Design and Implementation of DuckDB Internals

The ART of Indexing

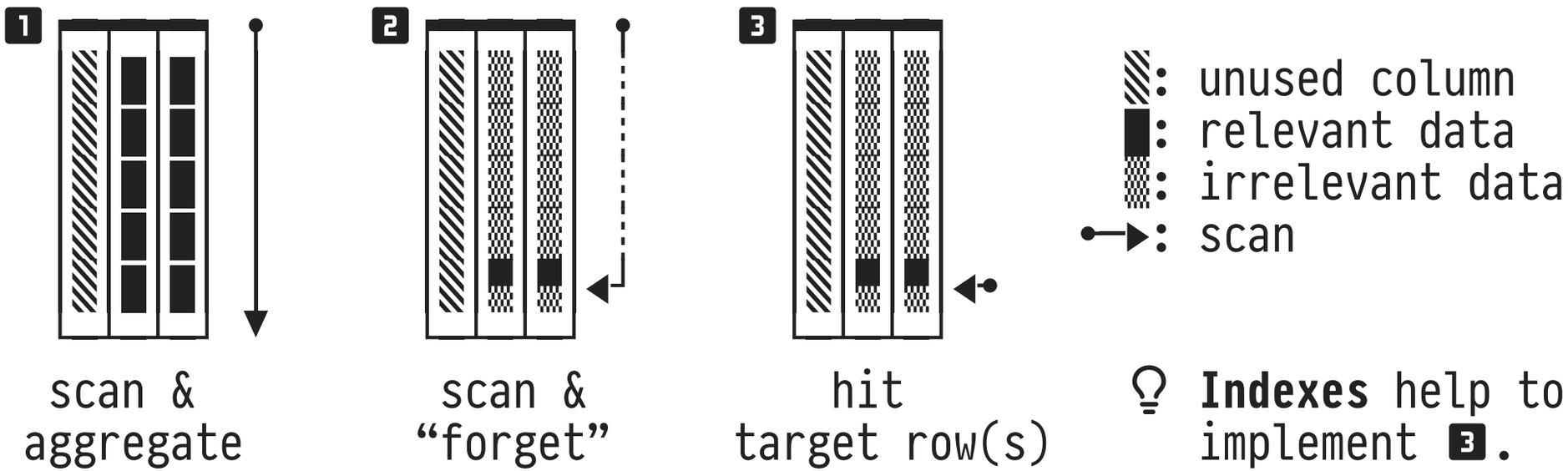
April 7, 2026

Torsten Grust
Universität Tübingen, Germany

1 Scanning All Rows vs. Narrowing In On Few Rows

The internals of DuckDB have been designed to efficiently support analytical SQL queries: “***o**nline **a**nalytical **p**rocessing*” (**OLAP**).

- Typically, these queries read (and aggregate) *all rows* of the input tables 1.
- Yet, scanning all rows wastes I/O and memory bandwidth 2 if a query focuses on *small row subsets (or even a single row)*:



Indexes in DuckDB

DuckDB implements two kinds of **indexes:**

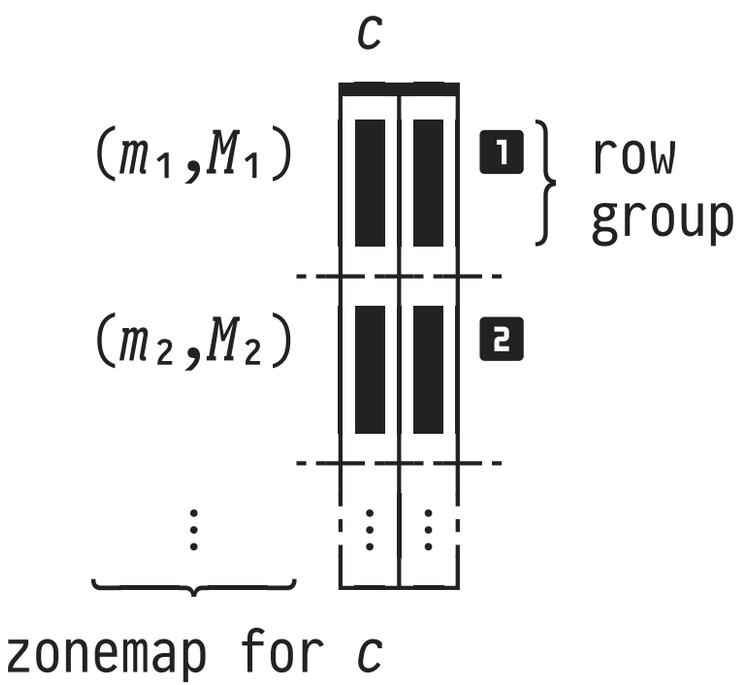
1. **Zonemaps** (also: *min–max indexes*) are integral part of the columnar table storage 🏛.
   - Automatically created, always present for all columns.
   - Used when predicates are pushed down into Sequential Scan.

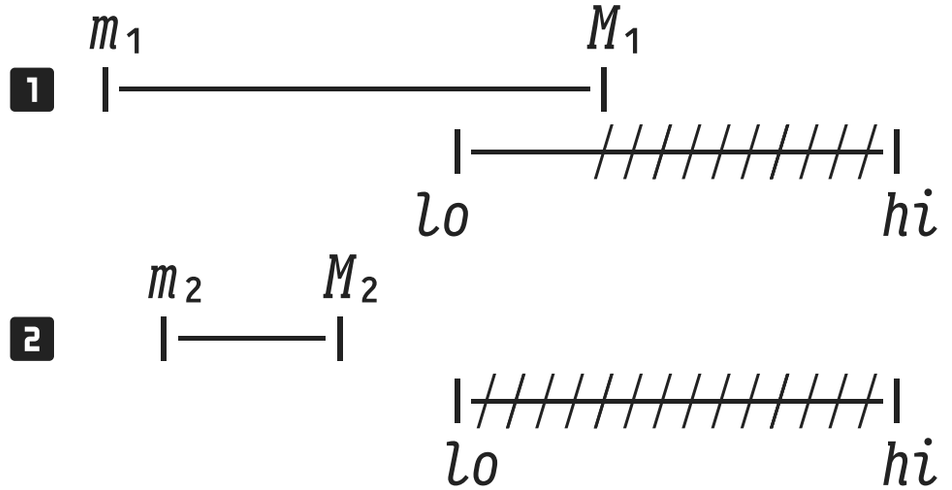
2. **Adaptive radix trees** (**ART**) are tree-shaped data structures maintained outside/in addition to table storage 🏛+🌳.
   - Consume working memory space.
   - Require maintenance on table updates.
   - Created either
     - manually via SQL's DDL statement CREATE INDEX or
     - automatically when constraints UNIQUE, PRIMARY KEY, and FOREIGN KEY are declared for a table.

2 Zonemaps

Zonemaps are baked into DuckDB's columnar table storage format:

- Each column *c* is divided into *row groups* of 120K (122880) rows.
- Each row group holds a (*min*,*Max*) entry that—quite roughly—characterizes the contained values.
- Operator Sequential Scan can safely **skip row groups** for which a predicate $lo \leqslant c \leqslant hi$ will always fail (///):

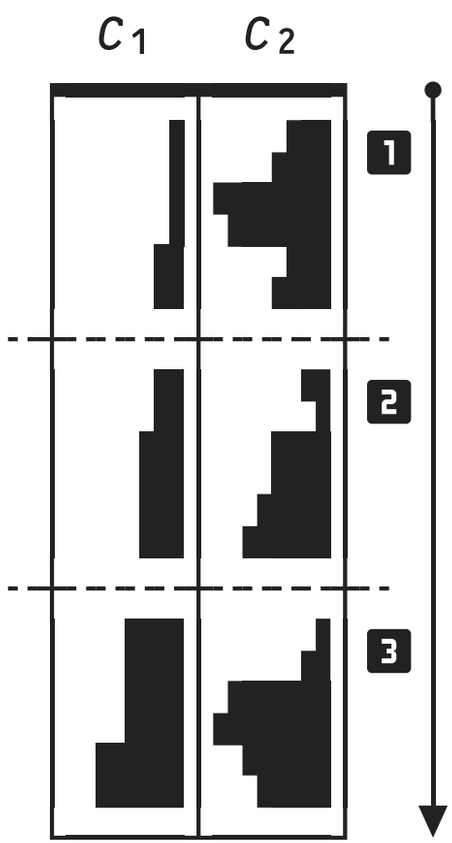




- Need to scan row group 1
- May skip row group 2

Zonemaps and Column Ordering

Column ordering affects the effectiveness of zonemaps. #015



c_1: In each row group, the zonemap entries (*min*,*Max*) for c_1 have a small *span* |—|.

⇒ A large number of row groups will never satisfy $lo \leqslant c_1 \leqslant hi$. Can skip. 👍

c_2: All spans in unordered column c_2 are wide |————————| and cover the active domain of c_2.

⇒ Most (all) row groups will potentially contain hits for $lo \leqslant c_2 \leqslant hi$.
No skipping. 👎

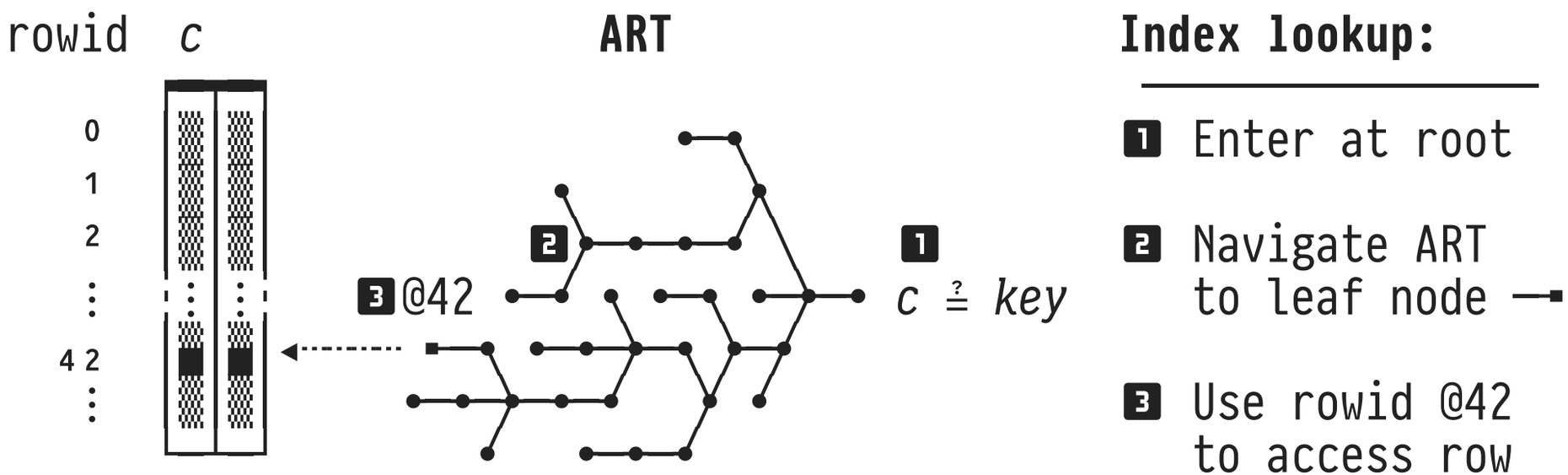
💡 Keeping base tables ordered may help predicate evaluation.

3 ¦ Adaptive Radix Tree (ART) Indexes

ART indexes are **ordered search tree structures** built to evaluate simple **equality and range predicates** (assume an ART on column *c*):

$c = key$ ¦ c IN $(key_1, key_2, \ldots)$ ¦ $c < key$ ¦ $c > key$

- ARTs and tables are separate. These extra data structures
  - use **row IDs** to refer to rows in their associated table,
  - must be created, then maintained under table updates, and
  - occupy space in working memory.



Creating ART Indexes in DuckDB

1. **Manual:**

```
CREATE [UNIQUE] INDEX index ON table (column [, column, ...]);
DROP INDEX [IF EXISTS] index;
```

   - **NB.** More indexes aid query evaluation but incur maintenance and space[1] overhead. A tradeoff in physical database design.

2. **Implicit** (supports constraint enforcement). DDL statement ❶ creates on a unique ART index on $t(c)$ behind the scenes:

```
❶ CREATE TABLE t (..., c ... PRIMARY KEY, ...);  -- also: UNIQUE
❷ CREATE TABLE s (..., f ... REFERENCES t(c), ...);
```

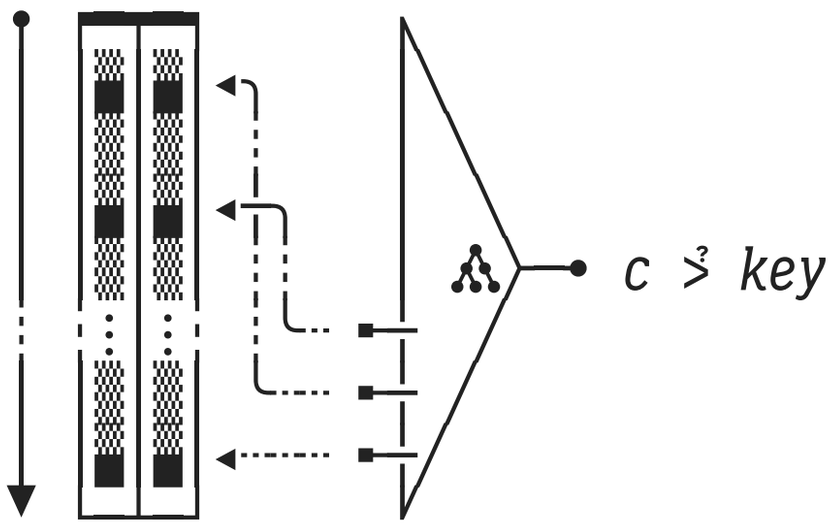
   - DDL statements like `INSERT INTO s VALUES (..., key, ...)` or `UPDATE t SET c = key` lead to lookups $c \stackrel{?}{=} key$ in that index.

[1] As of DuckDB 1.4, memory occupied by ART indexes is under control of the buffer manager but cannot be evicted. Work to rectify this is underway as I type this.

Use the Index? Ignore the Index?

An index lookup predicate p may yield *multiple rows* in table t:

- $p \equiv c \stackrel{?}{=} key$: If c is not unqiue in t: a single leaf node ⟞ holds multiple rowids ⋯▸ ⋯▸ ⋯▸.
- $p \equiv c \stackrel{?}{>} key$: ❶ perform lookup $c \stackrel{?}{=} key$, ❷ access rows using the rowids in all leaves ➟ ➟ ➟ following the found leaf:



- Lookup finds one leaf (or adjacent leaves ➟ ➟ ➟).
- Yet the rowids ⋯▸ may point all over table t. May need to "jump around" to collect rows. Violates memory locality. 👎

#016

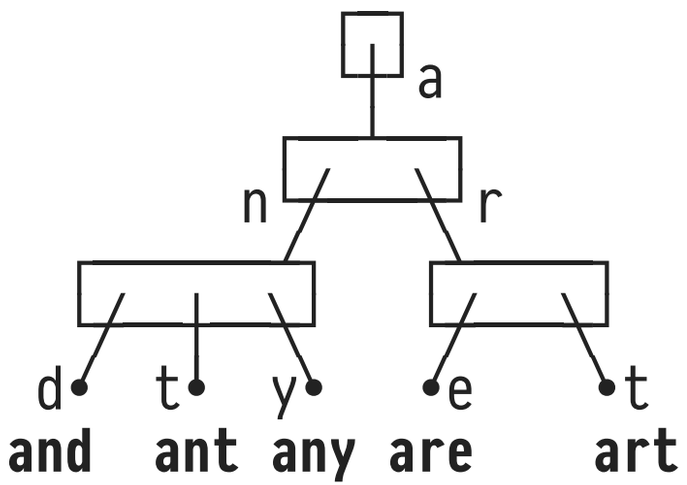
💡 Use index only if **selectivity** $sel(p)$ of predicate p is low:

$$0 \leqslant sel(p) \stackrel{\text{def}}{=} (\textbf{SELECT}\ \text{count}(*)\ \textbf{FILTER}\ (p)\ /\ \text{count}(*)\ \textbf{FROM}\ t) \leqslant 1$$

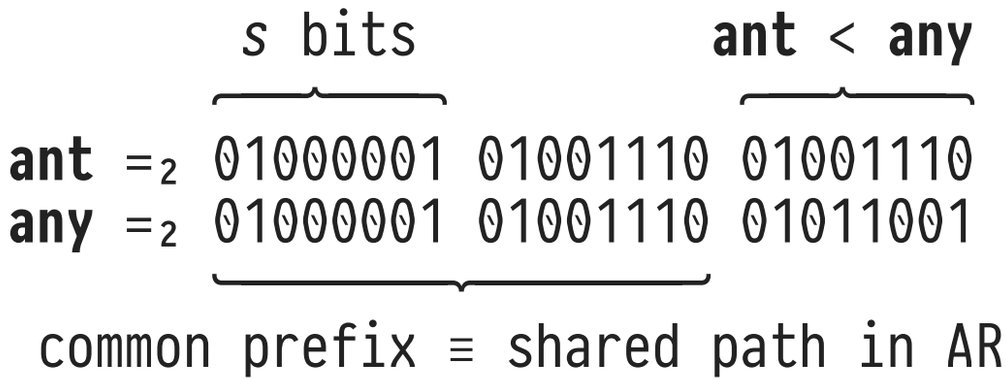
4 The Internals of Adaptive Radix Trees

An ART on $t(c)$ uses the **bit-wise representation of values** in column c to organize itself. Values are *not* hashed or compared.

- Divide values into groups of s bits each (s is the *span*).
- During tree traversal, the next s bits of *key* determine the child node to descend into.



- Example: c :: text, span s = 8 bits.
- Values have k = 24 bits.



ART Internals: Mapping Values to Bit Sequences

Map values to bit sequences whose **lexicographic order** properly reflects value sort order:

- **Unsigned ints:** use standard binary representation (little-endian machines: reverse byte order so that MSB comes first).
- **Signed ints** (in two's complement): flip the sign bit, then treat like unsigned ints.
- **IEEE 754 floats:** 1 always flip the sign bit, 2 if the sign bit was orignally set, now flip all bits. #017
- **Text strings:** map UTF-8 byte sequence to binary[2], end strings with NUL (values must not be prefixes of other values).
- **Composite values** $(v_1,\ldots,v_n)$: map the individual fields v_i, then *concatenate* the resulting bit sequences.
- **NULL:** increase bit length k (*e.g.*, by one bit/one byte) to accomodate the additional value.

[2] DuckDB uses function ucol_getSortKey() of the C/C++ library ICU to perform this mapping.

ART Internals: What is a Good Span?

- The height of ARTs depend on the value bit length k (*not* the number n of entries).
  - An ART for k-bit values has $\lceil k/s \rceil$ levels of inner nodes.
  - ART lookup and insert operations have complexity $O(k)$.

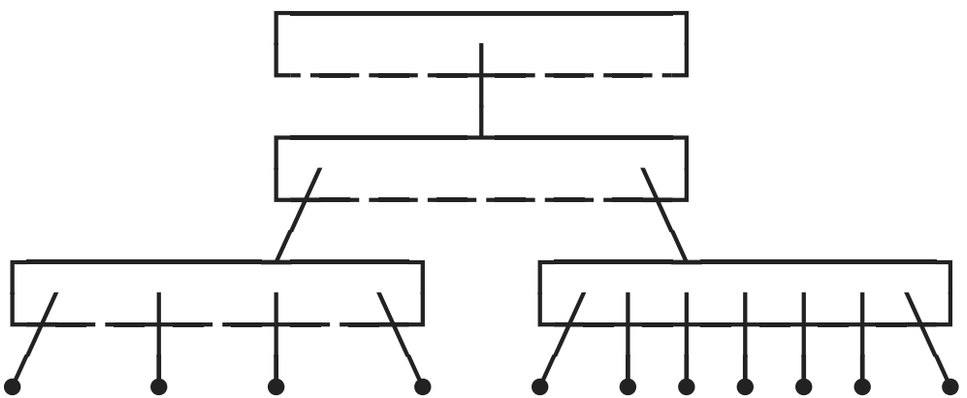
- ARTs *vs.* binary search trees (BSTs):
  - Height: if $n > 2^{k/s}$, ARTs have smaller height than BSTs ($\log_2(n)$).
  - Lookup: for values of k bits (k large), comparison ($<$) is $O(k)$. Complexity of BST lookup thus is $O(k \cdot \log_2(n))$.

💡 Optimize ART **performance:** Work with a large span s! 👍
🚨 But how about the **space usage** of such ARTs? 👎

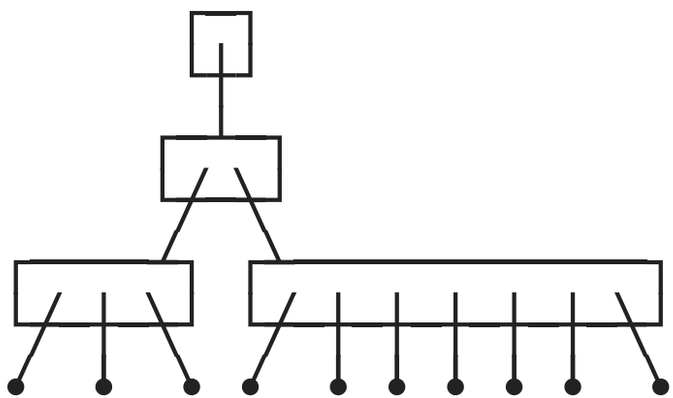
ART Internals: Designing Inner Nodes

- A simple representation of inner ART nodes:
  - Array of 2^s child pointers. (Nodes would have **fixed size.**)
  - Use s-bit chunk of *key* to index array + access child node.
  - But: Most child pointers will be NULL (└——), space usage will be excessive if s is large (grows exponentially):

Node size fixed



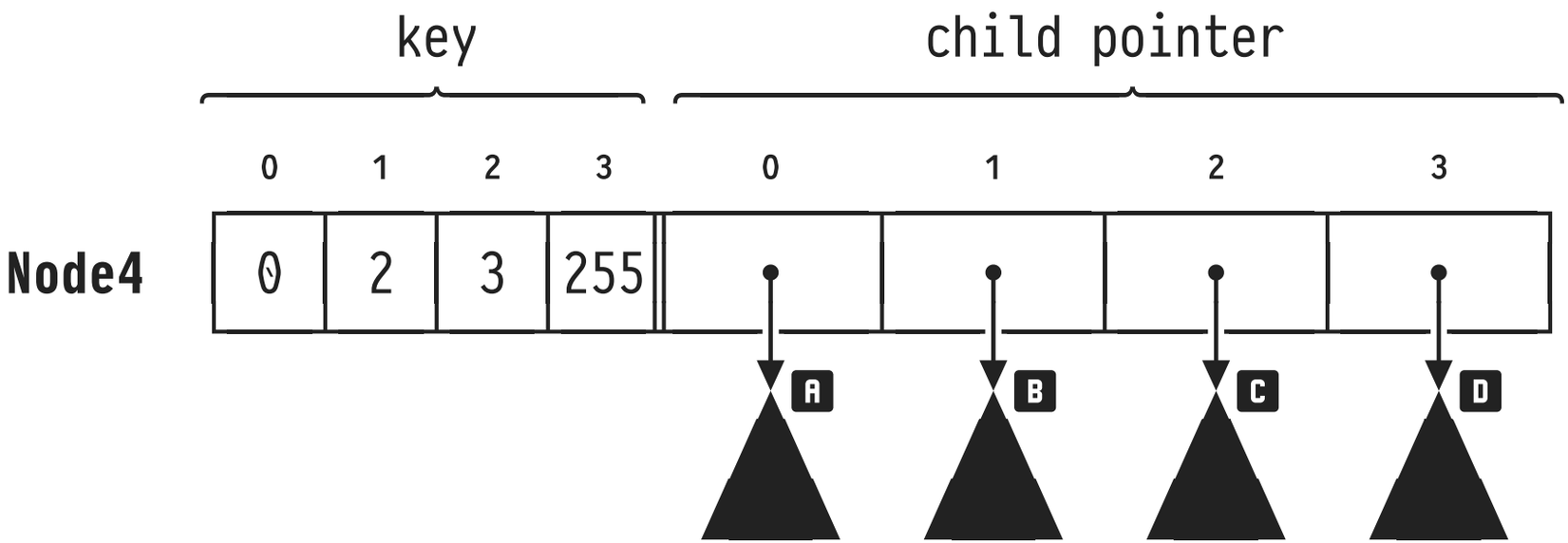
Node size adaptive



💡 Use a fixed large s but **adapt node size** (use variable fanout).

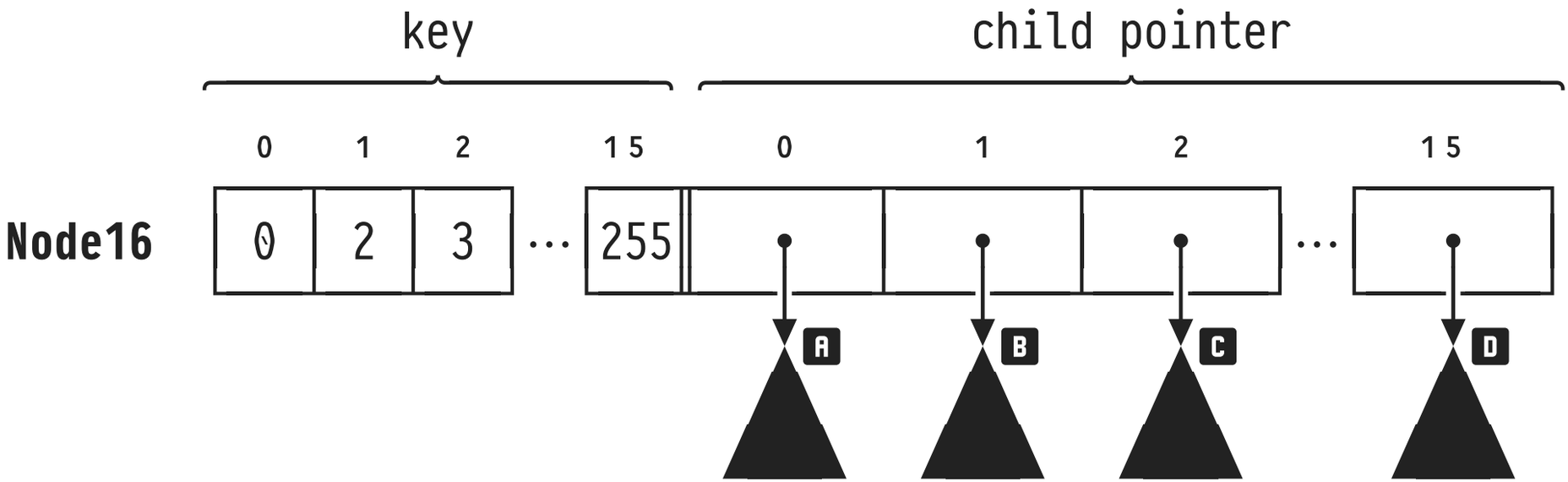
ART Internals: Four Inner Node Types ❶

- DuckDB: s = 8 (can cut values into bytes, admits large fanout).
- Choose inner node type based on the number of non-NULL child pointers: 4, 16, 48, or 256 (= 2^s).
  - On value insertion/deletion, switch to larger/smaller node type when node overflows/underflows.



- **Node4:** Array of 4 keys is ordered (search left to right).
- Value/pointer pairs stored at corresponding indices 0…3.

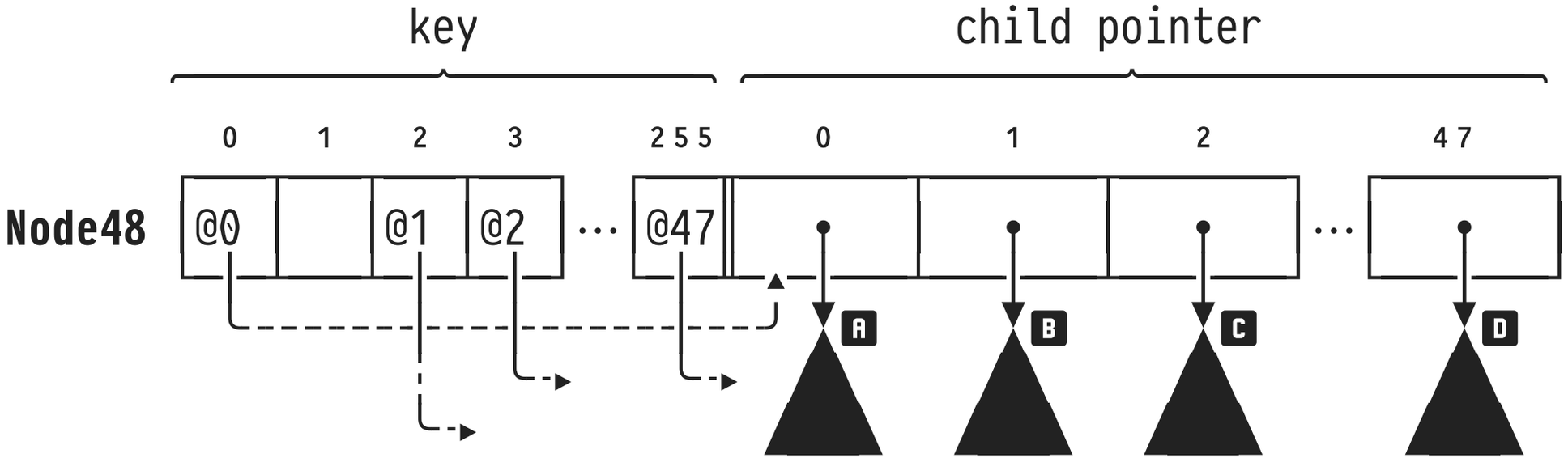
ART Internals: Four Inner Node Types ❷



- **Node16:** stores 5…16 child/pointer pairs.
- Locate current *key* byte by binary search or parallel SIMD comparisons.

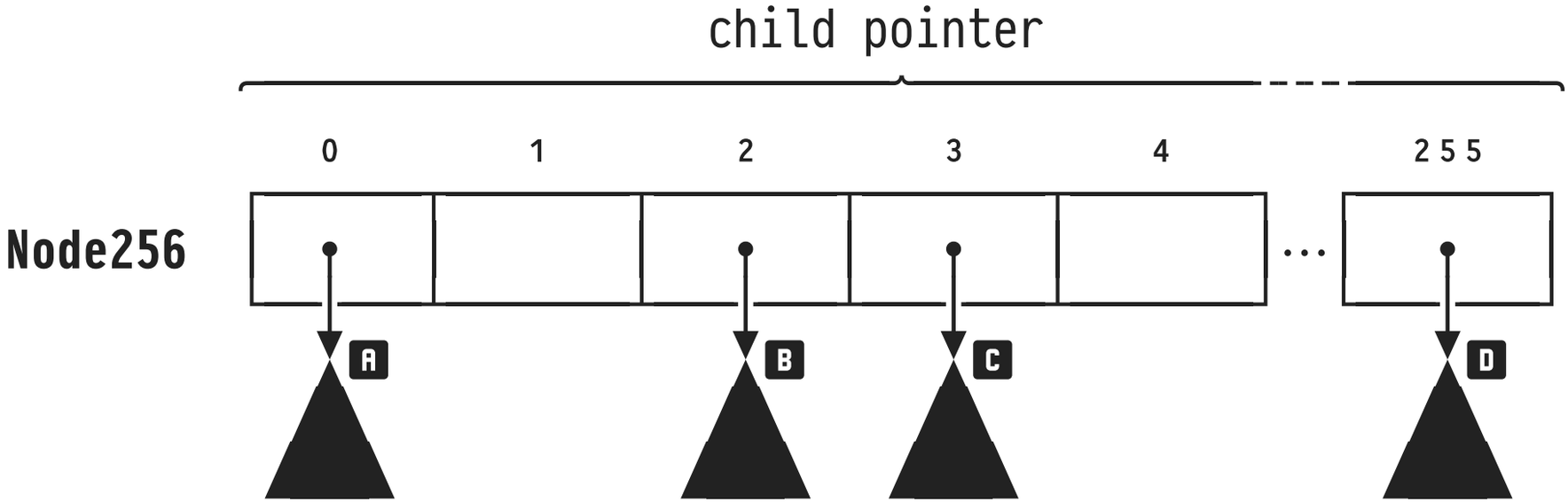
#018

ART Internals: Four Inner Node Types ❸



- **Node48:** With 17…48 values in a node, searching the value array becomes expensive. Thus: do *not* store values. Instead:
  - Use current byte of *key* as index into array of 256 indices.
  - key[‹*key* byte›] holds the index *@i* ∈ {0…47} of the corresponding child pointer. Access pointer[*@i*].
- Saves space compared to an array of 256 8-byte pointers: 640B = 256B + 48 × 8B < 256 × 8B = 2048B.

ART Internals: Four Inner Node Types ❸

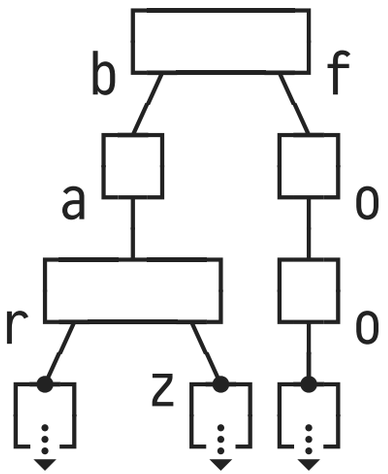


- **Node256:** If an inner node needs to hold 49...256 entries, invest 256 × 8B = 2kB to hold an array of 256 child pointers.
  - Simply return `pointer[<key byte>]`.

ART Internals: Leaf Nodes

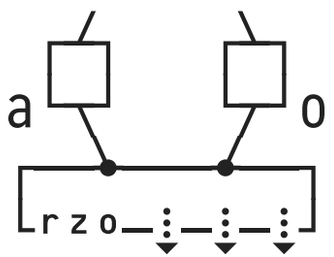
In an ART for *t*(*c*), **leaf nodes** hold rowids that point to the rows in table *t* in which column *c* holds the search value *key*.

- Possible ART leaf designs:



(This is an ART for values **bar**, **baz**, **foo**.)

Single-entry leaves, each hold rowids (⋮) for one value *val*.



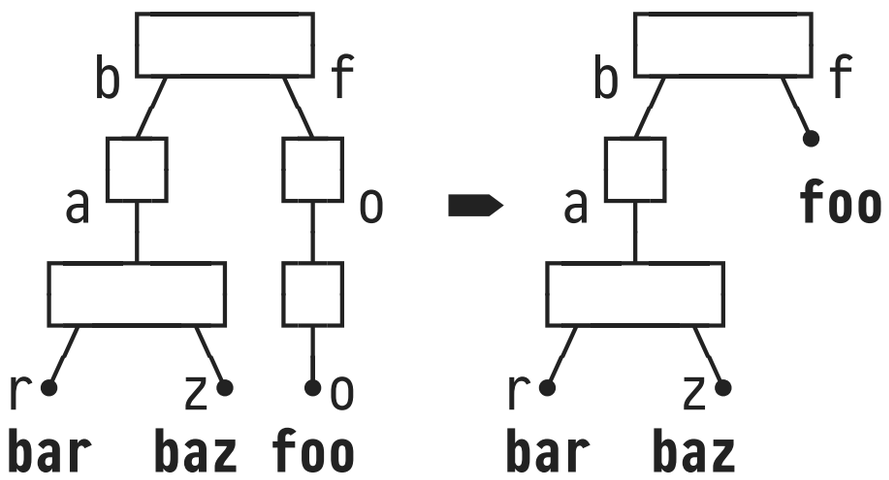
Multi-entry leaf, holds rowids for multiple values. DuckDB has Leaf7/Leaf15/Leaf256 much like Node4...256 (hold rowids, not pointers).

ART Optimizations: Lazy Expansion + Path Compression

To efficiently use space and save tree traversal effort during lookups, try to remove inner ART nodes—and thus **reduce tree height**—if possible.

Most effective if values are long (bit length k is large).

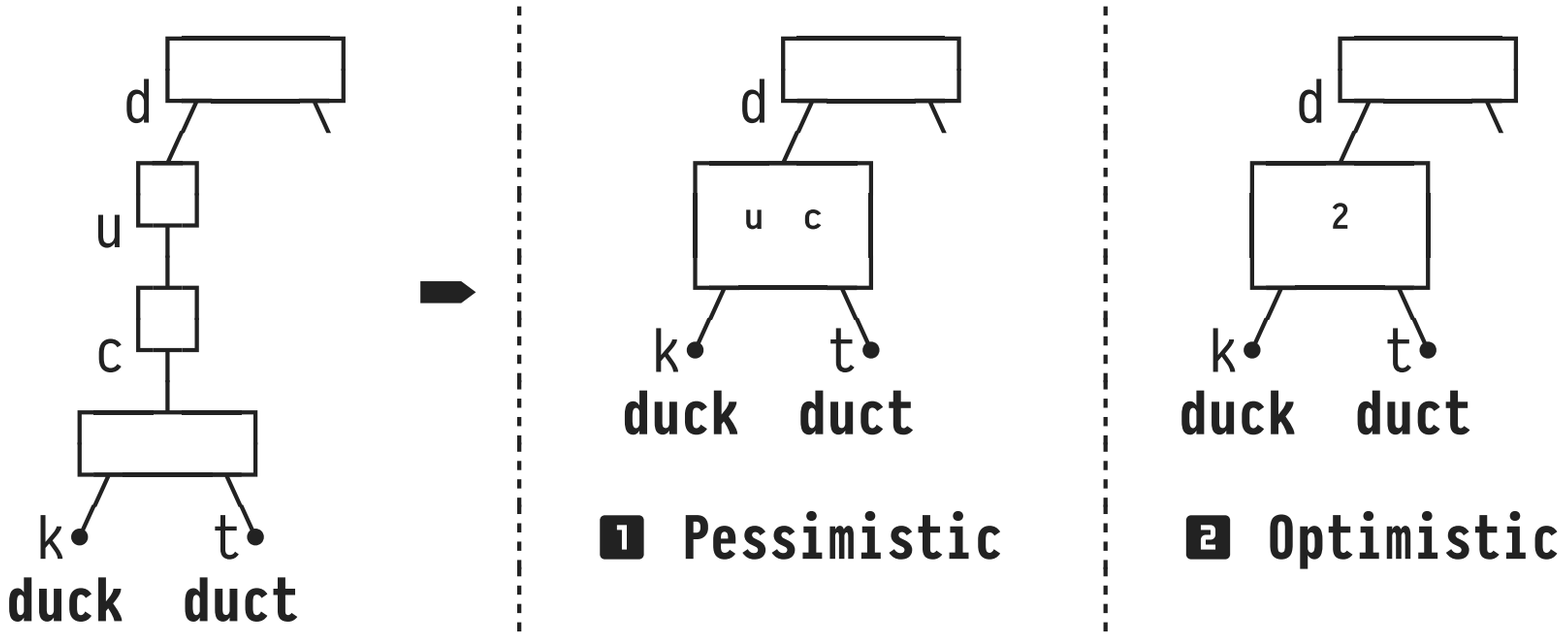
1. **Lazy expansion:** create inner nodes only if they are required to distinguish at least two leaf nodes.



- Saves two inner nodes on the path to leaf **foo**.
- Expand path if another leaf with prefix **f** is added.
- Path does not spell **foo**: store at leaf or in table.

ART Optimizations: Lazy Expansion + Path Compression

2. **Path compression:** remove inner nodes that have a single child.



- 1 Inner node holds byte sequence (here: uc) of preceding single-child nodes. Compare this sequence against *key*.
- 2 Inner holds count (2) of removed single-child nodes. Skip that many bytes in *key*. At a leaf, compare its value to *key* to ensure that traversal indeed reached the proper leaf.

ART: Index Lookup (Lazy Expansion + Pessimistic Path Compression)

```
search(node, key, depth):
 1 if node = NULL                                    | search failed (findChild()
 2 ⌊ return NULL                                     |   returned NULL)
 3 if isLeaf(node)                                   | reached leaf level?
 4 ⌊ if leafMatches(node, key)                       | reached the proper leaf?
 5 ⌊ ⌊ return node                                   |   yes, success
 6 ⌊ return NULL                                     |   no, failure
 7 s ← byteSequence(node)                            | byte sequence in inner node
 8 if s ≠ key[depth...depth+len(s)]                  | does partial key match
 9 ⌊ return NULL                                     |   byte sequence?
10 depth ← depth + len(s)                            | skip to next key byte
11 child ← findChild(node, key[depth])               | key byte selects child
12 return search(child, key, depth+1)                | descend into subtree
```

- Invoke with **search**(*node*: ‹*ART root*›, *key*: 'duck'[0 1 2 3], *depth*: 0).
- Returns leaf node that holds rowid(s) or NULL.

5 ¦ Making the Most of Indexes

DuckDB is an OLAP DBMS, optimized to scan entire tables. Basic index support is present—but there certainly is unused potential.

- The following predicates *could* be evaluated by Index Scan:[3]

Predicate	Index Constellation	Index used by 🦆?
$c_1 \circledless val_1$	$t(c_1)$	👍 (if selective)
	$t(c_1,c_2)$ (composite)	👎
$c_1 \circledless val_1$ **AND** $c_2 \circledless val_2$	$t(c_1,c_2)$ (composite)	👎
	$t(c_1)$ and/or $t(c_2)$	👎
$c_1 \circledless val_1$ **OR** $c_2 \circledless val_2$	$t(c_1)$ and $t(c_2)$	👎
c_1 **LIKE** *'patt_rn%'*	$t(c_1)$	👎

Index support in DuckDB (as of version 1.4)

- (PostgreSQL🐘 is optimized to operate as an OLTP DBMS and can use indexes in all of these scenarios.)

[3] In this table, $\circledless$ represents a SQL comparison operator: <, <=, =, >=, >.

Index Traversals

Index traversals (beyond equality-based index lookups for a key *val*) can support **range predicates:** #019

- c_1 >= val_1 or c_1 >= val_1 **AND** c_1 <= val_2: perform lookup ⟞ for val_1, then scan leaves left to right ⟞⇢ (until > val_2). 1
- c_1 **LIKE** '*prefix%*': perform lookup ⟞ for *prefix*, hit inner node. Scan leaves in subtree ⟞⟞. 2