COMP 430/533 Intro. to Database Systems

Indexing

How does DB find records quickly?

- Various forms of *indexing*
- An index is automatically created for primary key.
- SQL gives us some control, so we should understand the options.
  - Mainly concerned with user-visible effects, not underlying implementation.

```
CREATE INDEX index_city_salary
ON Employees (city, salary);
```

```
CREATE UNIQUE CLUSTERED INDEX idx
ON MyTable (attr1 DESC, attr2 ASC);
```

Options vary.
Not PostgreSQL.

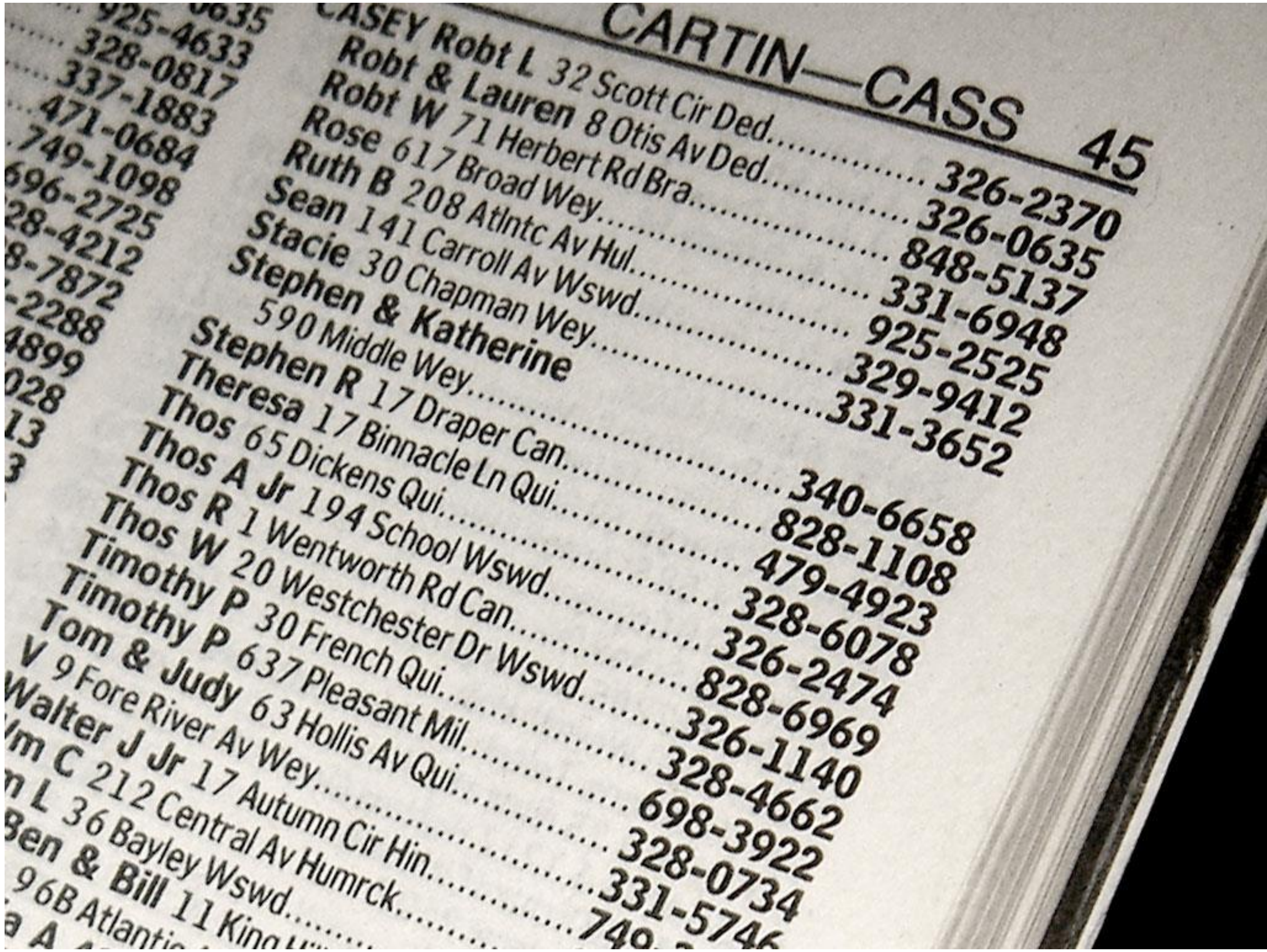
Phone book model

Data is stored with search key.

Clustered index.

Organized by **one search key**:

- Last name, first name.
- Searching by any other key is slow.



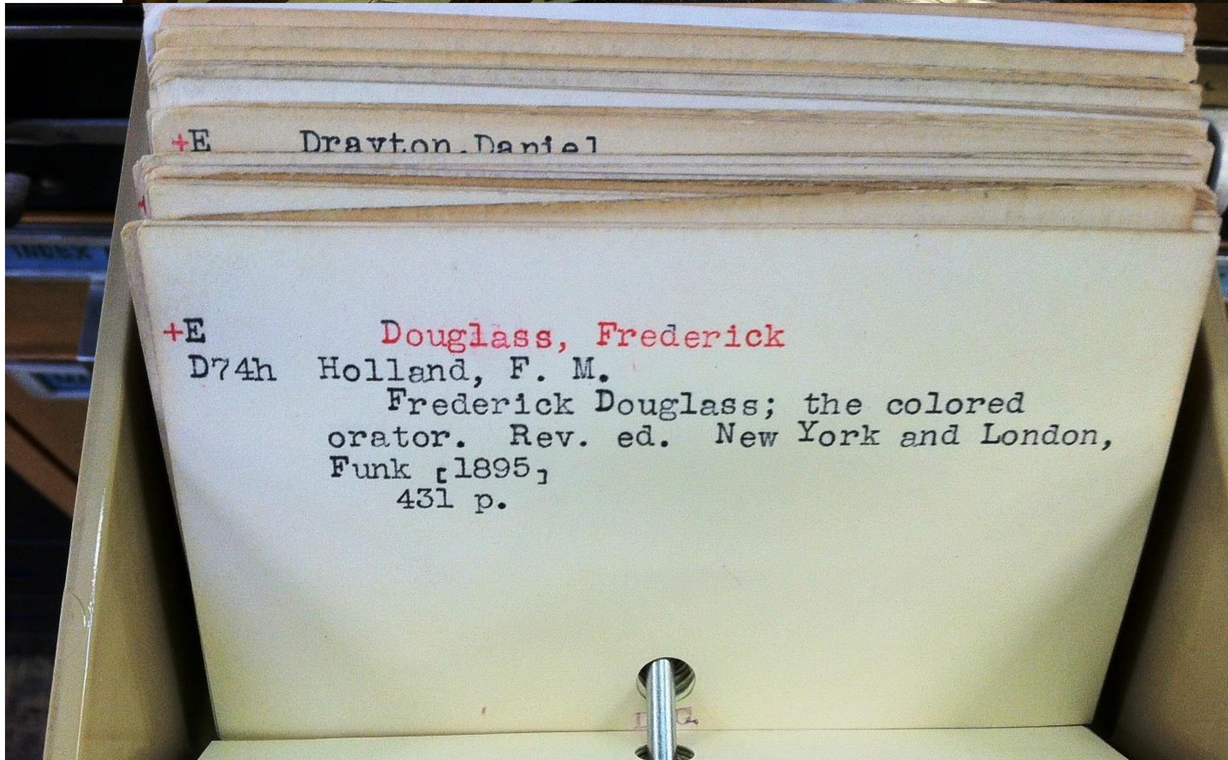
Library card catalog model

Index stores pointers to data.

Non-clustered index.

Organized by search key(s).

- Author last name, first name.
- Title
- Subject



Evaluating an indexing scheme

- Access type flexibility
  - Specific key value – e.g., “John”, “Smith”
  - Key value range – e.g., salary between \$50K and \$60K

Advantage:

- Access time

Disadvantages:

- Update time
- Space overhead

Creating an index can slow down the system!

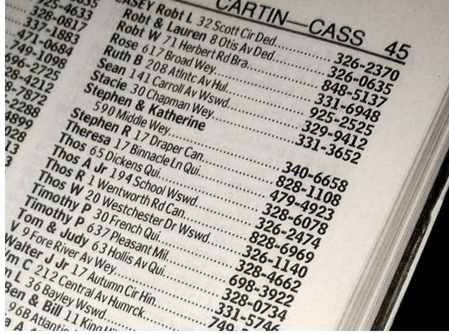
Ordered indices

Sorted, as in previous human-oriented examples

Types of indices we’ll see

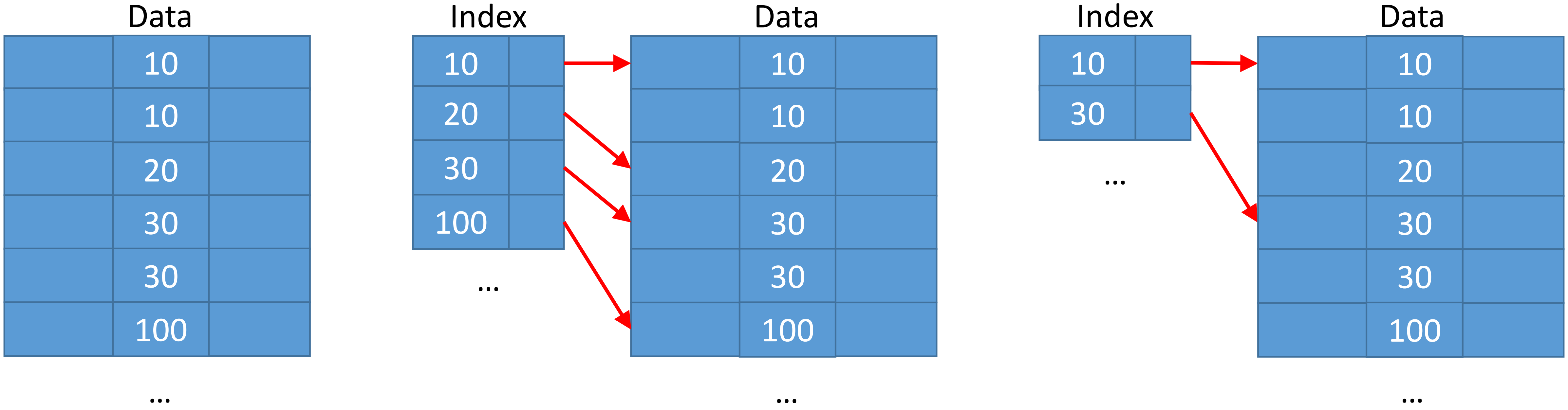
- Clustered vs. non-clustered
- Dense vs. sparse
- Unique data vs. not-unique
- Single- vs. multi-level

These ideas can be combined in various ways.



Clustered index – dense or sparse

No index | **Dense index** | **Sparse index**



- Common for primary keys
- Not supported in PostgreSQL
- Sparse – typically pointer to top of each file block

- Faster access?
- Faster insert, update, delete?
- Less space?

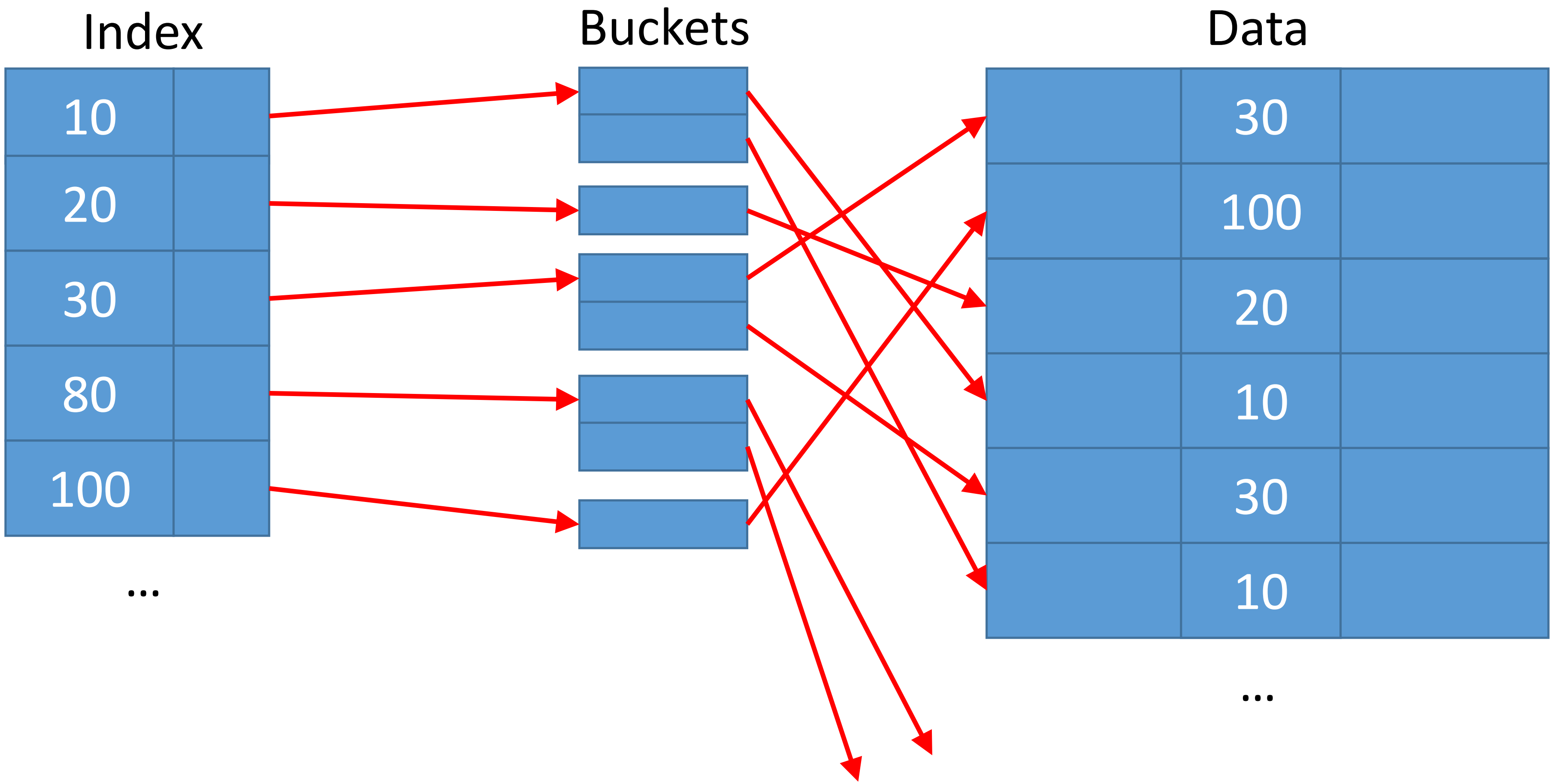
PostgreSQL & clustering

- Clustered index not supported

- Can cluster data (i.e., physically sort) for efficient access.
  - This sorted order is **not** maintained after CRUD operations.

```
CLUSTER Employees USING index_city_salary;
```

Simple non-clustered index – must be dense



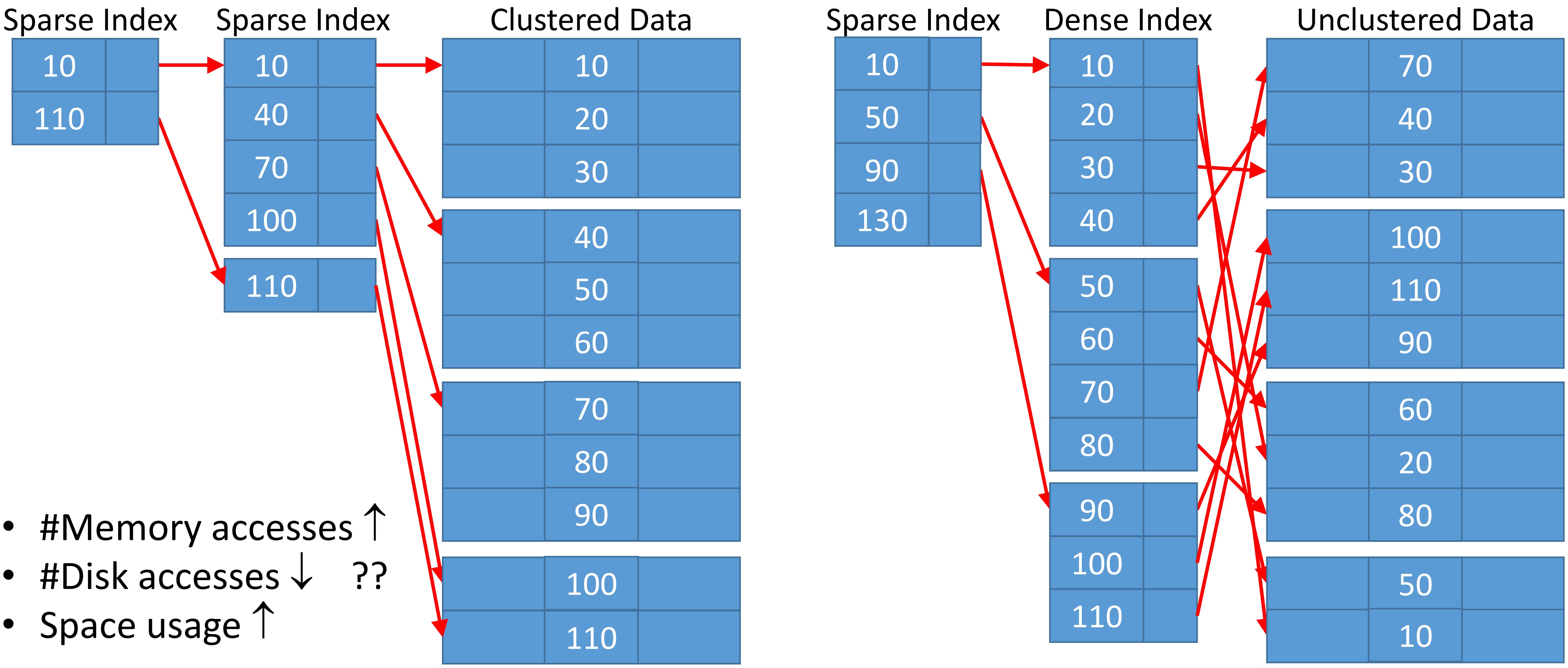
So far, index size proportional to #values

For efficient access, want index to fit in memory.

Solution: Multi-level index

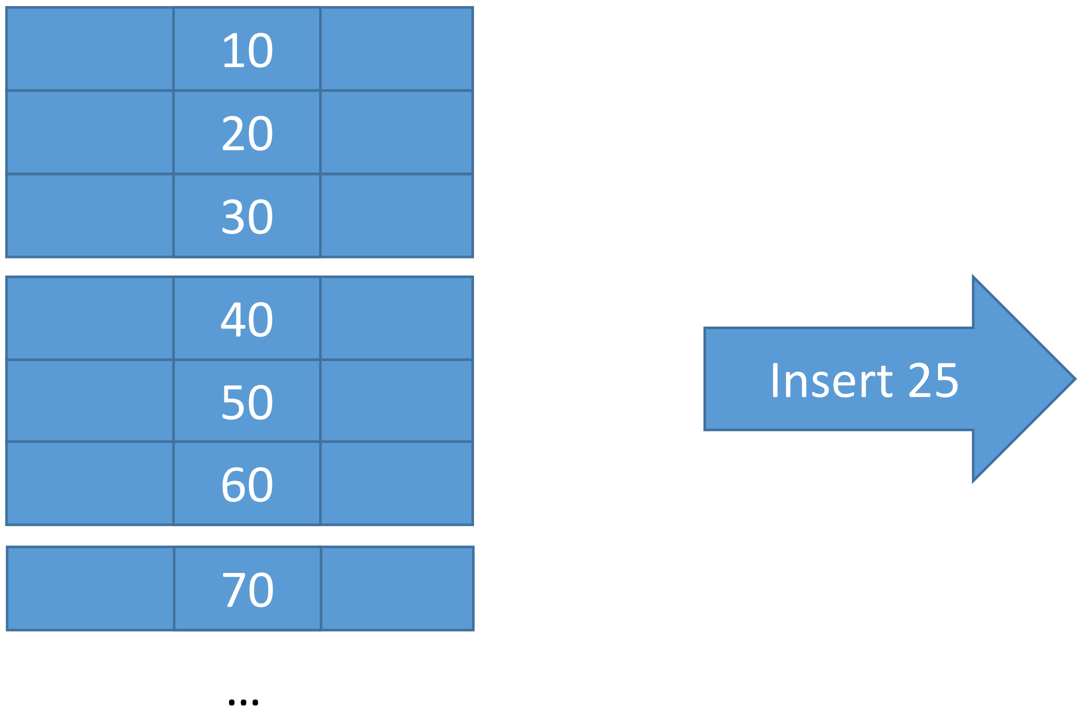
Same problem & solution as for page tables in virtual memory.

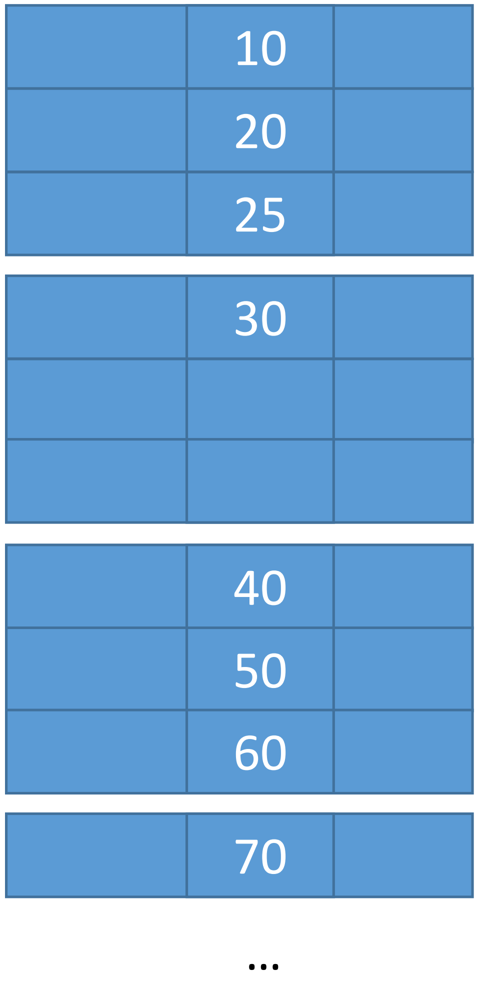
Multi-level indices



- #Memory accesses ↑
- #Disk accesses ↓ ??
- Space usage ↑

CRUD on dense indices or clustered data





Moving all records is too expensive.
Otherwise, results in fragmentation.

Fragmentation consequences

Fragmentation (empty spaces) increases with usage.

Performance degrades with usage.

Need to periodically reorganize data & indices.

Solution: B+-trees

Ordered indices – B+-trees

The most commonly used indexing algorithm

B+-trees

- Balanced search trees optimized for disk-based access
  - Reorganizes a little on every update
  - Shallow & wide (high fan-out)
  - Typically, node size = disk block

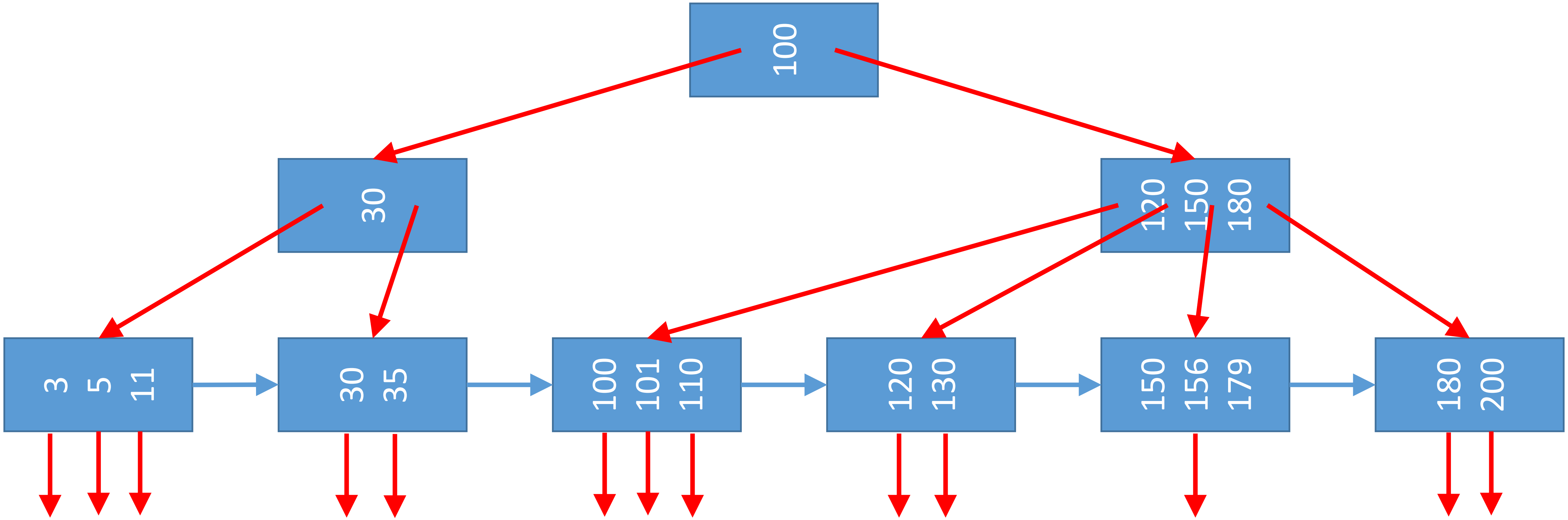
- Slight differences from more commonly-known B-trees
  - All data in leaf nodes
  - Leaf nodes sequentially linked

Easily get all data in order.

```
CREATE INDEX … ON … USING BTREE;
```

Or, is the default in many DBMSs.

B+-tree example



… Data records …

B+-tree performance

- Very similar to multi-level index structure

− Slightly higher per-operation access & update time

+ No degradation over time
+ No periodic reorganization

B+-tree widely used in relational DBMSs

What about non-unique search keys?

- Can allow duplicates in tree.
  - Maintain $\leq$ order instead of $<$.
  - Complicates details.

- Can either return first or all duplicates.
  - Easy, since duplicates are adjacent.

Indexing on VARCHAR keys

- Key size variable, so number of keys that fit into a node also varies.

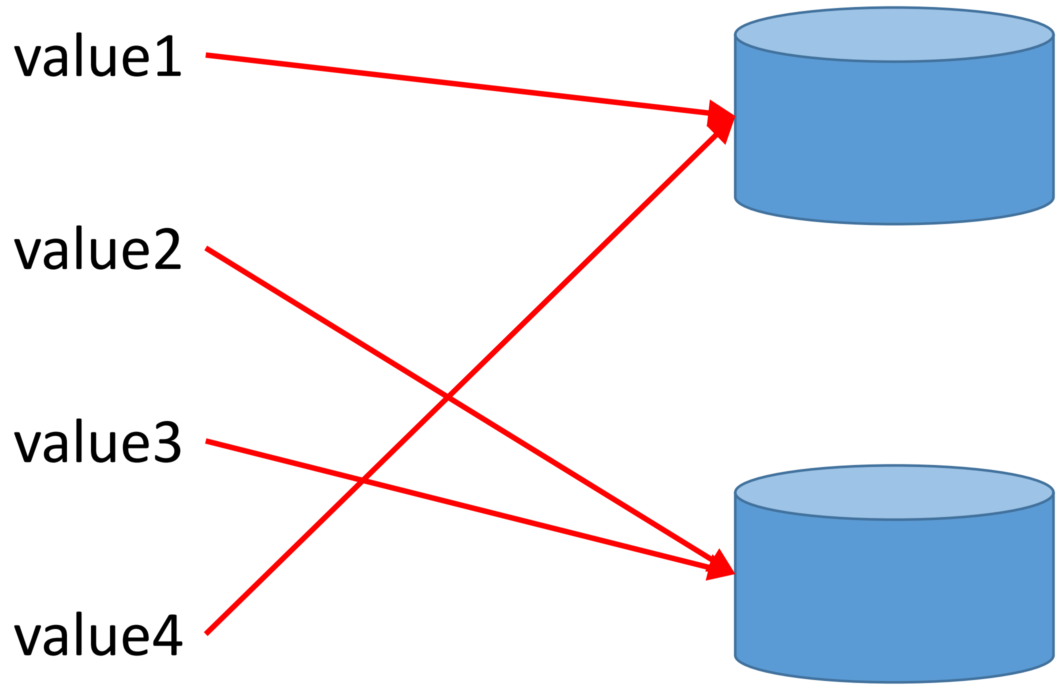
- Techniques to maximize fan-out:
  - Key values at internal nodes can be prefixes of full key. E.g., “Johnson” and “Jones” can be separated by “Jon”.

  - Key values at leaf nodes can be compressed by sharing common prefixes. E.g., “Johnson” and “Jones” can be stored as “Jo” + “hnson”/”nes”

Hash indices

An alternative to ordered indices

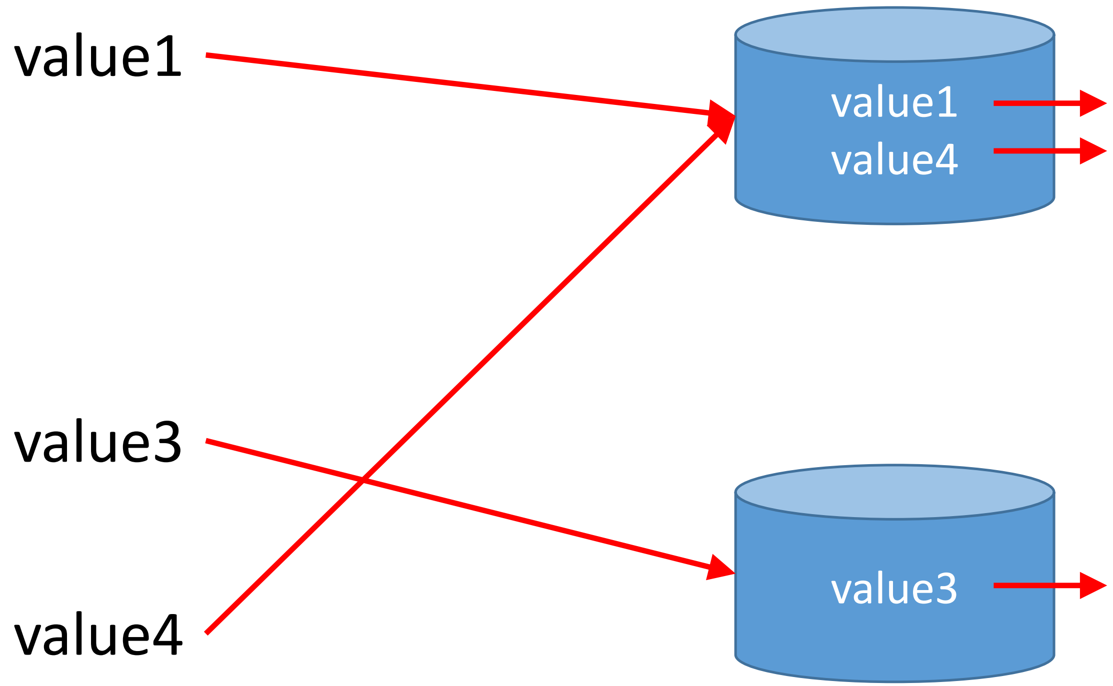
Basic idea of a hash function



h() distributes values uniformly among the buckets.

h() distributes typical subsets of values uniformly among the buckets.

Using hash function for indexing



Motivation: Constant-time hash instead of multi-level/tree-based.

Use hash function to find bucket.
Search/insert/delete from bucket.
Bucket items possibly sorted.

Buckets can overflow

- Overflow some buckets – skewed usage
- Overflow many buckets – not enough space reserved

Solutions:

- Chain additional buckets – degrades to linear search
- Stop and reorganize
- Dynamic hashing (extensible or linear hashing) – techniques that allow the number of buckets to grow without rehashing existing data

Advantages & disadvantages of hash indexing

+ One hash function vs. multiple levels of indexing or B+-tree

- Storage about the same.
    + Don’t need multiple levels.
    − But, need buckets about half empty for good performance.

- Can’t easily access a data range. Thus, poor for relations like >.
- Simple hashing degrades poorly when data is skewed relative to the hash function.

Multiple indices & Multiple keys

Using indices for multiple attributes

What if queries like the following are common?

```
SELECT …
FROM Employee
WHERE job_code = 2 AND performance_rating = 5;
```

Strategy 1 – Index one attribute

```
CREATE INDEX idx_job_code on Employee (job_code);

SELECT …
FROM Employee
WHERE job_code = 2 AND performance_rating = 5;
```

Internal strategy:

1. Use index to find Employees with job_code = 2.
2. Linear search of those to check performance_rating = 5.

Strategy 2 – Index both attributes

```
CREATE INDEX idx_job_code on Employee (job_code);
CREATE INDEX idx_perf_rating on Employee (performance_rating);

SELECT …
FROM Employee
WHERE job_code = 2 AND performance_rating = 5;
```

Internal strategy – chooses between

- Use job_code index, then linear search on performance_rating.
- Use performance_rating index, then linear search on job_code.
- Use both indices, then intersect resulting sets of pointers.

Strategy 3 – Index attribute set

```
CREATE INDEX idx_job_perf on Employee (job_code, performance_rating);

SELECT …
FROM Employee
WHERE job_code = 2 AND performance_rating = 5;
```

Attribute sets ordered lexicographically:

This strategy typically uses ordered index, not hashing.

(jc1, pr1) < (jc2, pr2) iff either

- jc1 < jc2
- jc1 = jc2 and pr1 < pr2

Note that this prioritizes job_code over performance_rating!

Strategy 3 – Index attribute set

Efficient

```
CREATE INDEX idx_job_perf on Employee (job_code, performance_rating);

SELECT … FROM Employee WHERE job_code = 2 AND performance_rating = 5;

SELECT … FROM Employee WHERE job_code = 2 AND performance_rating < 5;
```

Inefficient

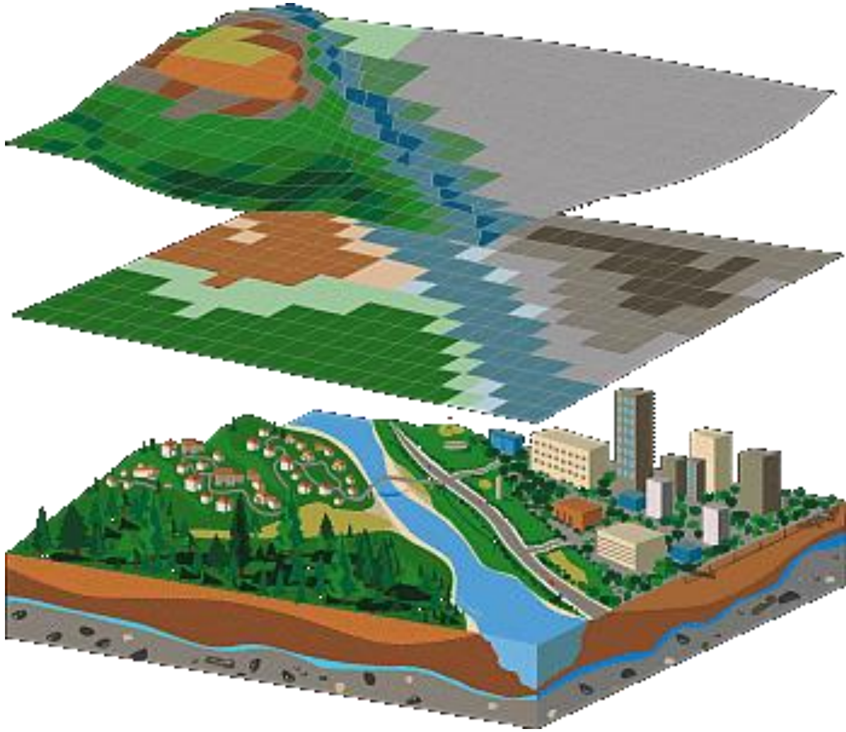
```
CREATE INDEX idx_job_perf on Employee (job_code, performance_rating);

SELECT … FROM Employee WHERE job_code < 2 AND performance_rating = 5;

SELECT … FROM Employee WHERE job_code = 2 OR performance_rating = 5;
```

Strategy 4 – Grid indexing

- n attributes viewed as being in n-dimensional grid
- Numerous implementations
  - Grid file, K-D tree, R-tree, …
- Mainly used for spatial data



- GIS DBs are a separate, specialized topic.
  - Many RDBMS's have a GIS-specialized component.

Strategy 5 – Bitmap indices

- Coming next…

Bitmap indices

One more alternative approach for indexing

Basic idea of bitmap indices

Assume records numbered 0, 1, …, and easy to find record #i.

Record #	id	gender	income_level
0	51351	M	1
1	73864	F	2
2	13428	F	1
3	53718	M	4
4	83923	F	3

Bitmaps

M	F	IL1	IL2	IL3	IL4	IL5
1	0	1	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	1	0
0	1	0	0	1	0	0

F's bitmap: 01101

Queries use standard bitmap operations

```
SELECT … FROM … WHERE gender = 'F' AND income_level = 1;
```

F's bitmap 01101 & 1's bitmap 10100 = 00100

```
SELECT … FROM … WHERE gender = 'F'  OR  income_level = 1;
```

F's bitmap 01101 | 1's bitmap 10100 = 11101

```
SELECT … FROM … WHERE income_level <> 1;
```

~ 1's bitmap 10100 = 01011

Space overhead

- Normal: #records × (#values + 1) bits, if nullable
  - Typically used when # of values for attribute is low.
- Encoded: #records × log(#values)
  - But need to use more bitmaps
- Compressed: ~50% of normal
  - Can do bitmap operations on compressed form.

A couple details

- When deleting records, either …
  - Delete bit from every bitmap for that table, or
  - Have an *existence bitmap* indicating whether that row exists.

- Can combine bitmaps with B+-trees.
  - B+-tree leaf holds list of pointers to records with a particular attribute value.
  - Can instead hold bitmap for that value.

Summary of index types

Single-level	Index generally too large to fit in memory
Multi-level	Degrades due to fragmentation
B+-tree	General-purpose choice
Hash	Good for equality tests; potentially degrades due to skew & overflow
Spatial	Good for GIS/spatial coordinates
Bitmap	Good for multiple attributes each with few values