can help compression.

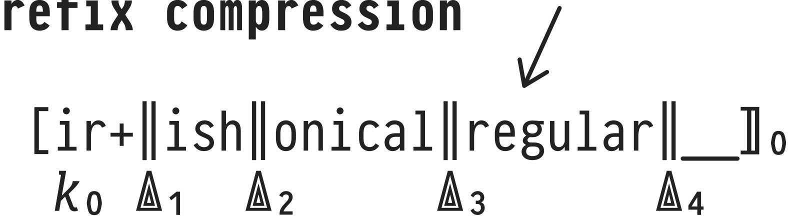
❶ Original B+Tree



⟦irish‖ironical‖irregular‖__⟧$_0$

$\triangle_1$ $\triangle_2$ $\triangle_3$ $\triangle_4$

❷ B+Tree after **key prefix compression**



[ir+‖ish‖onical‖regular‖__⟧$_0$

k_0 $\triangle_1$ $\triangle_2$ $\triangle_3$ $\triangle_4$

9 ¦ B^+Tree Bulk Loading

Grab a hot cup of ☕ and start a war on Stack Overflow:[10]

Q: *Which order of operations is better?*

```
❶ CREATE TABLE T (…);
❷ INSERT INTO T VALUES (‹5 × 10^6 rows›);
❸ CREATE INDEX I ON T USING btree (…);
```

or

```
❶ CREATE TABLE T (…);
❸ CREATE INDEX I ON T USING btree (…);      ◀┐
❷ INSERT INTO T VALUES (‹5 × 10^6 rows›);   ◀┘
```

[10] See, for example, https://stackoverflow.com/questions/5910486/indexes-on-a-table-database

Experiment: time both order of operations (index creation before/after populating the associated table):

```
-- ❶ Populate table, then create index
  DROP TABLE IF EXISTS indexed;
  CREATE TABLE indexed (a int, b text, c numeric(3,2));  -- no PRIMARY KEY!

  INSERT INTO indexed(a,b,c)
    SELECT i, md5(i::text), sin(i)
    FROM   generate_series(1,5000000) AS i;

  CREATE INDEX indexed_a ON indexed USING btree (a);
```

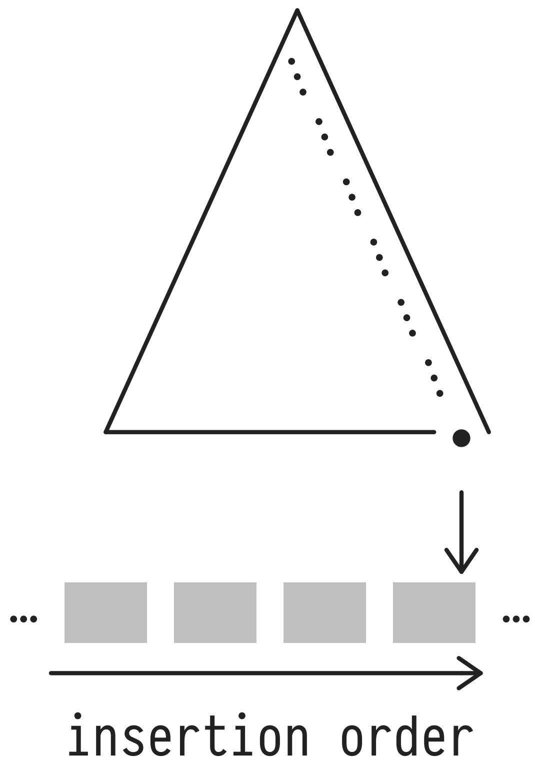
- Observed times:
    - Populate table: Time: 34656.176 ms (00:34.656)
    - Index creation: Time: 4569.847 ms (00:04.570) ⁂
    - Σ 39 s

```
-- ❷ Create index, then populate table

  DROP TABLE IF EXISTS indexed;
  CREATE TABLE indexed (a int, b text, c numeric(3,2));  -- no PRIMARY KEY!

  CREATE INDEX indexed_a ON indexed USING btree (a);

  INSERT INTO indexed(a,b,c)
    SELECT i, md5(i::text), sin(i)
    FROM   generate_series(1,5000000) AS i;
```

- Observed times:
    - Index creation: Time: 2.897 ms
    - Populate table: Time: 55233.198 ms (00:55.233)
    - Σ 55 s
- In option ❷, pages of the table and of the index compete for space in the buffer. Traverse the growing B⁺Tree from its root down to the leaf level 50,000,000 times.

B^+Tree Bulk Loading

If insertions happen in index key order (i.e., ascending values of *k*), we observe a particular B^+Tree access pattern:



heap file (sorted on keys *k*)

- TreeInsert() will always traverse path ⋱, will always hit the righmost leaf.

⇒ Fix rightmost leaf in buffer, insert next entry right there (*no* traversal from root). Node splits only occur along path ⋱.

- We effectively create a clustered index.

PostgreSQL documentation on index bulk loading suport (in file src/backend/access/nbtree/README):

```
Fastpath For Index Insertion
----------------------------

We optimize for a common case of insertion of increasing index key
values by caching the last page to which this backend inserted the last
value, if this page was the rightmost leaf page. For the next insert, we
can then quickly check if the cached page is still the rightmost leaf
page and also the correct place to hold the current value. We can avoid
the cost of walking down the tree in such common cases.
```

- If we make life hard for PostgreSQL and insert keys in *descending* order, we cannot benefit from the bulk loading fast path. Experiment:

```
DROP TABLE IF EXISTS indexed;
CREATE TABLE indexed (a int, b text, c numeric(3,2));

INSERT INTO indexed(a,b,c)
  SELECT i, md5(i::text), sin(i)
  FROM   generate_series(5000000,1,-1) AS i; -- descending order keys in heap file

SELECT i.ctid, i.*
FROM   indexed AS i
LIMIT  10;
```

```
  ctid  |    a    |                b                 |   c
--------+---------+----------------------------------+-------
 (0,1)  | 5000000 | d1524adbbd8eed2bf4d424a311a3c6fd | -0.98
 (0,2)  | 4999999 | d9ef05881dece9e118a8c8256d10a5fb | -0.35
 (0,3)  | 4999998 | 218e4c17645afbac89c374b7d17106c1 |  0.60
 (0,4)  | 4999997 | 41d102a9acfbb62bb6597ba31b174071 |  1.00
 (0,5)  | 4999996 | d38781884f602f357a3646016860f2e7 |  0.48
 (0,6)  | 4999995 | c8db87704b006bd06f7fdf4f9e08014b | -0.48
 (0,7)  | 4999994 | 43ded31a0364a7795d8096c1dc461f48 | -1.00
 (0,8)  | 4999993 | 0fe2697a372f2afc17ec4c47b1c8eb06 | -0.59
 (0,9)  | 4999992 | 6a3318fe75d882af52a50010256132e0 |  0.36
 (0,10) | 4999991 | 429ebf4515e29f9189918a5b4260ffe8 |  0.98
```

```
  CREATE INDEX indexed_a ON indexed USING btree (a);
CREATE INDEX
Time: 7164.370 ms (00:07.164)      ⬅ slower! (formerly 4569.847 ms, see ⁂ above)
```

10 ┆ Q_8 — Filtering a Table

```
SELECT i.b, i.c
FROM   indexed AS i
WHERE  i.a = 42 [i.c = 0.42] -- either filter on i.a or i.c
```

Indexes in MonetDB play a secondary role and are *not* organized in tree shapes.

MMDBMSs try to exploit that data resides in directly adressable memory and primarily aim to avoid access to separate index data structures (to avoid pointer chasing and potential cache misses).

Using EXPLAIN on Q_8: Filter on Column a

```
sql> EXPLAIN SELECT i.b, t.c
             FROM   indexed AS i
             WHERE  i.a = 42;
   ⋮
  indexed :bat[:oid] := sql.tid(sql, "sys", "indexed");
  a0      :bat[:int] := sql.bind(sql, "sys", "indexed", "a", 0:int);
  p1      :bat[:oid] := algebra.thetaselect(a0, indexed, 42:int, "=="); ⬅ ≡ a = 42
  c0      :bat[:sht] := sql.bind(sql, "sys", "indexed", "c", 0:int);
  c       :bat[:sht] := algebra.projection(p1, c0);
  b0      :bat[:str] := sql.bind(sql, "sys", "indexed", "b", 0:int);
  b       :bat[:str] := algebra.projection(p1, b0);
   ⋮
```

- MonetDB uses `algebra.thetaselect(…, 42:int, "==")` to implement the predicate filter.

Using EXPLAIN on Q_8: Filter on Column c[11]

```
sql> EXPLAIN SELECT i.b, t.c
             FROM   indexed AS i
             WHERE  i.c = 0.42;
   ⋮
  indexed :bat[:oid] := sql.tid(sql, "sys", "indexed");
  c0      :bat[:sht] := sql.bind(sql, "sys", "indexed", "c", 0:int);
  p1      :bat[:oid] := algebra.thetaselect(c0, indexed, 42:int, "=="); ⬅ ≡ c = 0.42
  c       :bat[:sht] := algebra.projection(p1, c0);
  b0      :bat[:str] := sql.bind(sql, "sys", "indexed", "b", 0:int);
  b       :bat[:str] := algebra.projection(p1, b0);
   ⋮
```

- Plan is nearly identical (modulo access to the a BAT).
- MonetDB *appears* to use the same `algebra.thetaselect(…, 42:int, "==")` MAL operation.

[11] Note how MonetDB maps the domain of type numeric(3,2) of column c, i.e., the set $N_{3,2} \equiv \{-9.99,…,9.99\}$ with $|N_{3,2}| = 1999$, to a 16-bit value of type :sht. Nifty.

BAT Tail Properties

When MonetDB constructs a BAT *t*, a family of tail column **properties** *prop*(*t*) is derived/maintained:[12]

BAT Property *prop*(*t*)	Description
dense (tails of type :oid only)	ascending values, no gaps
key	unique values
sorted	ascending values
revsorted	descending values
nil/nonil	at least one/no nil value

- Use bat.info(*t*) to inspect current properties of *t*.
- ⚠️ Incomplete: *t*'s tail may be sorted although sorted(*t*) = false (⇒ but not ⇔).

[12] Additional properties nokey, nosorted, norevsorted give “proofs” (tail positions) why property does not hold. Example: nosorted = 3 ≡ tail value for row 3@0 < tail value for row 2@0.

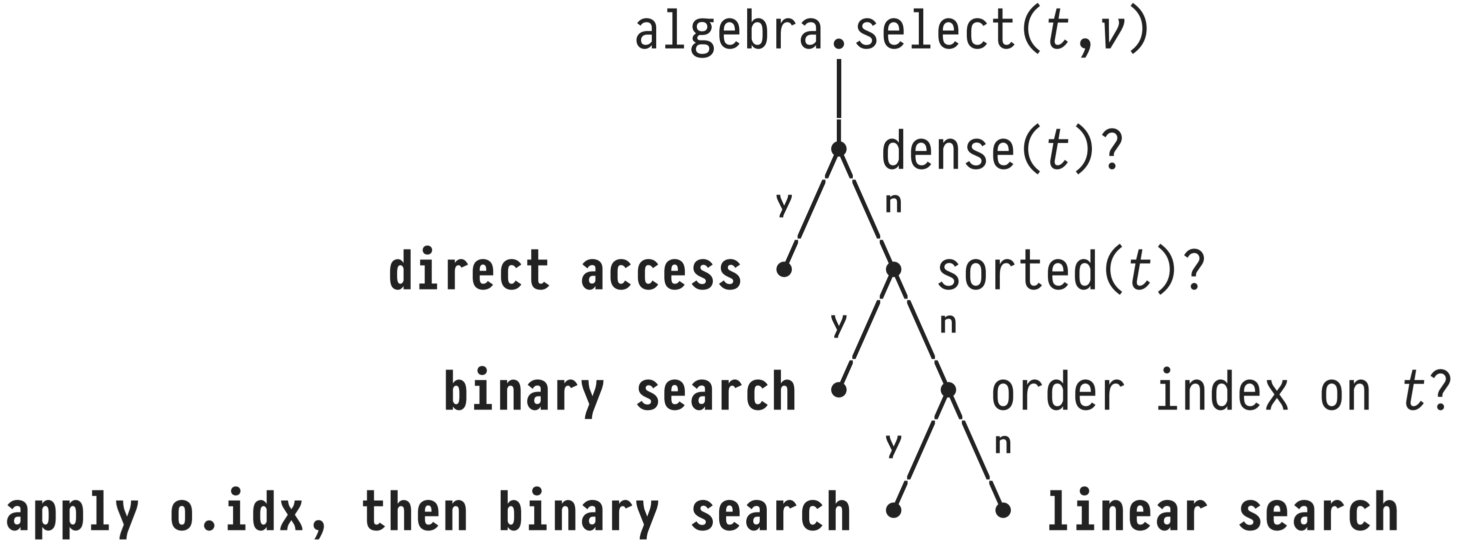
Q: What does `sorted(t) ∧ revsorted(t)` indicate? (**A:** constant tail column)

Demonstrate that property inference is incomplete. Below, MonetDB missed to re-establish sorted() after the bat.delete():

```
$ mclient -d scratch -l mal

 t := bat.new(:int);
 bat.append(t, 1);
 bat.append(t, 2);
 bat.append(t, 3);
 io.print(t);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  1  ]
[ 1@0,  2  ]
[ 2@0,  3  ]

 (i1,i2) := bat.info(t);
 io.print(i1,i2);
#--------------------------#
# t t t  # name
# void  str str  # type
#--------------------------#
[...]
[ 15@0, "tdense", "0"  ]
[ 16@0, "tseqbase", "0@0"  ]
[ 17@0, "tsorted",  "1"  ]          ⬅ :-)
[ 18@0, "trevsorted", "0"  ]        ⬅ :-)
[ 19@0, "tkey", "1"  ]              ⬅ :-)
[ 20@0, "tvarsized",  "0"  ]
[ 21@0, "tnosorted",  "0"  ]
[ 22@0, "tnorevsorted", "1"  ]      ⬅  proof: 2 (1@0) > 1 (0@0)
[ 23@0, "tnodense", "0"  ]
[ 24@0, "tnokey[0]",  "0"  ]
[ 25@0, "tnokey[1]",  "0"  ]
[ 26@0, "tnonil", "1"  ]            ⬅ :-)
[ 27@0, "tnil", "0"  ]              ⬅ :-)

 bat.append(t, 5);
 bat.append(t, 4);
 io.print(t);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  1  ]
[ 1@0,  2  ]
[ 2@0,  3  ]
[ 3@0,  5  ]
[ 4@0,  4  ]

 (i1,i2) := bat.info(t);
 io.print(i1,i2);

#-------------------------#
# t t t  # name
# void  str str  # type
#-------------------------#
[...]
[ 15@0, "tdense", "0"  ]
[ 16@0, "tseqbase", "0@0"  ]
[ 17@0, "tsorted",  "0"  ]           ⬅ :-)
[ 18@0, "trevsorted", "0"  ]         ⬅ :-)
[ 19@0, "tkey", "0"  ]               ⬅ :-( should be 1
[ 20@0, "tvarsized",  "0"  ]
[ 21@0, "tnosorted",  "4"  ]         ⬅ proof: 4 (4@0) < 5 (3@0)
[ 22@0, "tnorevsorted", "1"  ]
[ 23@0, "tnodense", "0"  ]
[ 24@0, "tnokey[0]",  "0"  ]
[ 25@0, "tnokey[1]",  "0"  ]
[ 26@0, "tnonil", "1"  ]             ⬅ :-)
[ 27@0, "tnil", "0"  ]               ⬅ :-)
[...]

 bat.delete(t, 3@0);
 io.print(t);
#-------------------------#
# h t  # name
# void  int  # type
#-------------------------#
[ 0@0,  1  ]
[ 1@0,  2  ]
[ 2@0,  3  ]
[ 3@0,  4  ]

mal>(i1,i2) := bat.info(t);
mal>io.print(i1,i2);
#-------------------------#
# t t t  # name
# void  str str  # type
#-------------------------#
[...]
[ 15@0, "tdense", "0"  ]
[ 16@0, "tseqbase", "0@0"  ]
[ 17@0, "tsorted",  "0"  ]           ⬅ :-( should be 1
[ 18@0, "trevsorted", "0"  ]        ⬅ :-)
[ 19@0, "tkey", "0"  ]              ⬅ :-( should be 1
[ 20@0, "tvarsized",  "0"  ]
[ 21@0, "tnosorted",  "0"  ]        ⬅ at least no proof... (0 ≡ no proof)
[ 22@0, "tnorevsorted", "1"  ]
[ 23@0, "tnodense", "0"  ]
[ 24@0, "tnokey[0]",  "0"  ]
[ 25@0, "tnokey[1]",  "0"  ]
[ 26@0, "tnonil", "1"  ]            ⬅ :-)
[ 27@0, "tnil", "0"  ]              ⬅ :-)
[...]
```

MonetDB: Tactical Optimization

MAL operations inspect BAT properties at *query runtime*, select one of several efficient implementations:



algebra.select(*t*,*v*)

dense(*t*)?

y

n

direct access

sorted(*t*)?

y

n

binary search

order index on *t*?

y

n

apply o.idx, then binary search

linear search

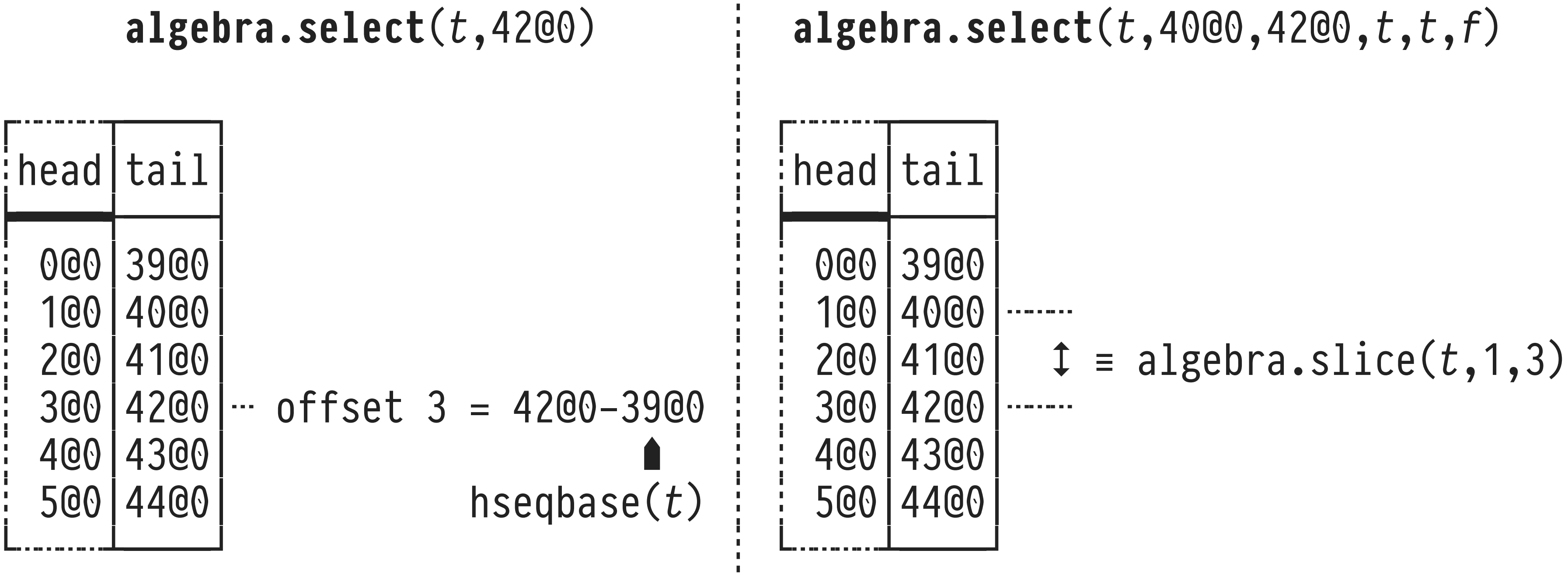
- This is coined **tactical optimization** (as opposed to strategical query optimization at *query compile time*).

Demonstrate tactical optimization in algebra.select(). Restart mserver5 with option --algorithms. Evaluate query Q_8 with filter on i.a (*sorted*) and i.c (creates order index on the fly).

```
$ mserver5 --dbpath=(pwd)/data/scratch --set monet_vault_key=(pwd)/data/scratch/.vaultkey --algorithms
[...]

(other terminal) $ mclient -d scratch -l sql

 SELECT i.b, i.c
 FROM   indexed AS i
 WHERE  i.a = 42;
+----------------------------------+-------+
| b                                | c     |
+==================================+=======+
| a1d0c6e83f027327d8461063f4ac58a6 | -0.92 |
+----------------------------------+-------+

(server terminal)
 [...]
 #BATselect(b=tmp_2155#1000000,s=tmp_344(dense),anti=0): sorted ◀
 [...]                                      ▲

(other terminal)
 SELECT i.b, i.c
 FROM   indexed AS i
 WHERE  i.c = 0.42;
+----------------------------------+-------+
| b                                | c     |
+==================================+=======+
| c51ce410c124a10e0db5e4b97fc2af39 |  0.42 |
| 58238e9ae2dd305d79c2ebc8c1883422 |  0.42 |
| 08419be897405321542838d77f855226 |  0.42 |
| f4dd765c12f2ef67f98f3558c282a9cd |  0.42 |
[...]
| 8080391ec648947e0d95dea7745fb0e0 |  0.42 |
| cf187e1ad5747afe972e5e59bb308cce |  0.42 |
+----------------------------------+-------+
3531 tuples (18.180ms)

(server terminal)
 [...]
#BATcheckorderidx: reusing persisted orderidx 1407
#BATsubselect(b=tmp_2577#1000000,s=tmp_1146(dense),anti=0): orderidx ◀
 [...]                                         ▲
```

The Tactics of algebra.select: dense(*t*)

If input BAT *t* is **dense**, use **positional access** and **slicing** to evaluate equality and range selections:



algebra.select(*t*,42@0)

head	tail
0@0	39@0
1@0	40@0
2@0	41@0
3@0	42@0
4@0	43@0
5@0	44@0

offset 3 = 42@0-39@0

hseqbase(*t*)

algebra.select(*t*,40@0,42@0,*t*,*t*,*f*)

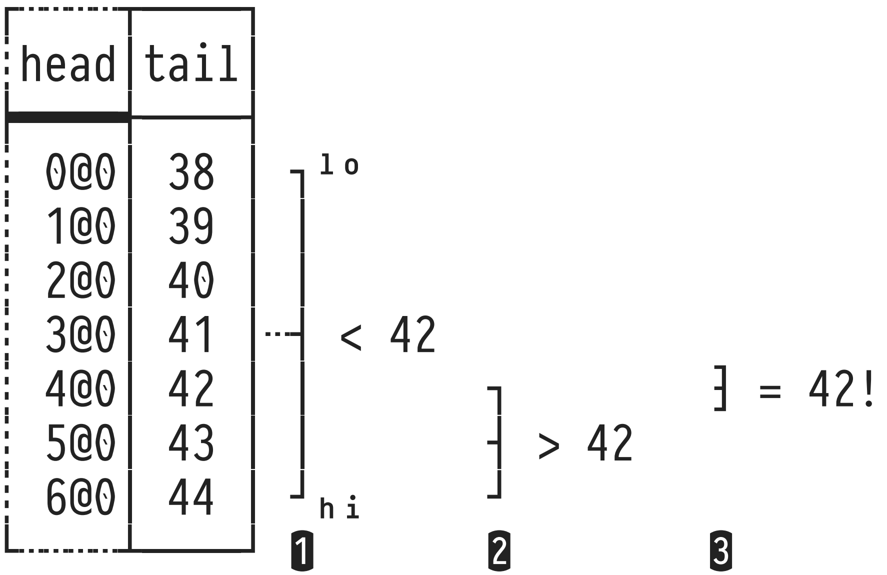
head	tail
0@0	39@0
1@0	40@0
2@0	41@0
3@0	42@0
4@0	43@0
5@0	44@0

↕ ≡ algebra.slice(*t*,1,3)

```
algebra.select(t,4000,4200,t,t,f): 4000 ⩽ x ⩽ 4200
```

The Tactics of algebra.select: sorted(t)

algebra.select(t,42)



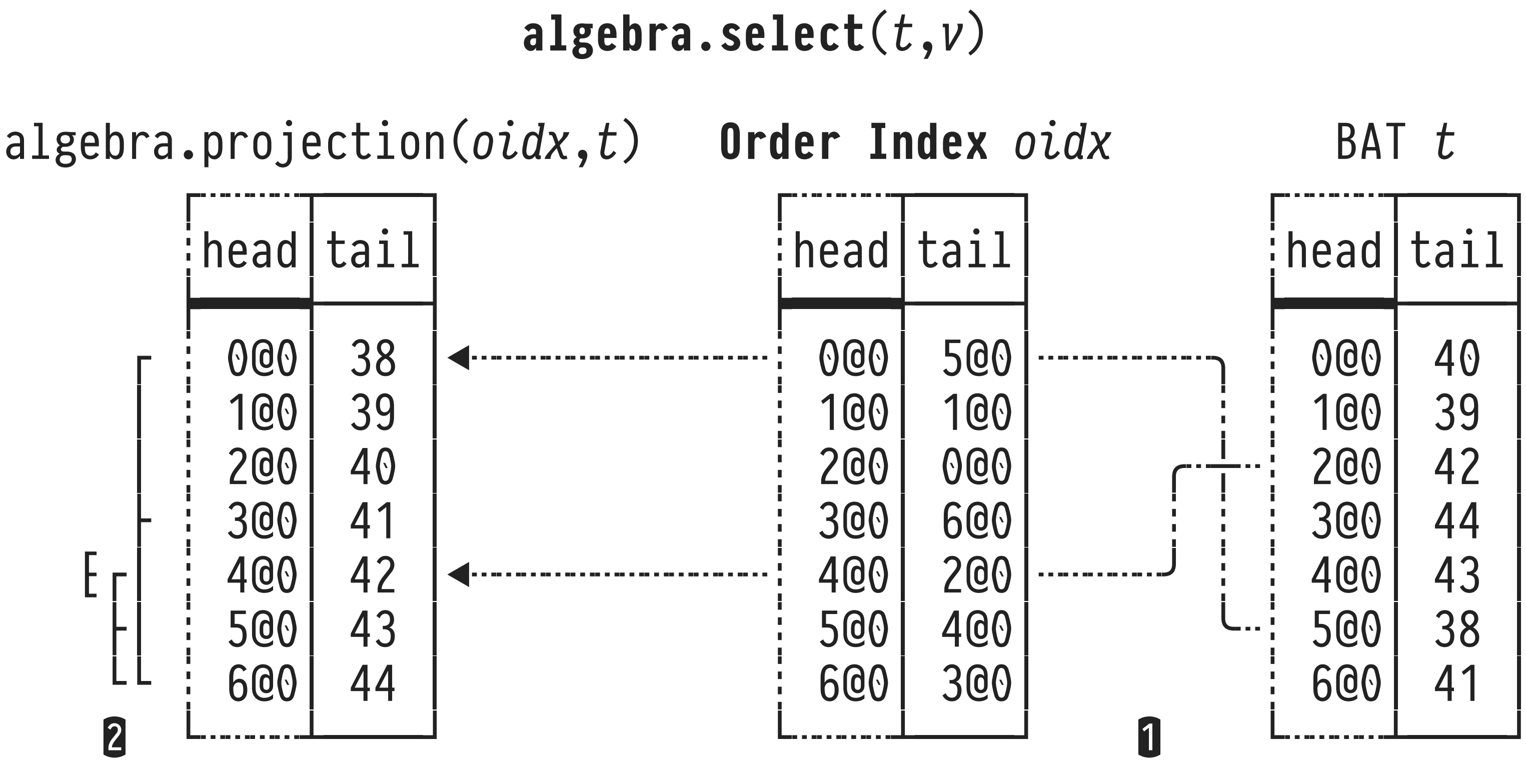
head	tail
0@0	38
1@0	39
2@0	40
3@0	41
4@0	42
5@0	43
6@0	44

Binary Search:

- Test middle value (pivot) between limits *lo* and *hi*
- Recurse into upper or lower partition based on test
- Finishes in $\log_2(|t|)$ steps

- **N.B.:** Unpredictable branches (≶ 42?) and jumps of pivot position less than ideal for CPU.

The Tactics of algebra.select: Order Indexes



algebra.select(t,v)

algebra.projection($oidx$,t)

head	tail
0@0	38
1@0	39
2@0	40
3@0	41
4@0	42
5@0	43
6@0	44

Order Index $oidx$

head	tail
0@0	5@0
1@0	1@0
2@0	0@0
3@0	6@0
4@0	2@0
5@0	4@0
6@0	3@0

BAT t

head	tail
0@0	40
1@0	39
2@0	42
3@0	44
4@0	43
5@0	38
6@0	41

- Row [i@0,j@0] ∈ $oidx$: value at offset j is ith largest in tail. Tactic: ❶ Apply $oidx$, ❷ then use binary search.

Demonstrate the use of order indexes in MAL. Replay example of previous slide:

```
$ mclient -d scratch -l mal

 include orderidx;

 t := bat.new(:int);
 bat.append(t, 40);
 bat.append(t, 39);
 bat.append(t, 42);
 bat.append(t, 44);
 bat.append(t, 43);
 bat.append(t, 38);
 bat.append(t, 41);
 io.print(t);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  40  ]
[ 1@0,  39  ]
[ 2@0,  42  ]
[ 3@0,  44  ]
[ 4@0,  43  ]
[ 5@0,  38  ]
[ 6@0,  41  ]


 bat.orderidx(t);
 oidx :bat[:oid] := bat.getorderidx(t);
 io.print(oidx);
#--------------------------#
# h t  # name
# void  oid  # type
#--------------------------#
[ 0@0,  5@0  ]
[ 1@0,  1@0  ]
[ 2@0,  0@0  ]
[ 3@0,  6@0  ]
[ 4@0,  2@0  ]
[ 5@0,  4@0  ]
[ 6@0,  3@0  ]

 # Q: which BAT properties does oidx have?  [ key(oidx), nonil(oidx) ]

 sorted :bat[:int] := algebra.projection(oidx,t);
 io.print(sorted);
```

```
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  38  ]
[ 1@0,  39  ]
[ 2@0,  40  ]
[ 3@0,  41  ]
[ 4@0,  42  ]
[ 5@0,  43  ]
[ 6@0,  44  ]
```

Creating Order Indexes (On the Fly)

MonetDB may *automatically* create a temporary order index to support predicates $lo \leqslant a \leqslant hi$ or other order-sensitive queries (e.g., ORDER BY, GROUP BY).

- Check current properties of column BATs and presence of indexes in MonetDB system table sys.storage:

```
sql> SELECT column, sorted, revsorted, "unique", orderidx
     FROM   sys.storage('sys', 'indexed');
+----------------+--------+-----------+--------+----------+
| column         | sorted | revsorted | unique | orderidx |
+================+========+===========+========+==========+
| a              | true   | null      | true   |        0 |
| b              | null   | null      | null   |        0 |
| c              | false  | false     | null   |        0 |
+----------------+--------+-----------+--------+----------+
```

Demonstrate on-the-fly creation of an order index in column c of table indexed.

```
$ mclient -d scratch -l sql

-- ❶ check current state of column BATs of table indexed

 SELECT column, type, mode, count, columnsize, hashes, imprints, sorted, revsorted, "unique", orderidx
 FROM  sys.storage('sys', 'indexed');

+----------------+---------+----------+---------+------------+--------+----------+--------+-----------+--------+----------+
| column         | type    | mode     | count   | columnsize | hashes | imprints | sorted | revsorted | unique | orderidx |
+================+=========+==========+=========+============+========+==========+========+===========+========+==========+
| a              | int     | writable | 1000000 |    4000000 |      0 |        0 | true   | null      | true   |        0 |
| b              | clob    | writable | 1000000 |    4000000 |      0 |        0 | null   | null      | null   |        0 |
| c              | decimal | writable | 1000000 |    2000000 |      0 |        0 | false  | false     | null   |        0 |
| indexed_a_pkey | oid     | writable |       0 |          0 |      0 |        0 | true   | true      | true   |        0 |
+----------------+---------+----------+----▲----+-------▲----+------▲-+--------▲-+----▲---+---▲-------+--▲-----+--------▲-+
                                      cardinality   card ×       there are other      column/BAT properties       no order
                                                  type width     kinds of indexes       (null ≡ unknown)       index defined


-- ❷ Perform ORDER BY on column c which can benefit from an order index

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

+------+----------------------------------+-------+
| a    | b                                | c     |
+======+==================================+=======+
|  231 | 9b04d152845ec0a378394003c96da594 | -1.00 |
|   42 | a1d0c6e83f027327d8461063f4ac58a6 | -1.00 |
|   55 | b53b3a3d6ab90ce0268229151c9bde11 | -1.00 |
|   99 | ac627ab1ccbdb62ec96e702f07f6425b | -1.00 |
|  124 | c8ffe9a587b126f152ed3d89a146b445 | -1.00 |
[...]


-- ❸ Re-check state of column BATs

 SELECT column, type, mode, count, columnsize, hashes, imprints, sorted, revsorted, "unique", orderidx
 FROM  sys.storage('sys', 'indexed');

+----------------+---------+----------+---------+------------+----------+--------+----------+--------+-----------+--------+----------+
| column         | type    | mode     | count   | columnsize | location | hashes | imprints | sorted | revsorted | unique | orderidx |
+================+=========+==========+=========+============+==========+========+==========+========+===========+========+==========+
| a              | int     | writable | 1000000 |    4000000 | 21/2155  |      0 |        0 | true   | null      | true   |        0 |
| b               | clob    | writable | 1000000 |    4000000 | 05/531    |      0 |        0 | null   | null      | null   |        0 |
| c               | decimal | writable | 1000000 |    2000000 | 25/2577 ◀ |      0 |        0 | false  | false     | null   |  8000024 |
| indexed_a_pkey  | oid     | writable |       0 |          0 | 11/1105 : |      0 |        0 | true   | true      | true   |      ▲ 0 |
+-----------------+---------+----------+---------+------------+---------:+--------+----------+--------+-----------+--------+-----:----+
                                                                 on-disk location                              order index created
                                                               of BATs for column c                  as a by-product of query evluation

-- ❹ Re-check state of column BATs

 (shell) $ ls -l MonetDB/data/scratch/bat/25
-rw-r--r--  1 grust  staff  2031616 May  8 14:05 2577.tail
-rw-r--r--  1 grust  staff  4325416 May  8 14:06 2577.thash
-rw-r--r--  1 grust  staff   393216 May  8 14:06 2577.timprints
-rw-r--r--  1 grust  staff  8060928 May  8 14:09 2577.torderidx  ◀


-- ❺ Order indexes are static, non-updatable index structures.  Updates on column c
--   invalidate the index ⇒ remove it

 UPDATE indexed
 SET    c = -1
 WHERE  a = 42;

 SELECT column, type, mode, count, columnsize, hashes, imprints, sorted, revsorted, "unique", orderidx
 FROM  sys.storage('sys', 'indexed');

+-----------------+---------+----------+---------+------------+----------+--------+----------+--------+-----------+--------+----------+
| column          | type    | mode     | count   | columnsize | location | hashes | imprints | sorted | revsorted | unique | orderidx |
+=================+=========+==========+=========+============+==========+========+==========+========+===========+========+==========+
| a               | int     | writable | 1000000 |    4000000 | 21/2155  |      0 |        0 | true   | null      | true   |        0 |
| b               | clob    | writable | 1000000 |    4000000 | 05/531   |      0 |        0 | null   | null      | null   |        0 |
| c               | decimal | writable | 1000000 |    2000000 | 25/2577  |      0 |        0 | false  | false     | null   |      ▶ 0 |
| indexed_a_pkey  | oid     | writable |       0 |          0 | 11/1105  |      0 |        0 | true   | true      | true   |      : 0 |
+-----------------+---------+----------+---------+------------+----------+--------+----------+--------+-----------+--------+-----:----+

 (shell) $ ls -l MonetDB/data/scratch/bat/25
-rw-r--r--  1 grust  staff  2031616 May  8 14:05 2577.tail
                                                                                                                      ordex index gone
```

Creating Order Indexes (Manually)

If this seems beneficial for the **query workload**, clients may *manually* create an order index.

- ⚠ Order indexes are **static** (i.e., not maintained under updates—costly) ⇒ underlying table must be *read-only*:

  ```
  ‹create and populate table T›
  sql> ALTER TABLE T SET READ ONLY;
  sql> CREATE ORDERED INDEX I ON T(a);
  ```

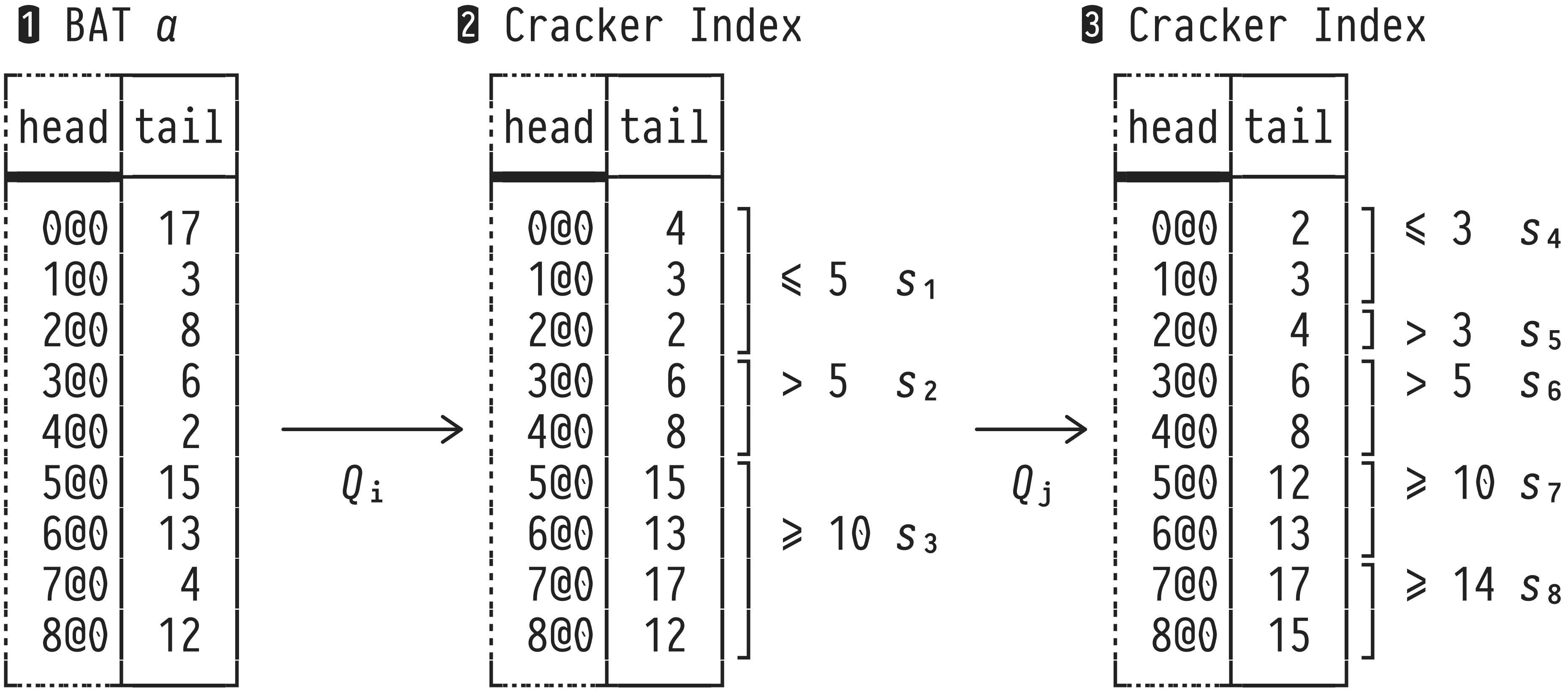
  - Order index *I* is made persistent (in a `*.torderidx` disk file) and will be used by future `algebra.select()`s on column *a*.

With **column cracking,**[13] MonetDB introduced a **self-organizing** (partially) ordered index structure.

- A **cracker index** for column a is created/updated as a **by-product of processing range predicates** $lo \leqslant a \leqslant hi$.
    - In the cracker index, the a values $\in [lo,hi]$ are stored physically contiguous.
- If the **query workload** focuses only on a subset of column a, that part is indexed with fine granularity (while the other parts remain largely non-indexed).

[13] *“Database Cracking”*, S. Idreos, M. Kersten, S. Manegold. Proc. CIDR, Asilomar (CA, USA), 2007.

Column Cracking As a By-Product of Query Processing



❶ BAT a

head	tail
0@0	17
1@0	3
2@0	8
3@0	6
4@0	2
5@0	15
6@0	13
7@0	4
8@0	12

$\xrightarrow{Q_i}$

❷ Cracker Index

head	tail		
0@0	4	$\leqslant 5$	s_1
1@0	3		
2@0	2		
3@0	6	> 5	s_2
4@0	8		
5@0	15	$\geqslant 10$	s_3
6@0	13		
7@0	17		
8@0	12		

$\xrightarrow{Q_j}$

❸ Cracker Index

head	tail		
0@0	2	$\leqslant 3$	s_4
1@0	3		
2@0	4	> 3	s_5
3@0	6	> 5	s_6
4@0	8		
5@0	12	$\geqslant 10$	s_7
6@0	13		
7@0	17	$\geqslant 14$	s_8
8@0	15		

- Q_i: … **WHERE** $a > 5$ **AND** $a < 10$ Result: slice s_2
- Q_j: … **WHERE** $a > 3$ **AND** $a < 14$ Result: slices $s_5 + s_6 + s_7$

Column Cracking Notes

- MonetDB implements slicing in terms of *views*[14] of the source BAT, no data copying involved. Cost free.

- $\forall x \in s_i,\ y \in s_{i+1}: x < y$: a fully cracked column ($\forall_i |s_i| = 1$) is completely ordered. This is uncommon (workload skew).

- First cracking step (❶→❷) copies source BAT. All further steps physically reorganize the cracker index.

- Physical cracker index reorganization (“tail shuffling”) can be efficiently performed *in-situ*.

[14] A possible BAT view: (*source BAT*, *first row*, *last row*).

Cracker Index Reorganization For Predicate $a < hi$

Reorganize column vector $a[\,]$ between row offsets $start$ and end, relocate its elements *in-situ*:

```
CrackInTwo($a$,$start$,$end$,$hi$):
 while ($start < end$)
   if ($a[start] < hi$)
     $start \leftarrow start + 1$;
   else
     while ($a[end] \geqslant hi \wedge end > start$)
       $end \leftarrow end - 1$;
     swap($a[start]$, $a[end]$);  ⁂
     $start \leftarrow start + 1$;
     $end \leftarrow end - 1$;
```

- ⁂ Either $a[start] \geqslant hi \wedge a[end] < hi$ or $start = end$.

At **swap()**, either

1. $a[start] \geqslant hi \wedge a[end] < hi$ or
2. $start = end$

In these cases, element swapping is either required (1.) or harmless (2.).