IN3020&4020 – Database Systems (2024)
Part 1: Query Processing
Lectures 9, 10: Indexes

12, 13 February

Egor V. Kostylev
IFI, University of Oslo
egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

SQL query processing relies on two key ingredients:
Relational Algebra (✓) and Indexes

We study these ingredients first (almost in isolation)
and then put all things together

Materials to read about Indexes

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 17 (and some of 16)
2. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 14 (+ 8.3 & 8.4)
3. IN2090 – Databaser og datamodellering (Leif Harald Karlsen, Dimitru Roman): Lectures 13 (basics)

   `https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/`
4. (English) Wikipedia
5. etc.

You can see the effect of indexes in PostgreSQL

```
CREATE TABLE t1(id int);

INSERT INTO t1
  SELECT n*random()
  FROM generate_series(1, 10000000) AS x(n);

CREATE TABLE t2 AS (SELECT * FROM t1);


CREATE INDEX t1_ind ON t1(id);


\timing on

SELECT * FROM t1 WHERE id = 100; --> Time:  0.792 ms

SELECT * FROM t2 WHERE id = 100; --> Time:  303.022 ms
```

Lecture Plan

1. How to store relations?
2. Types of indexes
3. Multi-level indexes (how to implement indexes)
4. Indexes in practice
5. Multi-dimensional indexes (indexes on several attributes)

1. How to store relations?

Main goal of a DBMS: answer queries quickly

Ingredients for efficiency:

1. Find a good query plan (i.e., RA expression)
2. Store data (i.e., relations) to maximise the speed of retrieval:

   `SELECT * FROM R WHERE R.A = 1345` (exact search)

   `SELECT * FROM R WHERE R.A > 1345` (interval search)

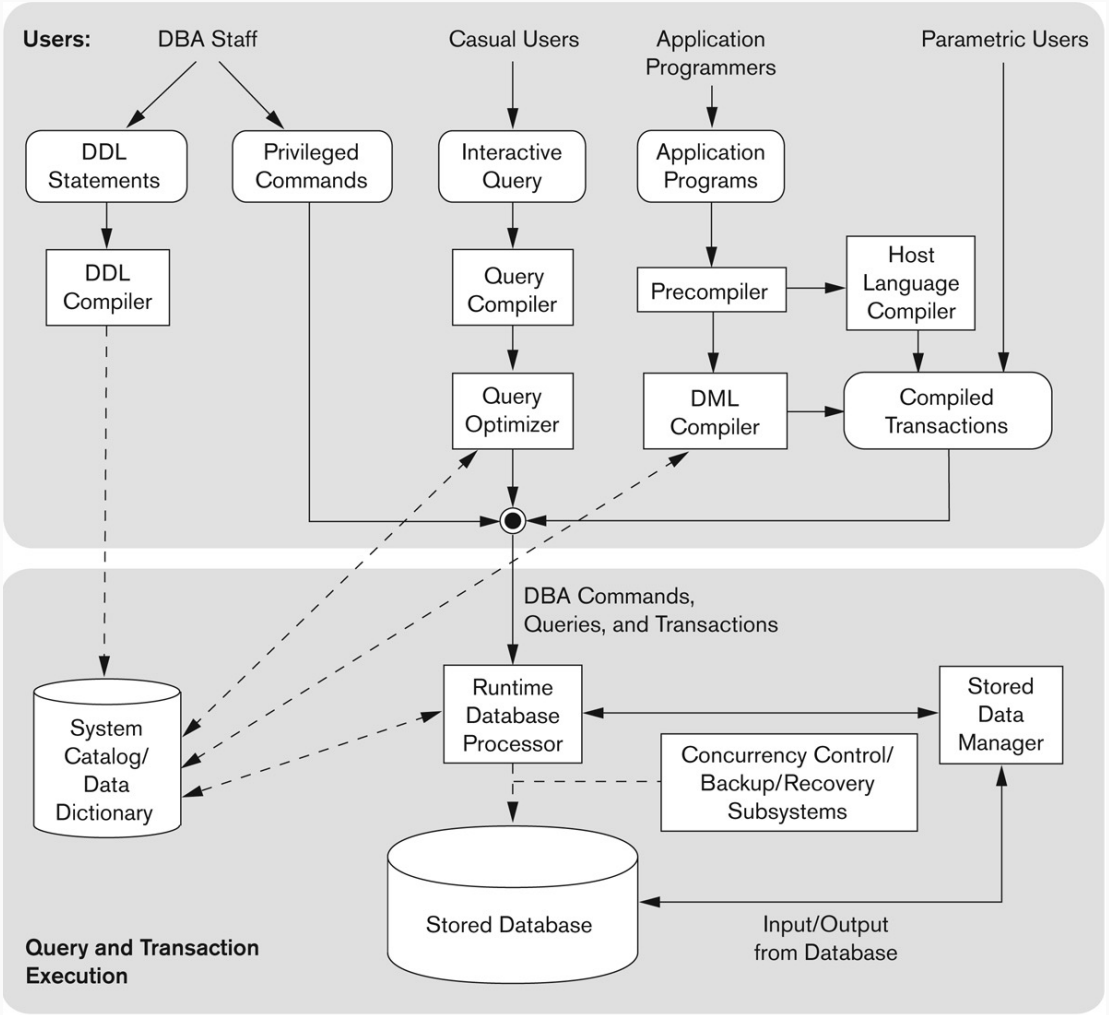
   `SELECT * FROM R WHERE R.A > 1345 AND R.B = 'a'` (multi-attribute search)

Challenge: There may be a lot of data (i.e., tuples)

Updates (incl. insertions and deletions) are less critical, but should also be taken into account

Today: look at how DBMSs manage with Ingredient 2

Place of data storage (incl. indexes) in DBMS architecture



What memory is available

Storage (memory) hierarchy:

- Primary storage:

  CPU main memory, cache memory (Static RAM, DRAM)
- Secondary storage:

  Magnetic disks, flash memory, solid-state drives
- Tertiary storage:

  Removable media, distributed over the Web, etc.

Data in standard DBMSs is stored in secondary memory:

- good size ($\sim$10–100 TBs now)
- slow access ($\sim$10–100 ms now)

Properties of secondary memory

Secondary memory (e.g., magnetic disk):

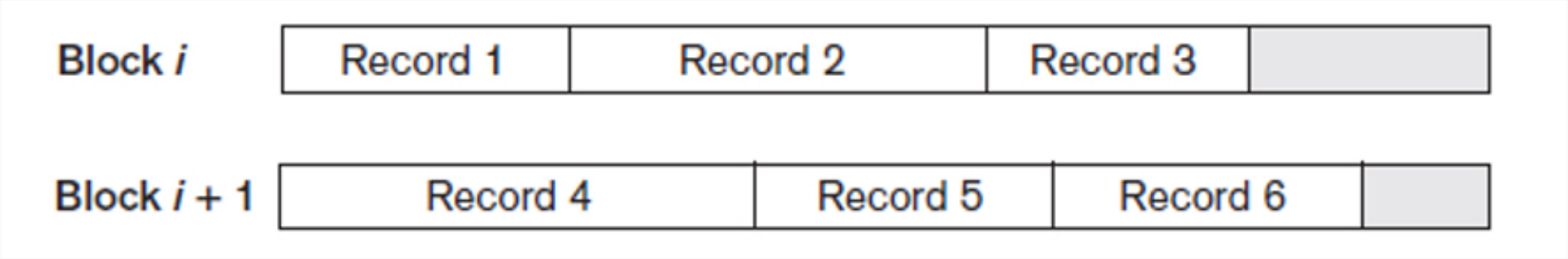
- Organised in blocks (∼10 KBs)
- Can be accessed using pointers,
  which may refer to any point (word) in any block
- However, read & write is possible only for the whole block
- Blocks are consecutive, e.g., we can retrieve
  "the block in the middle between two given blocks"
  (referring to blocks by their first pointers)

Storing a relation

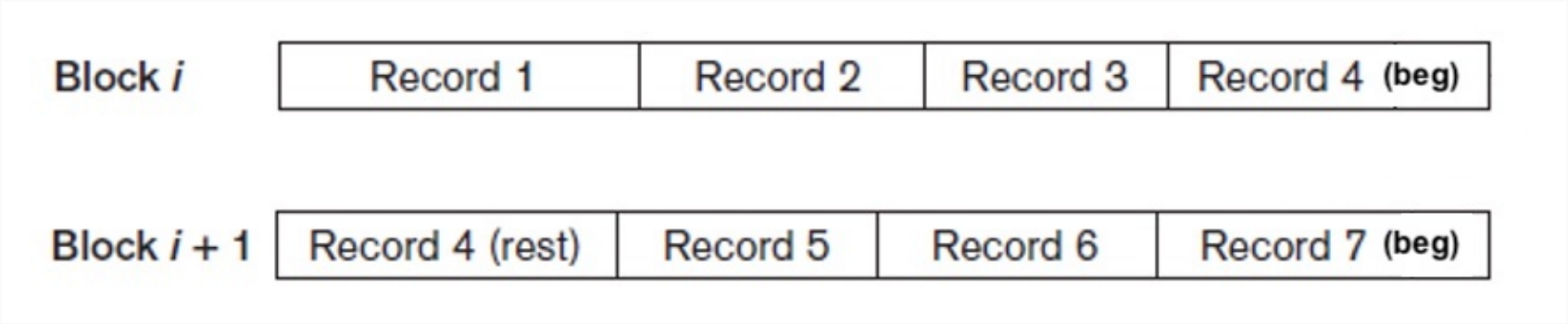
Usually, a block can accommodate several records (i.e., tuples)

There are options to store tuples of a relation in a file:

Continuous, unspanned:



Continuous, spanned:

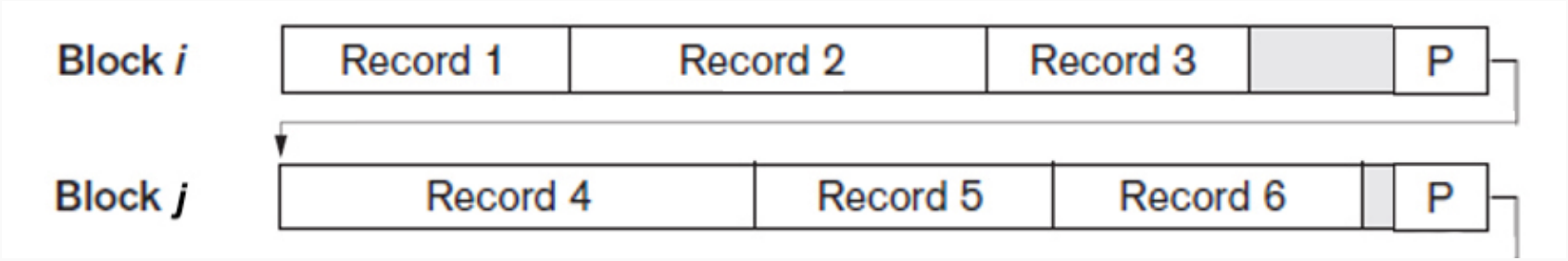


Storing a relation

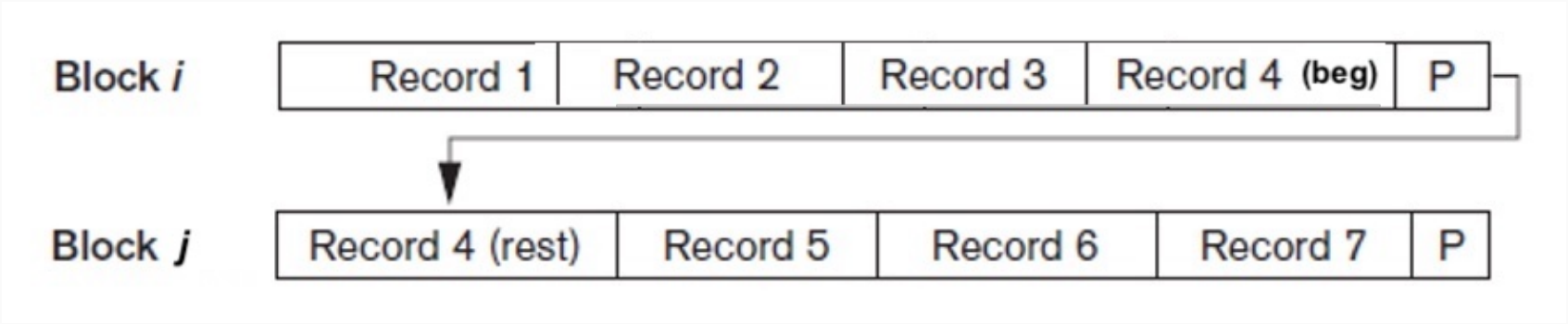
Usually, a block can accommodate several records (i.e., tuples)

There are options to store tuples of a relation in a file:

Linked, unspanned:



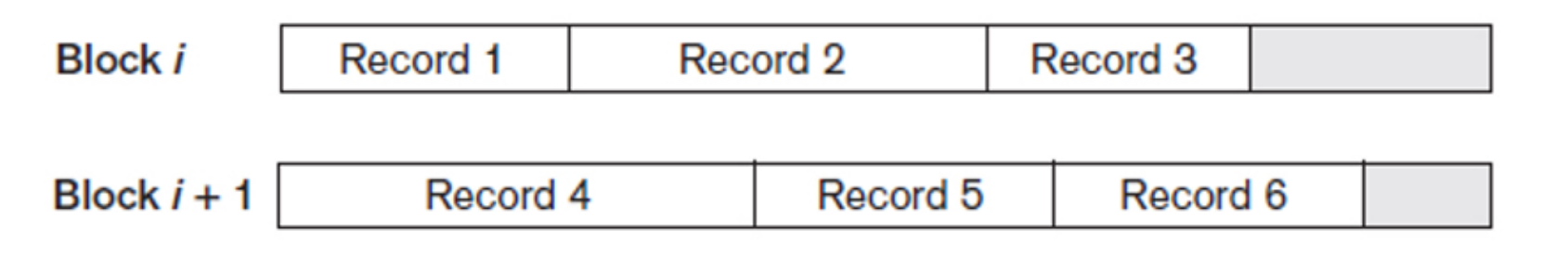
Linked, spanned:



Our assumptions

We assume:

- Generally, continuous, unspanned:



- (Sometimes, linked as well)
- all records (i.e., tuples) take the same size

All these assumptions are for simplicity,

(almost) everything applies to other settings

Naive tuple arrangement: (unordered) heap

Heap (or pile) file:

- tuples placed in file in order of insertion
- inserting a new tuple: very efficient
- deletion of a tuple:
  - real deletion and squeezing all following blocks (slow)
  - real deletion and squeezing only current block (efficient)
  - use deletion marker (efficient)
- searching with a given attribute value (`... WHERE R.A = 1245`)
  - b reads (where b is the number of used blocks)
  - $\sim b/2$ reads on average, if `A` is a candidate key (e.g., primary key)

Slow. What can we do?

Ordered files

File with tuples ordered by the search key `R.A`:

- inserting a new tuple: may be slow, but may use overflow blocks (see below)
- deletion of a tuple: similar to heap
- searching with a given key (where b is the number of used blocks)
  - $\sim b/2$ on average naively (even for non-keys)
  - $\leq \lceil \log_2 b + 1 \rceil$ using binary search

Better, but still... And what to do with several attributes?

Binary search

Let us need to find a tuple with key value k in an ordered relation

- try the middle block; if tuple with k found, success
- if not, identify the half with k above or below the block by checking whether the smallest key value in the block is smaller or greater than k
- try the middle block of the half; if tuple with k found, success
- ...

For example,

- for $b = 7$, a worst case may be retrieving blocks $\#4, \#2, \#3$ (e.g., $\lceil \log_2 b + 1 \rceil = 3$ is materialised)
- for $b = 8$, a worst case may be $\#4, \#6, \#7, \#8$ (e.g., $\lceil \log_2 b + 1 \rceil = 4$ is materialised)

2. Types of Indexes

Indexes Intro

An index on an attribute A (in a relation R):
a data structure that facilitates (i.e., speeds up) finding the tuples in R with a certain value of A

Attribute A is called the search key

It is possible to have an index on every attribute (and every set of), but there is a trade-off:

More indexes for the same relation means

- faster search
- slower updates & more storage

Usually, DBMSs automatically create an index on every primary key

Indexes on other attributes can be created manually

Types of Indexes

We will consider:

Primary Index (dense or sparse)

- the tuples are sorted (physically) on the search key
- at most one tuple for a search-key value (e.g., primary key)

Cluster Index (dense or sparse)

- the tuples are sorted (physically) on the search key
- any number of tuples for a search-key value

Secondary Index (dense only)

- the tuples are not sorted on the search key
- may be used in addition to other indexes on the same relation

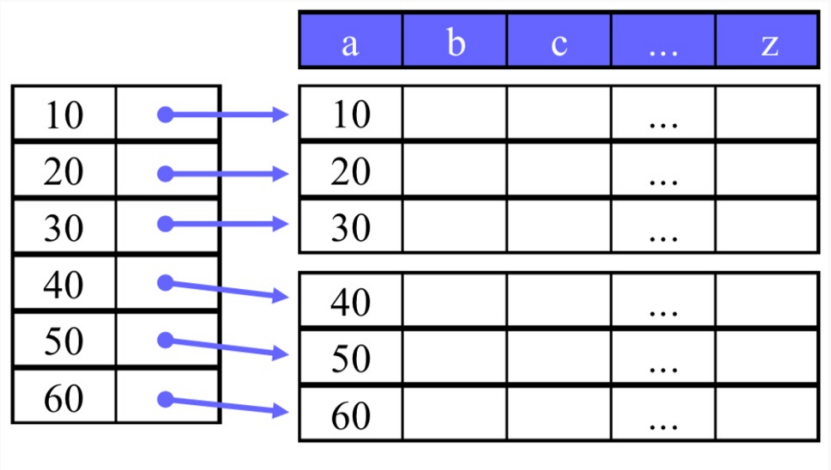
Primary Dense Index

Primary Dense Index on the primary (or candidate) key attribute A in a relation R stored as sorted by A:

- sorted table with two "attributes":

  primary key value, pointer to the tuple with this value
- each tuple in R has a pair in the index

Example:



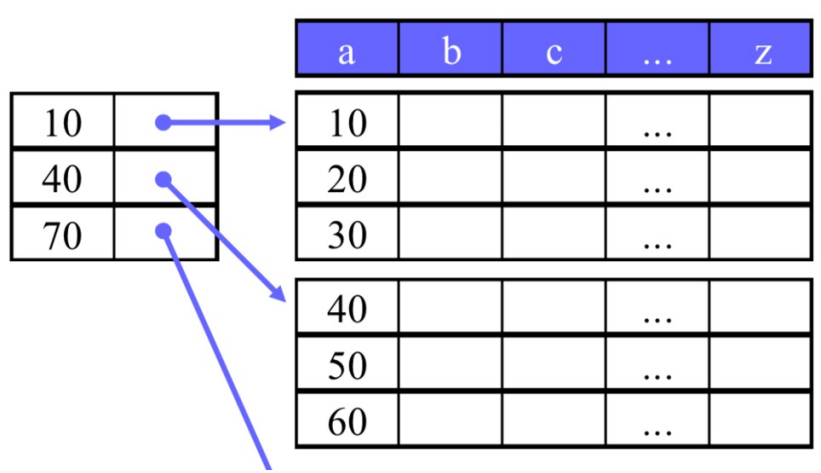
Benefit: index itself is also stored in secondary memory, but with much fewer blocks

Primary Sparse Index

Primary Sparse Index on the primary key attribute A in a relation R stored as sorted by A:

- sorted table with two "attributes":
  primary key value, pointer to the tuple with this value
- only the first tuple in each block of R has a pair in the index

Example:



Benefit: index itself is also stored in secondary memory,
but with even fewer blocks

Retrieval comparison

Assume that we have:

- 1000000 tuples of 300B each, 4B per search key (primary), 4B per pointer
- 4K block size, 1ms for fetching a block:
  - 13 tuples per block (i.e., 76924 unspanned blocks)
  - 512 indexes per block (i.e., 1954 blocks for a dense index and 151 blocks for a sparse index)

Using no index, no order:

$76924/2 = 38462$ block accesses on average (~ 38.5s!)

Using no index, but order (and binary search; think why $+1$ here and below):

$\lceil \log_2(76924 + 1) \rceil = 17$ block accesses at most

Using primary dense index:

$\lceil \log_2(1954 + 1) \rceil + 1 = 11 + 1 = 12$ block accesses at most

Using primary sparse index:

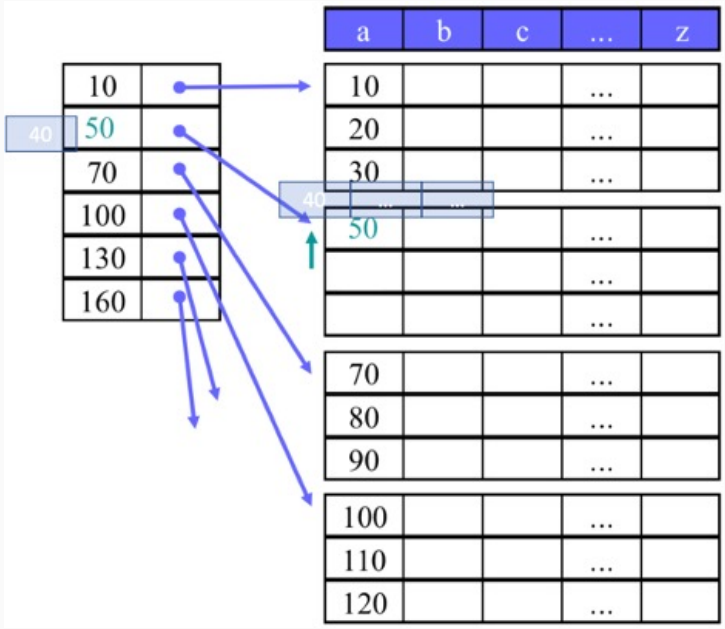
$\lceil \log_2(151 + 1) \rceil + 1 = 8 + 1 = 9$ block accesses at most

Record deletion with primary sparse index

Deletion (and insertion) of tuples needs care

Delete entry with $a = 60$:
index doesn't need an update

Delete entry with $a = 40$:
index needs to be updated
(see to the right)



One usually prefers to compress the data in the blocks

One can also compress the whole file,
but we may like to keep free slots for future inserts

Dense is similar

Record insertion with (sparse) primary index

Insertion (same as deletion) of tuples needs care

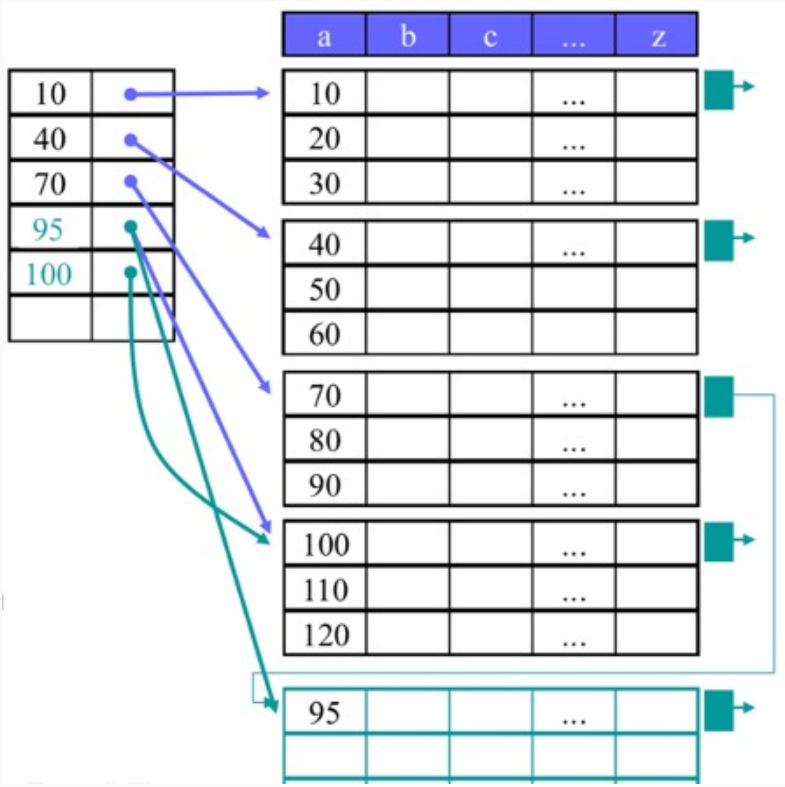
If there is space, similar

Insert entry where a = 95:

No space. Insert a new block

- overflow: change / no change for index
- normal: change to index, breaks continuality

Dense is similar



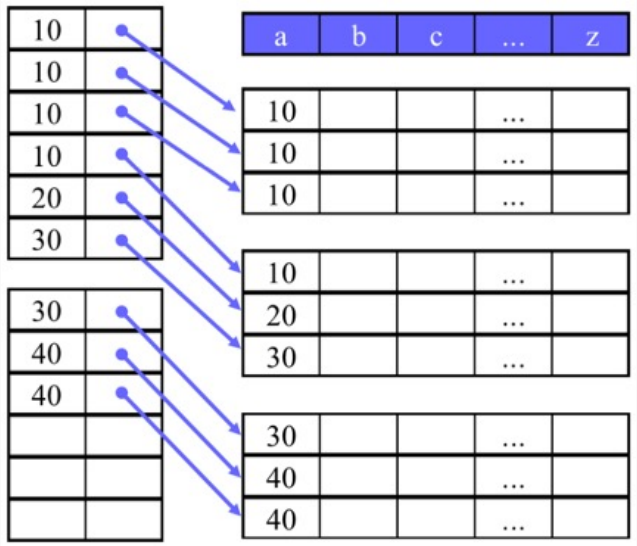
Clustering Dense Index

Clustering Dense Index on an attribute A (any!) in a relation R stored as sorted by A:

- sorted table with two "attributes":

  attribute value, pointer to the tuple with this value
- each tuple in R has a pair in the index

Example:



Not much different from primary dense index

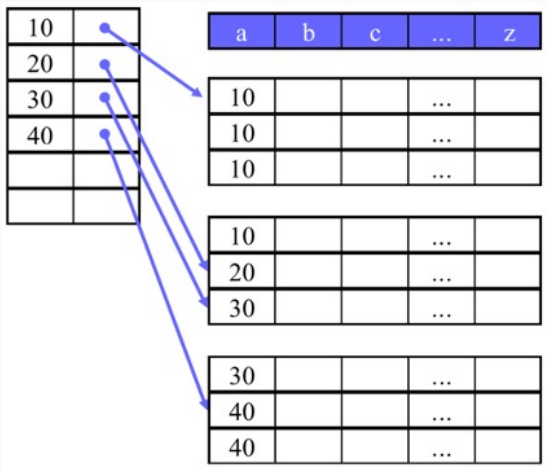
Clustering Sparse Index (variant 1)

Clustering sparse index (var. 1) on an attribute A (any!) in a relation R stored as sorted by A:

- sorted table with two "attributes":

  attribute value, pointer to the tuple with this value
- only one index for each value of A in R has a pair in the index

Example:



Less index and faster search (good for small number of search values)

But finding all tuples for a value is more complex

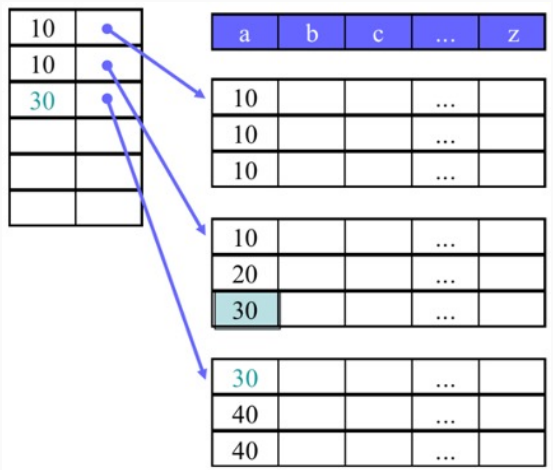
Clustering Sparse Index (variant 2)

Clustering sparse index (var. 2) on an attribute A (any!) in a relation R stored as sorted by A:

- sorted table with two "attributes":

  attribute value, pointer to the tuple with this value
- only the first tuple in each block of R has a pair in the index

Example:



Even less index and faster search (if the number of blocks is less than the number of values)

Finding all tuples for a value is easier

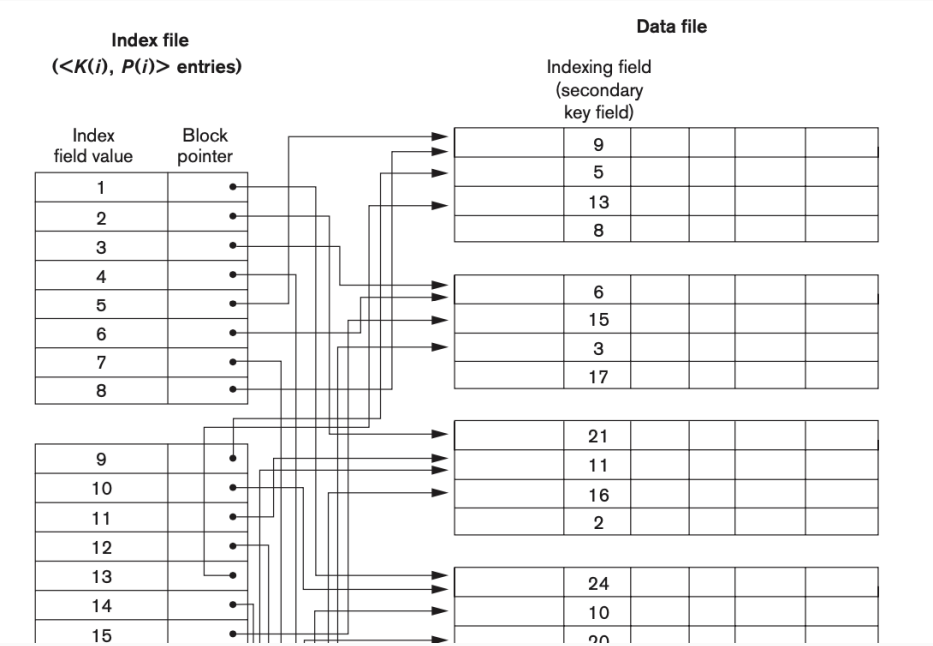
Secondary Index

Secondary index on an attribute A in a relation R (not sorted by A):

- sorted table with two "attributes":

  attribute value, pointer to the tuple with this value

- every tuple in each block of R has a pair in the index



Provide secondary means of accessing tuples

Used when some other primary access exists

Duplicates are allowed (not in the figure, may be optimised by a level of indirection)

Always dense

Dense vs Sparse indexes

	DENSE	SPARSE
Space required	One index field per tuple	One index field per data block
Block access	"Many"	"Few"
Access to entry	Direct access	Must search in the data block
Usage	All cases	Not on unsorted elements
Updates	Always updated if entry sequence is changed	Updated only if the first entry of the data block is changed

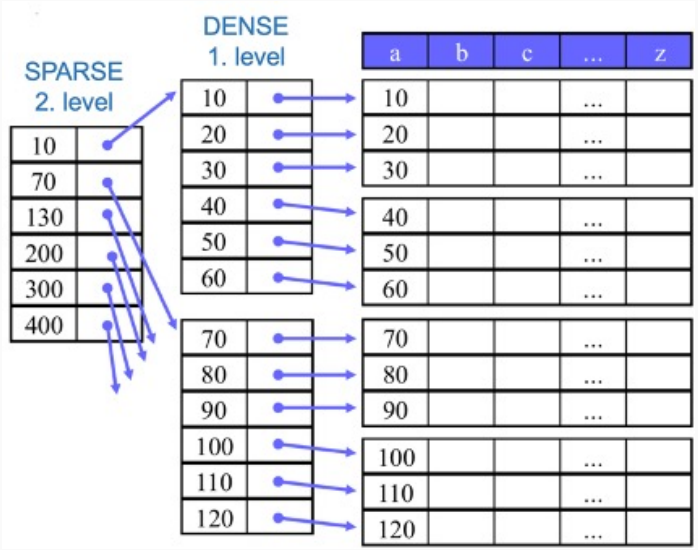
3. Multi-level Indexes

(Static) Multilevel Index

(Static) Multilevel Index on an attribute A in a relation R

- index on an index
- useful when the main index occupy several blocks

Example:



Second layer makes sense only if sparse

In principle, any number of layers is possible

Retrieval comparison

Assume that we have:

- 1000000 tuples of 300B each, 4B per search key, 4B per pointer
- 4K block size, 1ms for fetching a block:
  - 13 tuples per block (i.e., 76924 unspanned blocks)
  - 512 indexes per block (i.e., 1954 blocks for a dense index and 151 blocks for a sparse index)

No index, no order: $76924/2 = 38462$ block accesses

No index, but with order (and binary search): $\lceil \log_2 76925 \rceil = 17$ block accesses

Primary dense index: $\lceil \log_2 1955 \rceil + 1 = 11 + 1 = 12$ block accesses

Using primary sparse index: $\lceil \log_2 152 \rceil + 1 = 8 + 1 = 9$ block accesses

Using two-level index (dense 1st level, sparse 2nd level):
need $1954/512 = 4$ blocks for the 2nd level
$\lceil \log_2(4+1) \rceil + 1 + 1 = 5$ block accesses at most

(Static) Multilevel Indexes

Looks good, but

Problems:

1. Insertion and deletion are messy and difficult
2. How to decide how many levels we need?

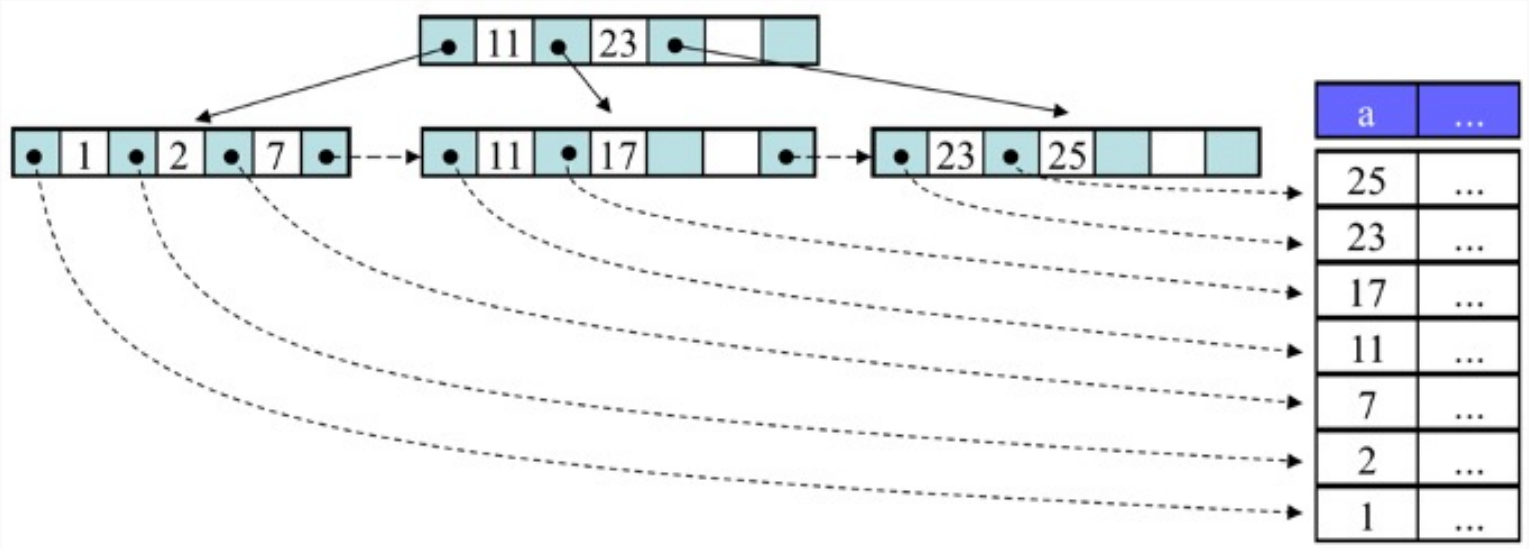
Solution: Dynamic Multilevel Indexes
based on B-trees, B+-trees (B*-trees, etc.)

"B" stays for "balanced"

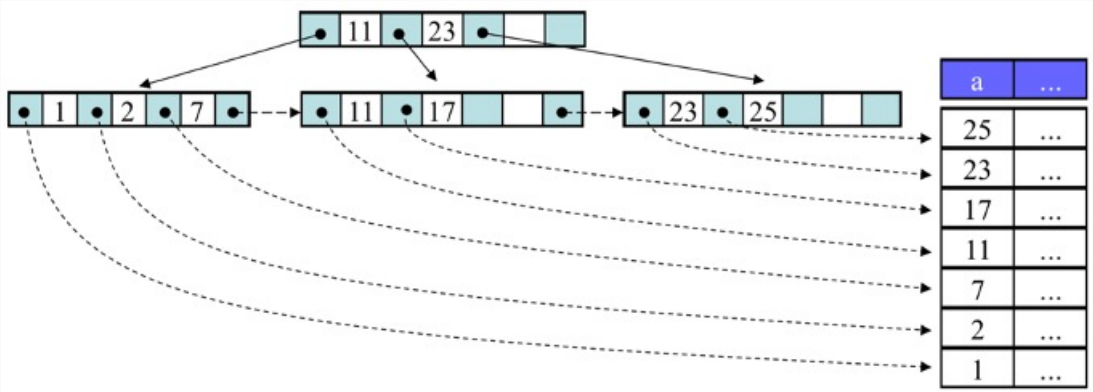
Dynamic Multilevel Indexes: B+-trees

B+-tree of order n:

- Balanced: all leaf nodes are at the same level
- each node has at least $\lceil (n+1)/2 \rceil$ and at most n search keys and $+1$ pointers (except the root)
- Inner node: all pointers are to sub-trees, key values specify values in sub-trees as in static case
- Leaf node: data-pointers (dense or sparse) and 1 next-pointer



Dynamic Multilevel Indexes: B+-trees



Search with B+-trees:

- from the root to leaves—that is, the number of block accesses is the height (usually, low; e.g., for $n = 340$ and 3 levels we can refer to $>$16M tuples)
- interval search is also fast

Insertion & deletion with B+-trees:

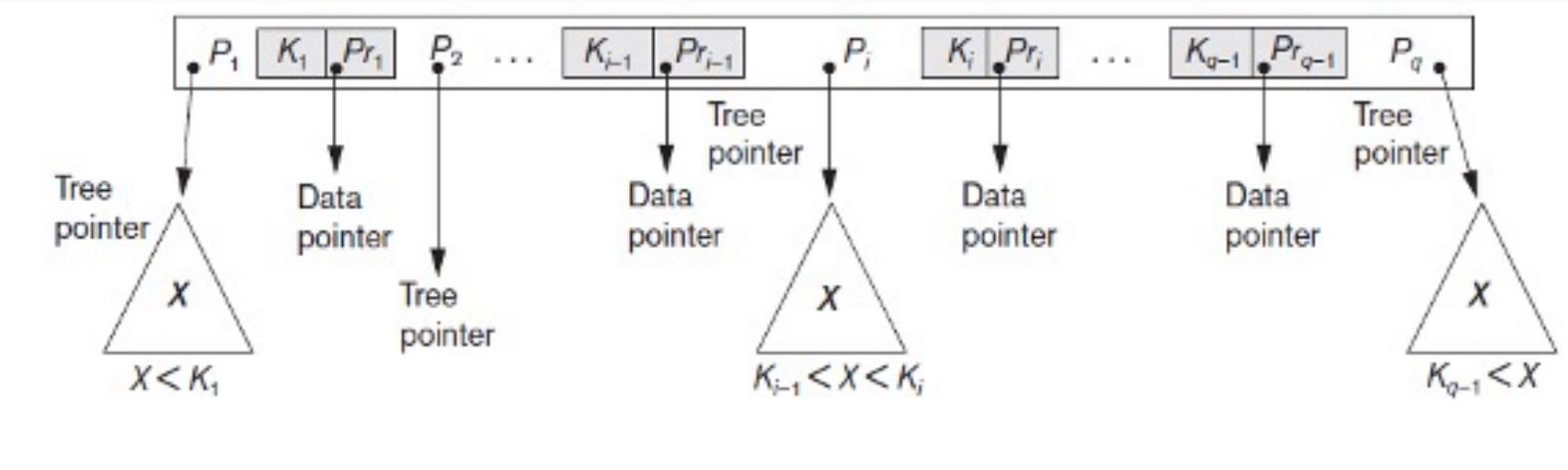
- need to ensure $\lceil (n+1)/2 \rceil$ and n bounds
- there are special smart algorithms (Check them!)
- the height and number of nodes may change

Dynamic Multilevel Indexes: B-trees

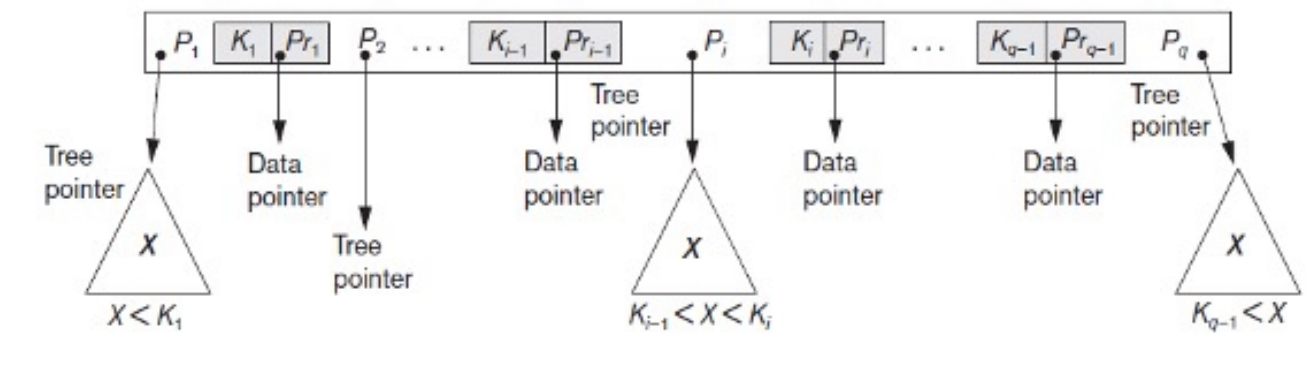
B-tree of order n

(same as B+, except that the inner nodes have pointers to data):

- all leaf nodes are at the same level
- each node has at least $\lceil (n+1)/2 \rceil$ and at most n search keys with pointers and $+1$ sub-tree pointers (except the root, which may have less, and leaves, where "sub-tree" pointers are nulls)
- Inner node: interleaving pointers are to sub-trees and data
- Leaf node: only keys and data-pointers (+ sub-tree nulls)



Dynamic Multilevel Indexes: B-trees



Search with B-trees:

- from the root to leaves—that is, the number of block accesses is at most the height (better than for B+, but the height might be larger)
- interval search is also fast

Insertion & deletion with B-trees:

- need to ensure $\lceil (n+1)/2 \rceil$ and n bounds
- there are smart algorithms (harder and slower than for B+; check them!)
- the height and number of nodes may change

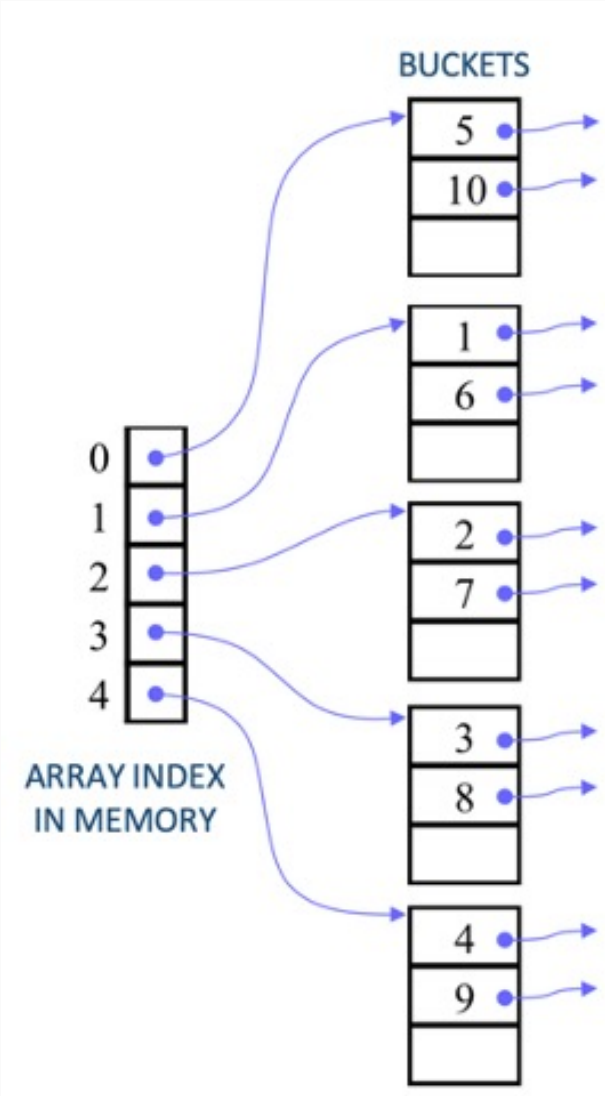
Another approach: Hash-based index

Hash-based index with hash function h (essentially, two-level index with one second-level block, with hash-values instead of search keys):

- first-level blocks are called buckets
- keys in buckets are not consecutive
- second-level block stores all possible hash values $h(key)$
- bounded by the size of a block

Common hash function $\mathrm{mod}(key, base)$:

- $base$ is usually a prime number
- ensures good distribution over buckets



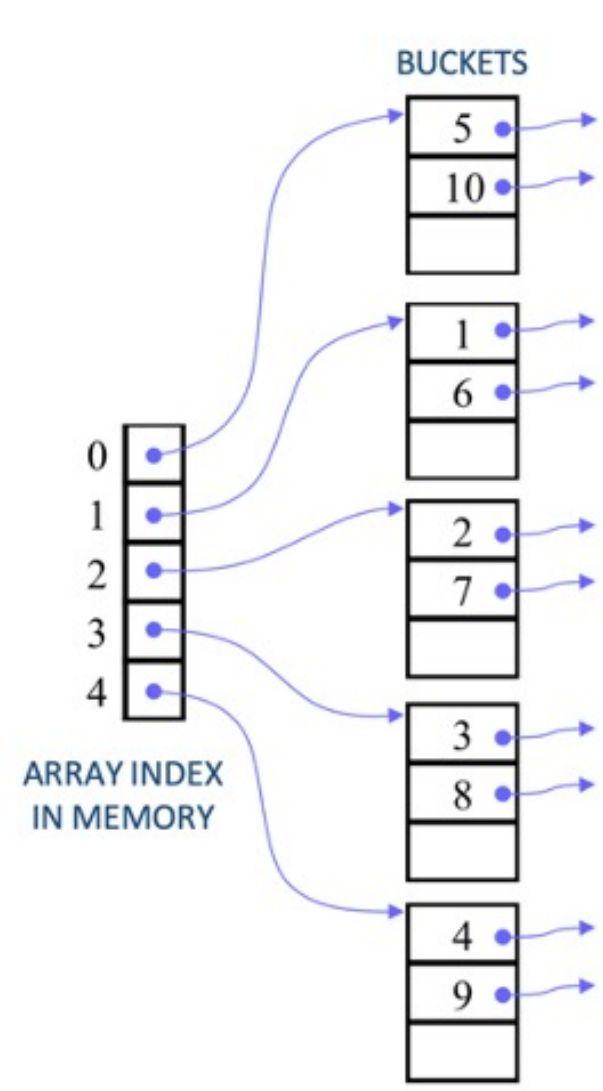
Hash-based index: Pros and cons

Hash-based index pros:

- fewer block accesses than with B & B+ for exact search (e.g., `R.A` $= 1245$)
  (hash values are shorter than keys)
- updates are easy when there is space

Hash-based index cons:

- poor for interval search (e.g., `R.A` > 1245)
- what if there is no space in the bucket on update?
  - overflow buckets
  - dynamic hashes (extensible, linear)
    (check it out!)



4. Indexes in practice

Indexes in DBMSs

Usually, DBMSs automatically create an index on every primary key

Indexes on other attributes can be created manually

The syntax is system dependent; for example:

`CREATE INDEX NAME ON` relName `(` attName `)`

`CREATE INDEX NAME ON` relName `(` attName1, `...`, attNameN `)`
(see the meaning below)

`DROP INDEX` indexName

Only some (e.g., professional) DBMSs allow us to specify the details of created index—that is, the type (e.g., B-tree, hashing), its parameters (e.g., B-tree order, hash function), etc.

Indexes in PostgreSQL

Default index type in PostgreSQL is B-tree

`https://www.postgresql.org/docs/14/indexes.html`:

> *"B-trees can handle equality and range queries on data that can be sorted into some ordering"*

PostgreSQL also has GiST:

> a mechanism for implementing indexes for complex data types.

You can see the effect of indexes in PostgreSQL

```
CREATE TABLE t1(id int);

INSERT INTO t1
  SELECT n*random()
  FROM generate_series(1, 10000000) AS x(n);

CREATE TABLE t2 AS (SELECT * FROM t1);


CREATE INDEX t1_ind ON t1(id);


\timing on

SELECT * FROM t1 WHERE id = 100; --> Time:  0.792 ms

SELECT * FROM t2 WHERE id = 100; --> Time:  303.022 ms
```

5. Multi-dimensional indexes

Multi-attribute search: Intro

What to do with multi-attribute search?

```
SELECT * FROM R WHERE A = 30 AND B < 5
```

Naive strategy:

1. Use an index on A to fetch all tuples with $A = 30$
2. Filter the tuples $B < 5$

Not very efficient: a lot of unnecessary block accesses

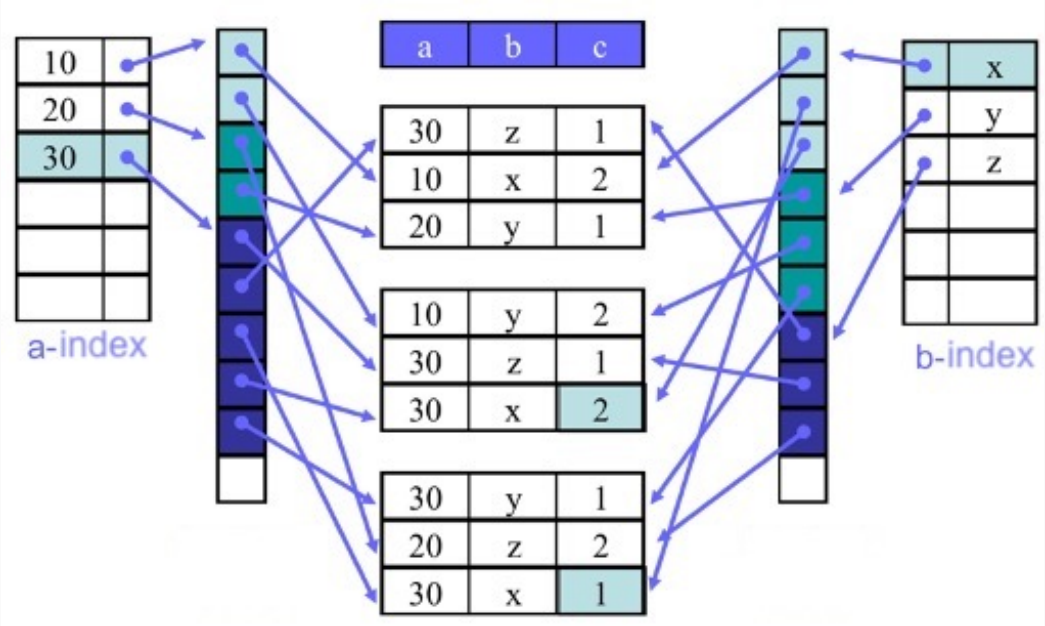
Question: Any ideas how to do it better?

Multi-attribute search: Better strategy

```
SELECT * FROM R WHERE A = 30 AND B < 5
```

More advanced strategy:

1. find all pointers(!) to tuples with $A = 30$ using a dense index for A
2. find all pointers(!) to tuples with $B < 5$ using a dense index for B
3. intersect the sets of pointers and fetch the relevant tuples

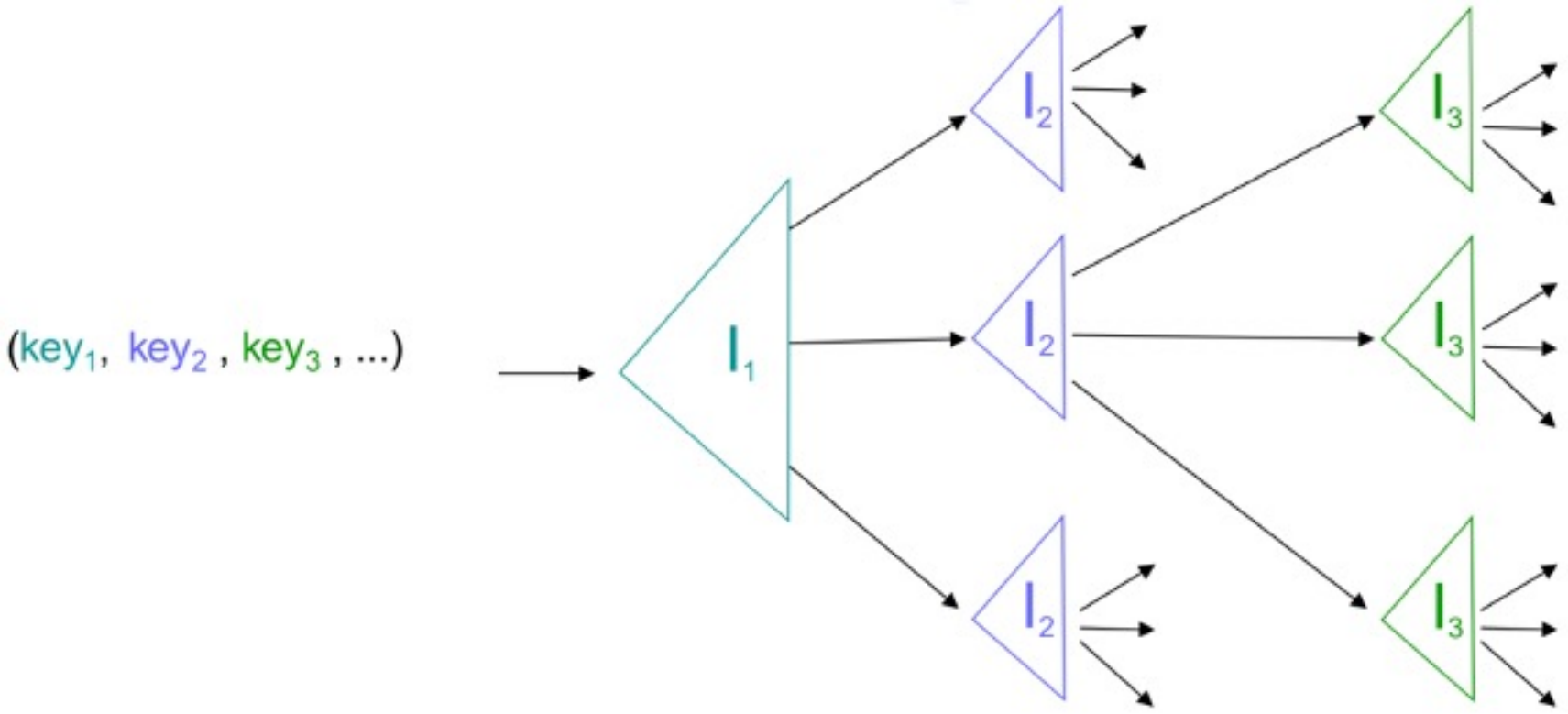


But again: two dense indexes does not look like the best solution

Multiple-Key Indexes

Multiple-Key index (hierarchical multi-dimensional approach):

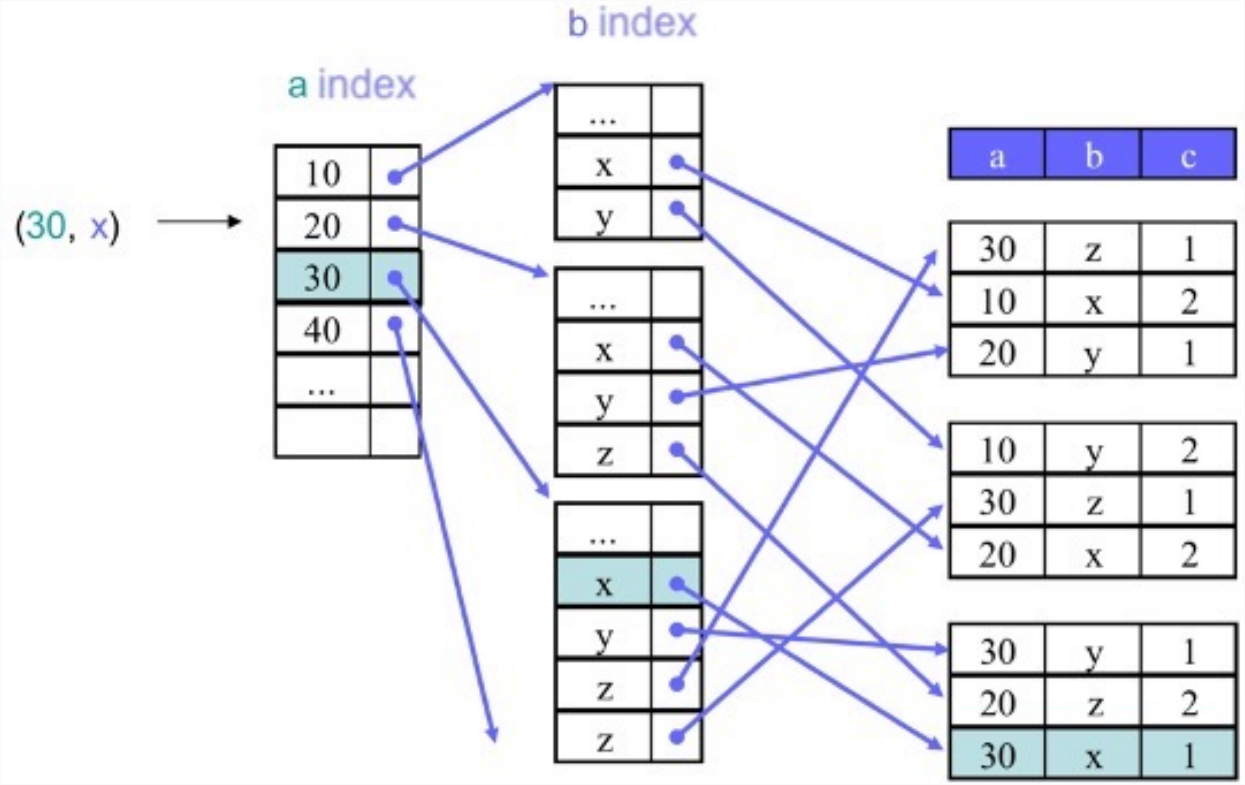
an own index for inferior dimensions for every exterior keys



Multiple-Key Index: Example

```
SELECT * FROM R WHERE A = 30 AND B = 'x'
```

Search first for A = 30, then for B = 'x' in the index for A = 30



Multiple-Key indexes: When to use

This index is effective for:

```
SELECT * FROM R WHERE A = 30 AND B = 'x'      ✓
SELECT * FROM R WHERE A = 30 AND B > 'x'      ✓
SELECT * FROM R WHERE A = 10      ✓
SELECT * FROM R WHERE B = 'x'      ✗
SELECT * FROM R WHERE A >= 30 AND B = 'x'      ✗
```

Would work with other dimension order (another index)

But there are many other alternatives:

- Other multidimensional tree-like structures
- Multidimensional hash-like structures
- Bitmap indexes

Multidimensional search visualisation

We can visualise two-dimensional search as a map:

```
SELECT * FROM R WHERE a = 30 AND b = 'x'
```

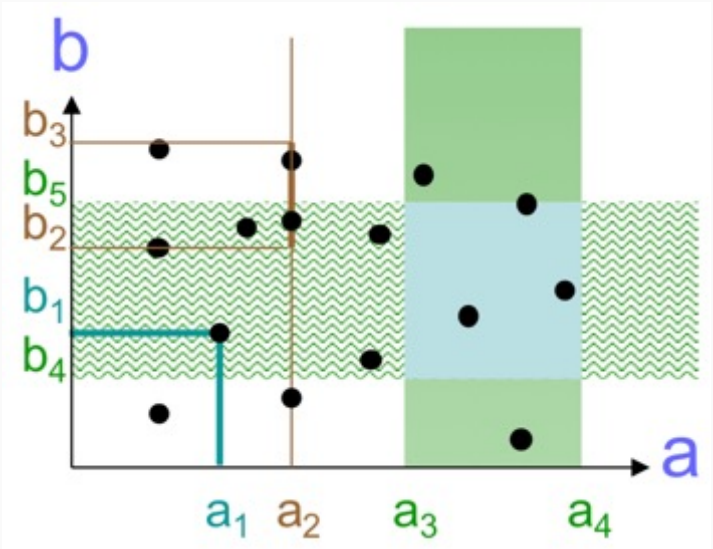
point

```
SELECT * FROM R WHERE a = 30 AND b >= 'x' AND b <= 'y'
```

segment

```
SELECT * FROM R WHERE a >= 30 AND a <= 40 AND b >= 'x' AND b <= 'y'
```

area



(Advanced) Multidimensional tree structures

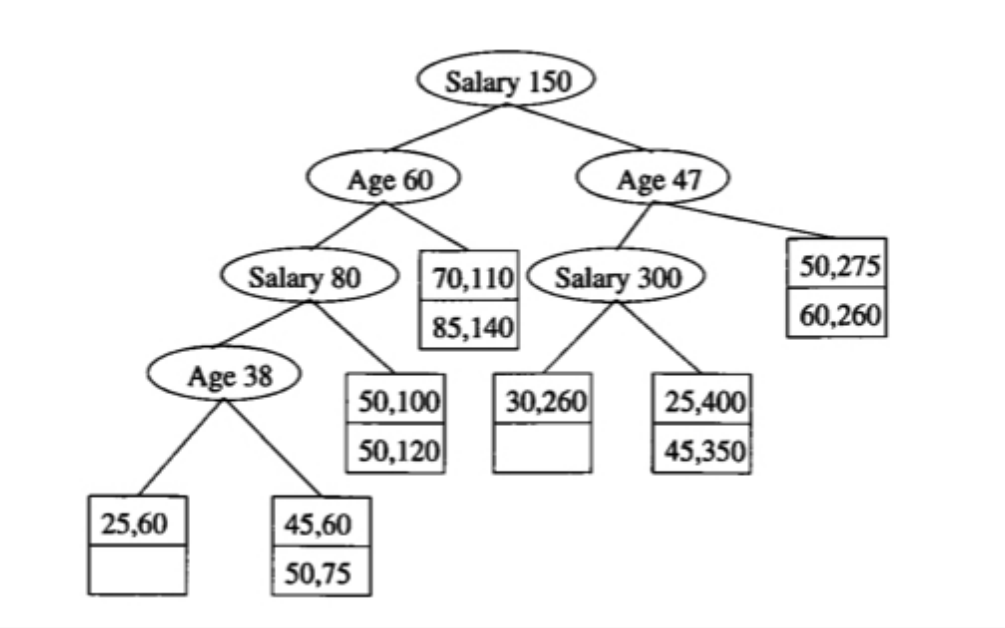
There are tree-like structures for multidimensional search, inheriting from B(+)-trees in some ways (balancing, easy updates, etc.):

- *k*-d trees: a binary search tree in which intermediate nodes are splitting values for individual dimensions every leaf node is a block of k-dimensional points
- Quad-trees: a tree data structure where each internal node has exactly four children (used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions)
- R-trees: tree data structures used for spatial access methods (i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons)
- ...

k-d trees: idea

k-d trees: a binary search tree in which intermediate nodes are splitting values for individual dimensions every leaf node is a block of k-dimensional points (dimensions 'circulate' along a branch)

Illustration:

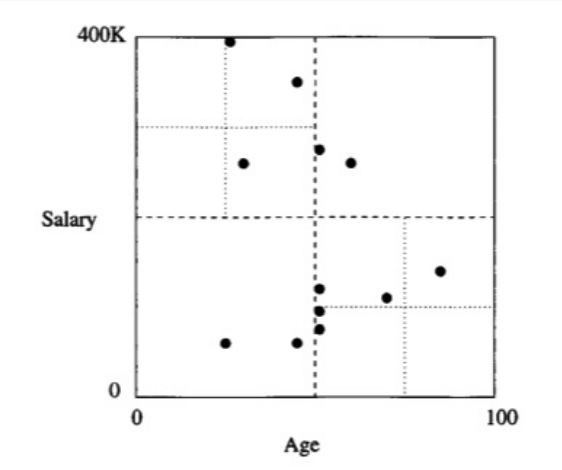


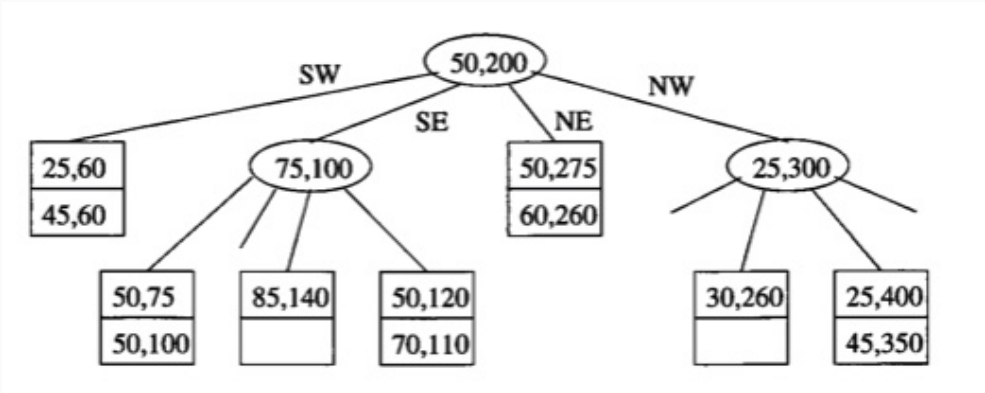
Useful for range and nearest neighbour searches

Quad-trees: idea

Quad-tree: a tree data structure where each internal node has exactly four children (often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions)

Illustration:





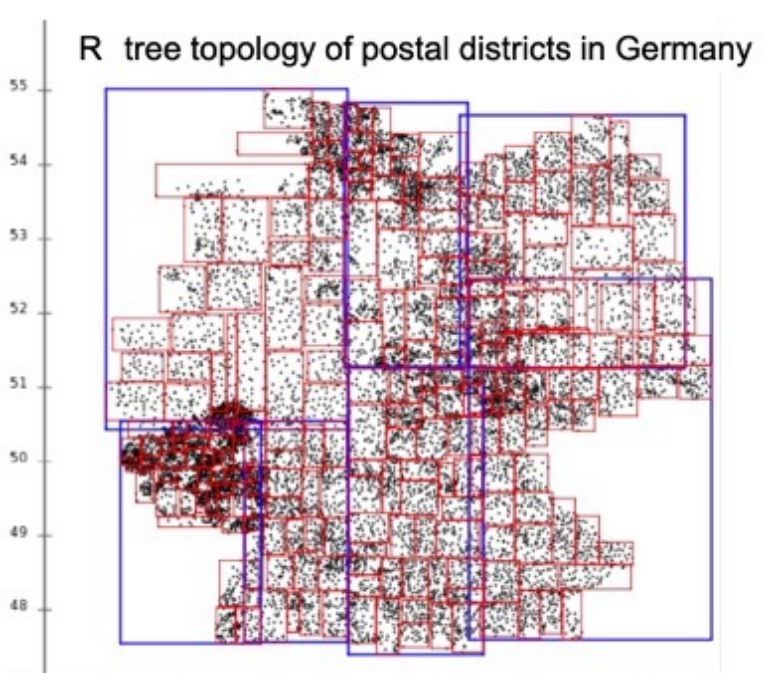
R-trees: idea

Balanced tree, where nodes group close points

- inner node: smallest rectangle containing all subtree points (may overlap)
- leaf node: point (with a pointer to a tuple)

Search (incl. intersection, contained in, nearest neighbour): easy

Insertion: challenging



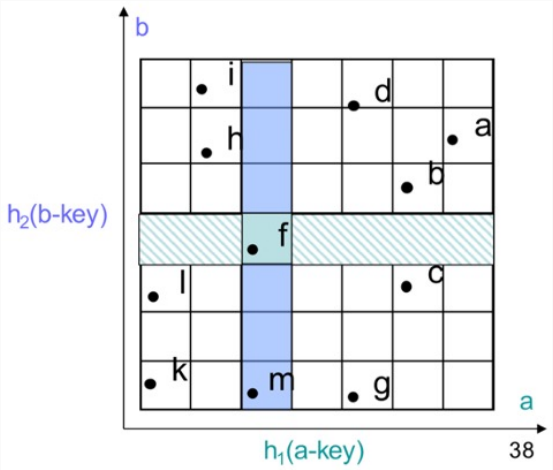
Multidimensional hash-like structure: Grid files

Grid files:

- 'Hash' function h_d for each dimension d individually
- 'Hashes' to the position number in the d-dimension of the matrix of bucket pointers: $h_d(key) = n$, where n is the number of $[x_d, y_d)$ with $key \in [x_d, y_d)$ in the dimension d

Example (2 dimensions): Find record with $(a, b) = (22, 31)$

- Apply 'hashes' $h_a(22) = \#[20, 30)_a = 3, \quad h_b(31) = \#[0, 100)_b = 4$
- Retrieve the pointer to the bucket for cell (3, 4) from pointer matrix



Search: easy

Insertion & Deletion: inherit the problems of usual hashes

Bitmap indexes

Bitmap index:

- each value of each indexed attribute has a bitmap vector
- each tuple has a component in each vector

Efficient for several attributes with small domains

Take a lot of space, but can be compressed

Search: bit-wise AND

Insertion & Deletion: simple for existing values

File

record-number	F	G	H
1	30	foo	65
2	30	bar	43
3	40	baz	84
4	50	fou	43
5	40	bar	65
6	30	baz	65

Bitmap vectors for F

30:	1	1	0	0	0	1
40:	0	0	1	0	1	0
50:	0	0	0	1	0	0

Bitmap vectors for G

bar:	0	1	0	0	1	0
baz:	0	0	1	0	0	1
foo:	1	0	0	0	0	0
fou:	0	0	0	1	0	0

What have we learned?

Indexes:

The second key DBMS component for efficient query evaluation