

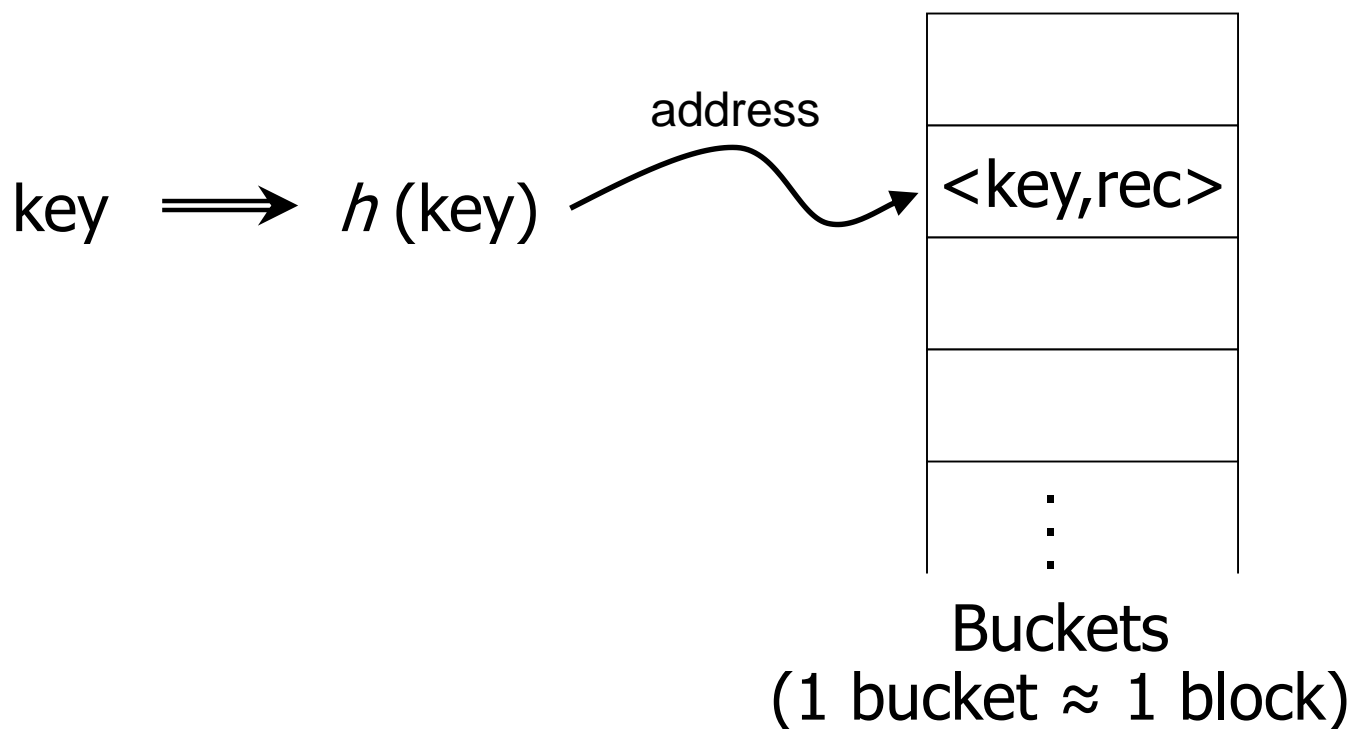


PA152: Efficient Use of DB  
**5. Hashing**

Vlastislav Dohnal

# Hashing – concept

- Key-to-address transformation
  - Based on a function returning an address
    - for an input key value

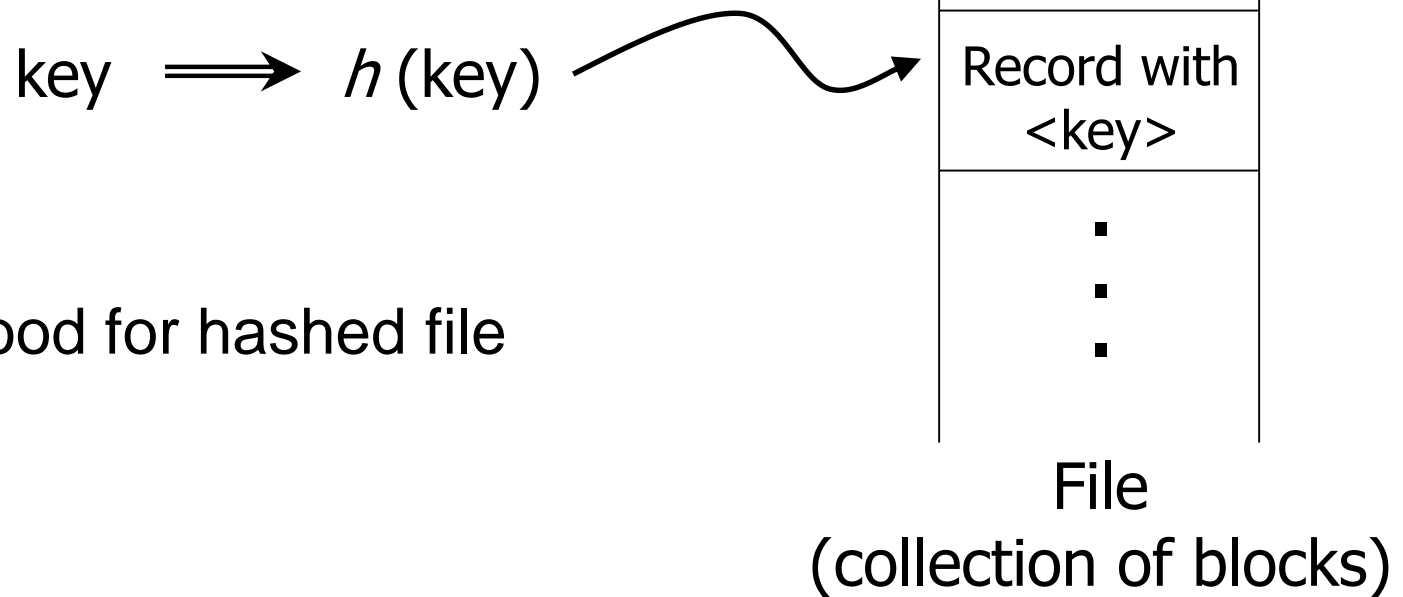


# Alternatives

- Direct addressing

- Record address

- Typically address of a block

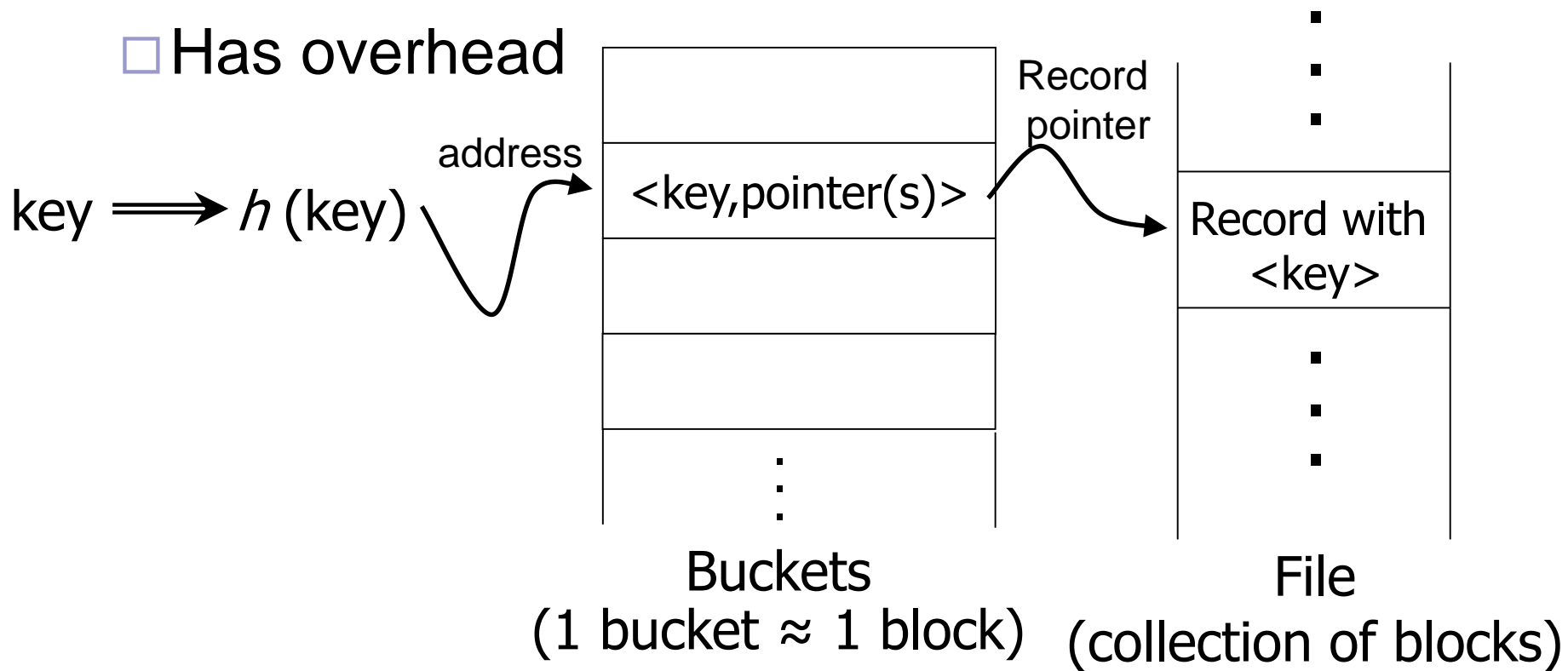


- Good for hashed file

# Alternatives

## ■ Indirect addressing

- Used in secondary indexes
- Has overhead



# Hash Function – Examples

- key is a string
  - $n$  bytes
- Address space
  - $B$  buckets
- Hash function  $h(x_0x_1\dots x_{n-1})$ 
  - $(x_0+x_1+\dots+x_{n-1}) \bmod B$
  - $(x_0*31^{n-1}+x_1*31^{n-2}+\dots+x_{n-1}) \bmod B$
  - Applicable to variable-length characters

# Hash Function

- Properties:

- Uniform

- Equal occupation of all buckets

- Random

- Low correlation between its input and output

- „Big science“ → special literature

- Good hash function must be uniform

# Address Space

- Collection of buckets
  
- Bucket
  - Capacity larger than 1 record
  - Sort the keys?
    - Yes, if access should be faster
    - Yes, if updates are rare
    - No, if updates are frequent

# Terminology

- Hash function

- Collisions

- Another record present on the returned address

- No problem if more keys/recs can be stored

- Overflows

- Bucket capacity is exhausted

- Overflow blocks

- Static vs. dynamic hashing

- Address space cardinality can be modified

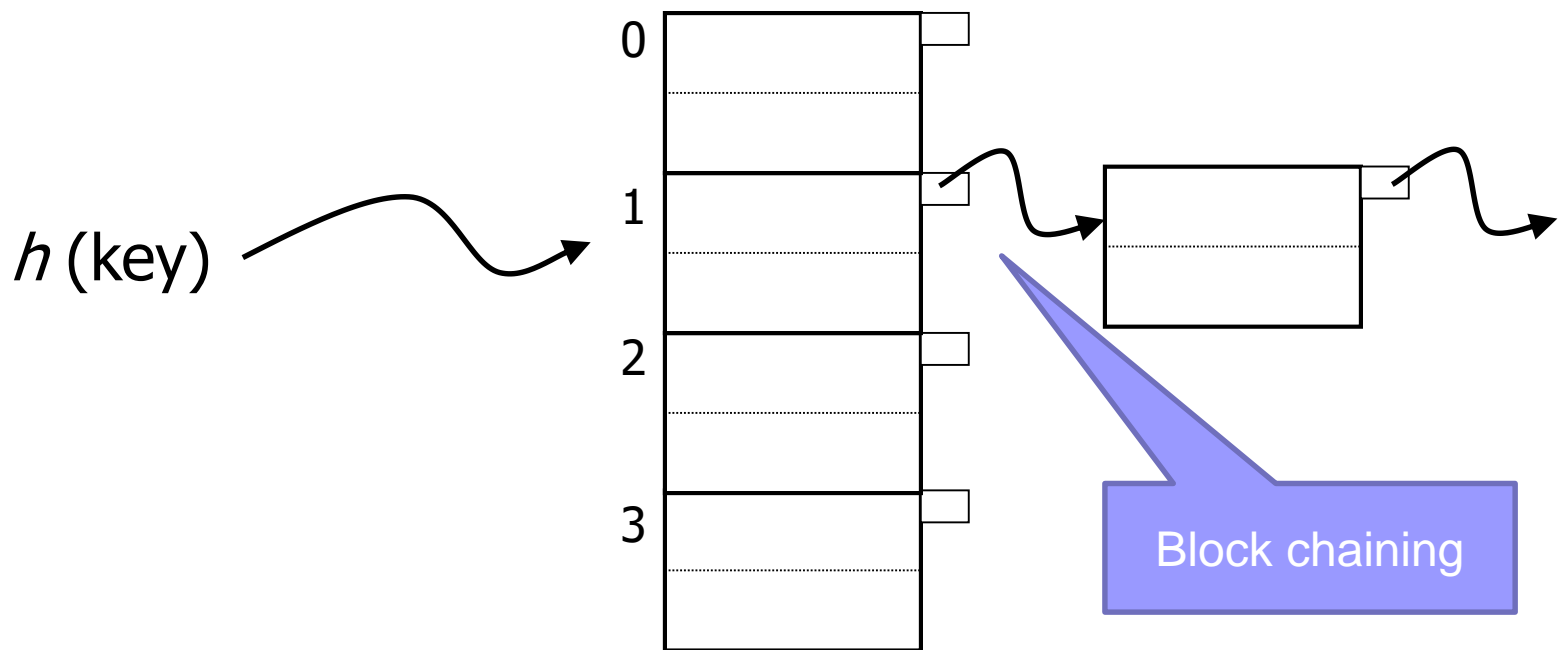


# Static Hashing: Collision Solutions

- Closed addressing (= open hashing)
  - Returned address is fixed
  - On overflow, allocate a new block (overflow area)
    - Chaining overflow blocks
- Open addressing (= closed hashing)
  - Collision function is introduced
    - Linear, quadratic, double-hashing

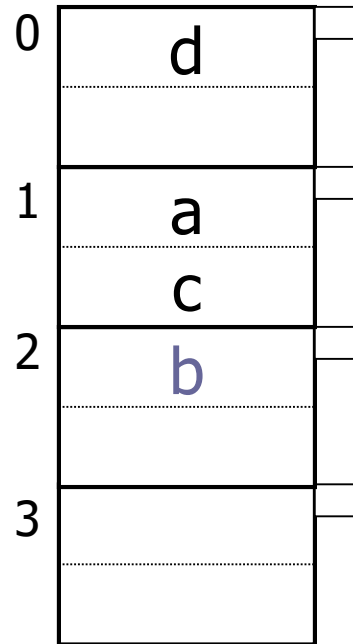
# Example

- Static closed addressing
  - Collisions handled by overflow blocks
  - Block capacity = 2 keys



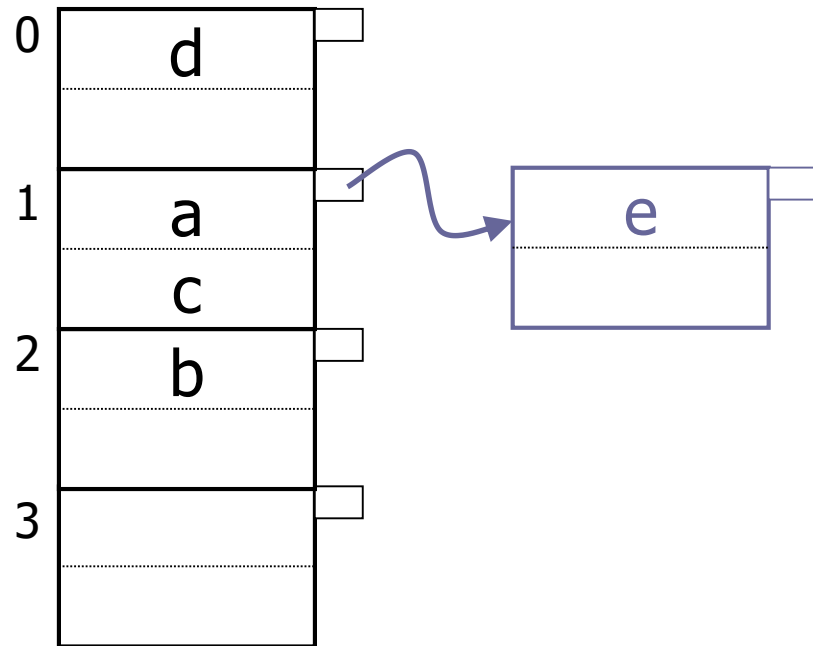
# Example: Insert

- $h(„b“) = 2$



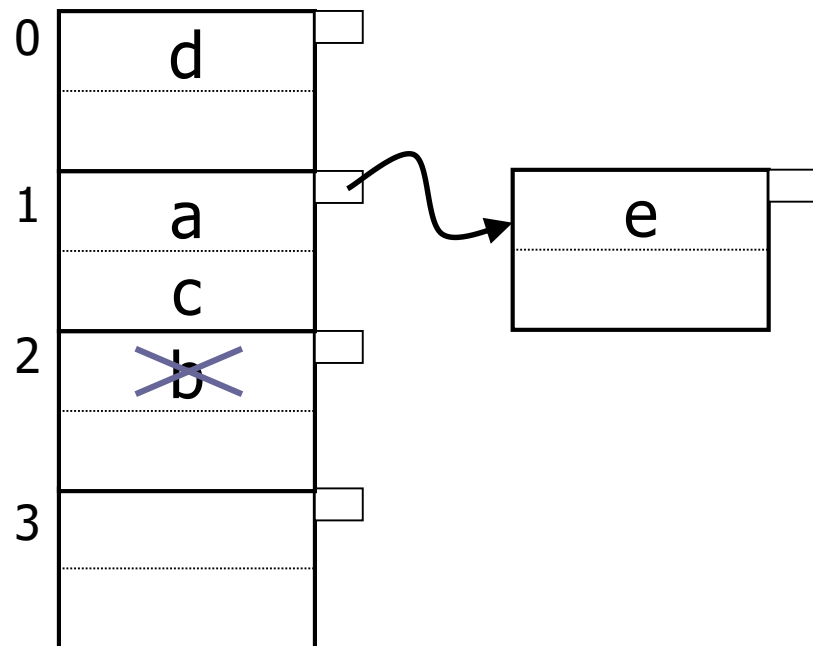
# Example: Insert

- $h(„e“) = 1$



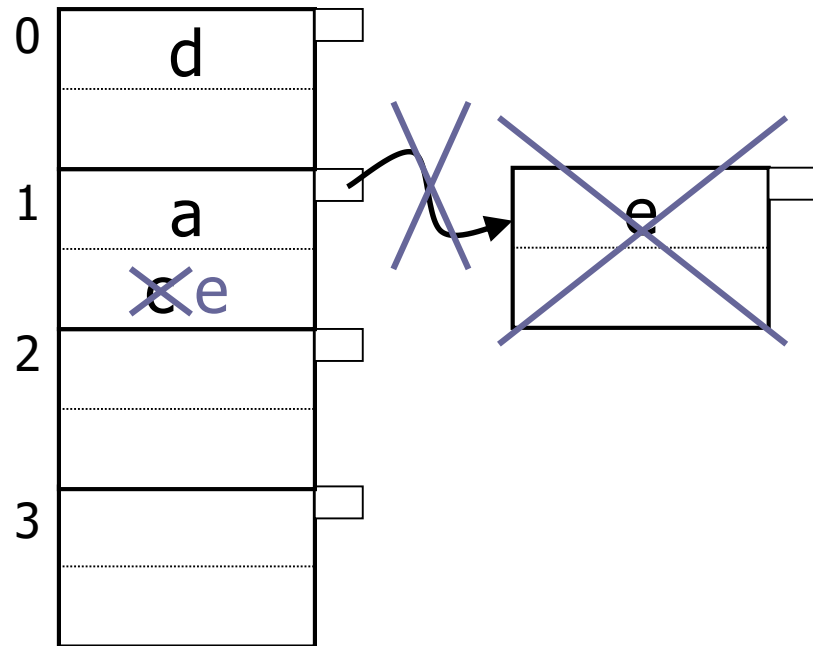
# Example: Deletion

- Free overflow blocks up
- Delete „b”



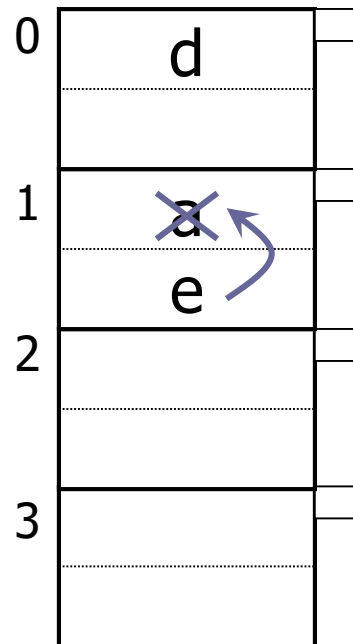
# Example: Deletion

- Free overflow blocks up
- Delete „c”



# Example: Deletion

- Free overflow blocks up
- Delete „a”



# Rule of thumb

- Keep space utilization btw. 50% and 80%
  - Utilization =  $\# \text{ stored keys} / \text{max. capacity}$  (in keys)
  - $< 50\%$   $\rightarrow$  wasting space
  - $\geq 80\%$   $\rightarrow$  too many collisions
    - Overflow areas degrade searching and insertion performance



# Dynamic Data

- Static hashing

- Overflows → reorganization

- i.e., designing a new hash function

- Dynamic hashing

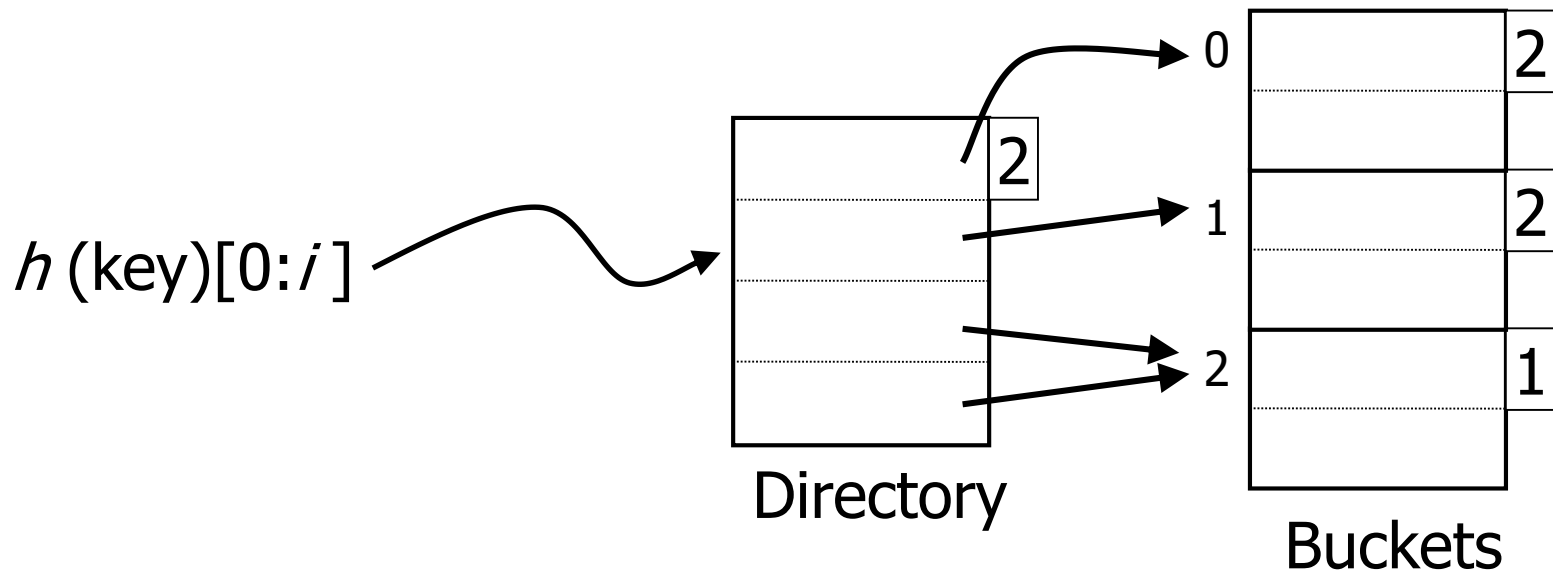
- Extendible

- Linear

# Extendible Hashing

## ■ Idea:

- Use top  $i$  of  $b$  bits output by hash function
- Indirection added – directory
  - Size is power of 2

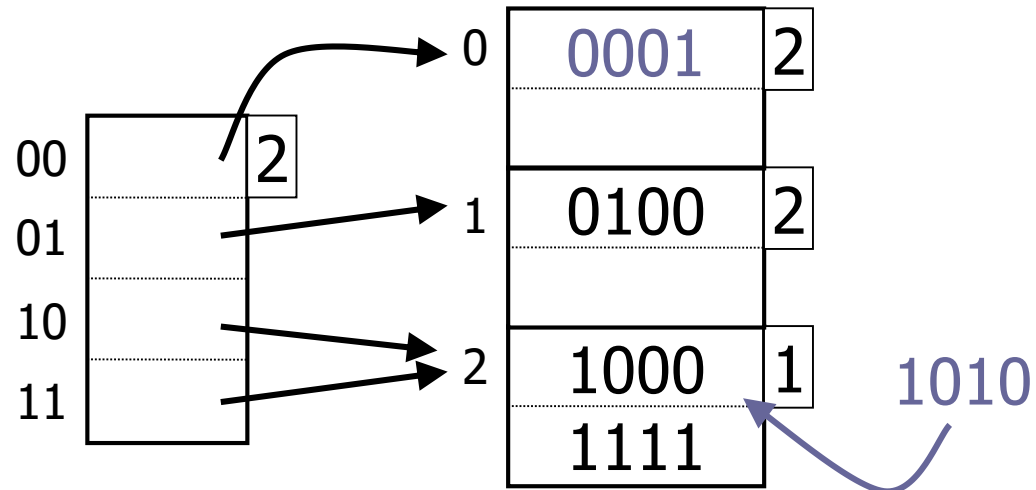


# Extendible Hashing: Insertion

- Example:  $h(x) = x$ , i.e., identity
- Locate a bucket
  - is free space available?
    - Yes  $\rightarrow$  insert;
    - No  $\rightarrow$  so split the bucket & redistribute its content

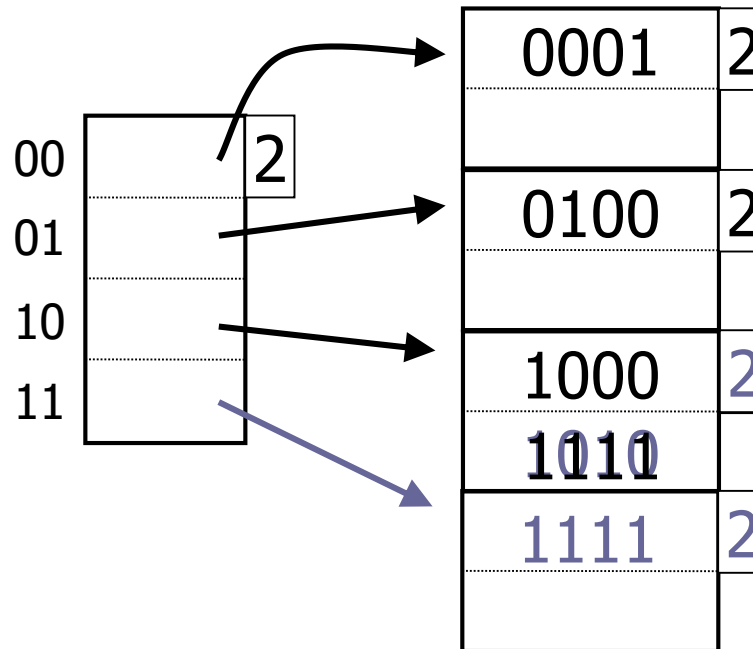
- Insert

- 0001
- 1010



# Extendible Hashing: Insertion

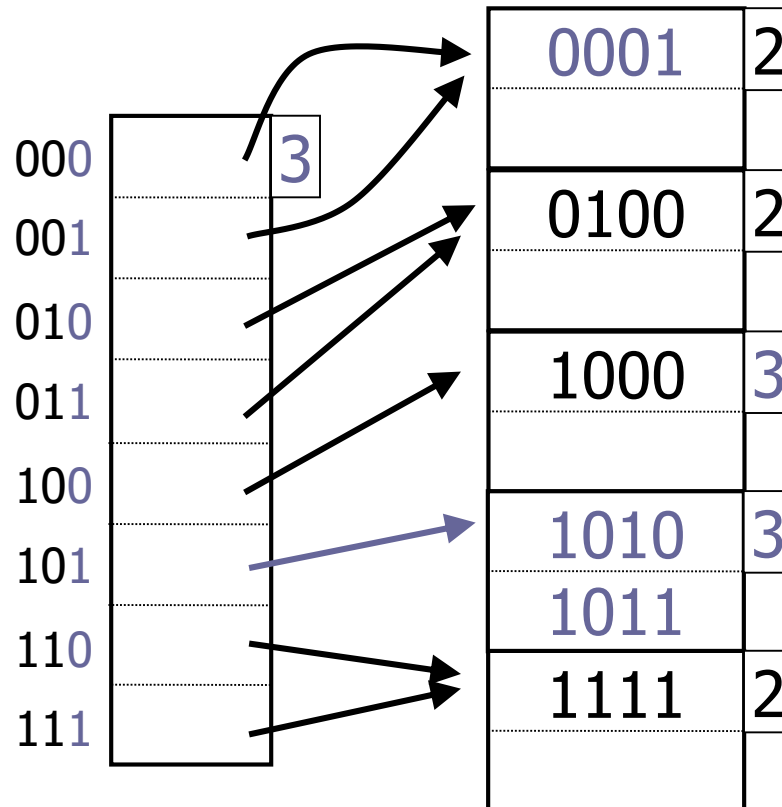
- Insert 1010
  - Split bucket



# Extendible Hashing: Insertion

- Insert 1011

- Directory is full → double it (use one more bit)



# Extendible Hashing: Deletion

- Delete keys
  - Delete key
  - Bucket gets empty
    - No operation → assume new data arriving
    - Merge neighboring buckets
      - Of the same prefix (shorted by one bit)
      - Directory can shrink too

# Extendible Hashing: Summary

## ■ Advantages

- Handles growing files
- Wasting less space (than static hashing)
- Local reorganization

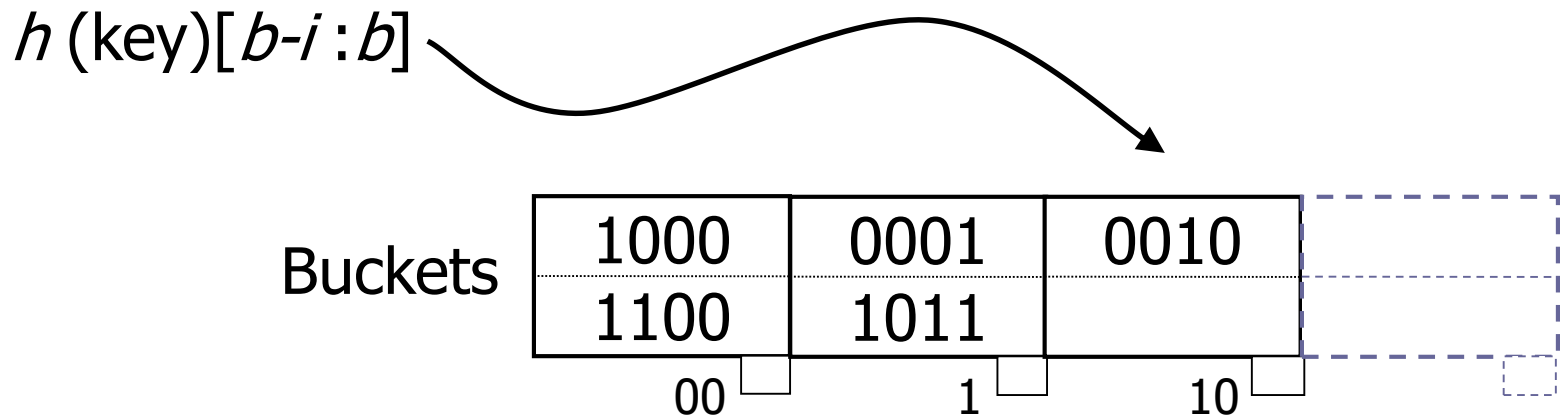
## ■ Disadvantages

- Indirection
  - Not bad if directory in memory
- Directory doubles
  - May not fit in memory
  - Number of buckets grows linearly

# Linear Hashing

## ■ Idea:

- Use  $i$  low order bits of address (of  $b$  bits)
- No directory
- File grows linearly





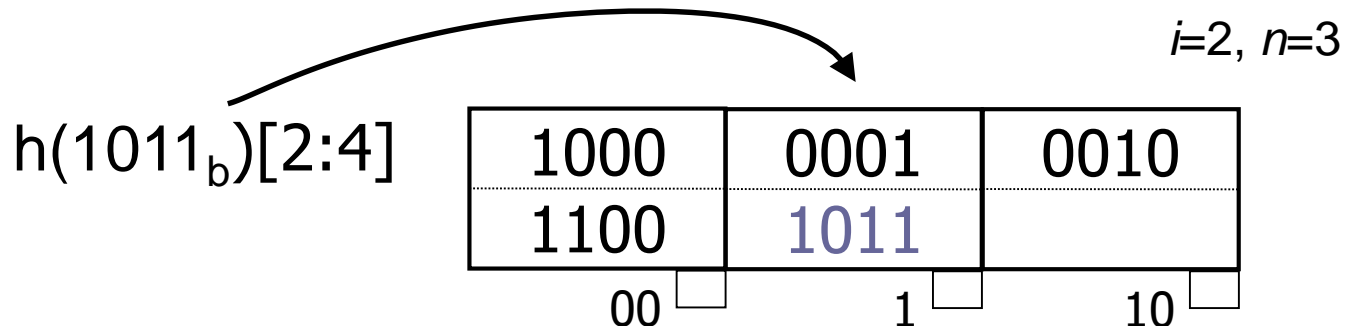
# Linear Hashing: Insertion

## ■ Parameters:

- $i$  – suffix length  $\Rightarrow h(k)[b-i:b]$
- $n$  – number of allocated buckets

## ■ Insert $1011_b$

- $h(1011_b)[2:4] = 11_b = m$ 
  - If  $m < n$ , insert into bucket  $m$
  - Otherwise, insert into bucket  $m - 2^{i-1}$

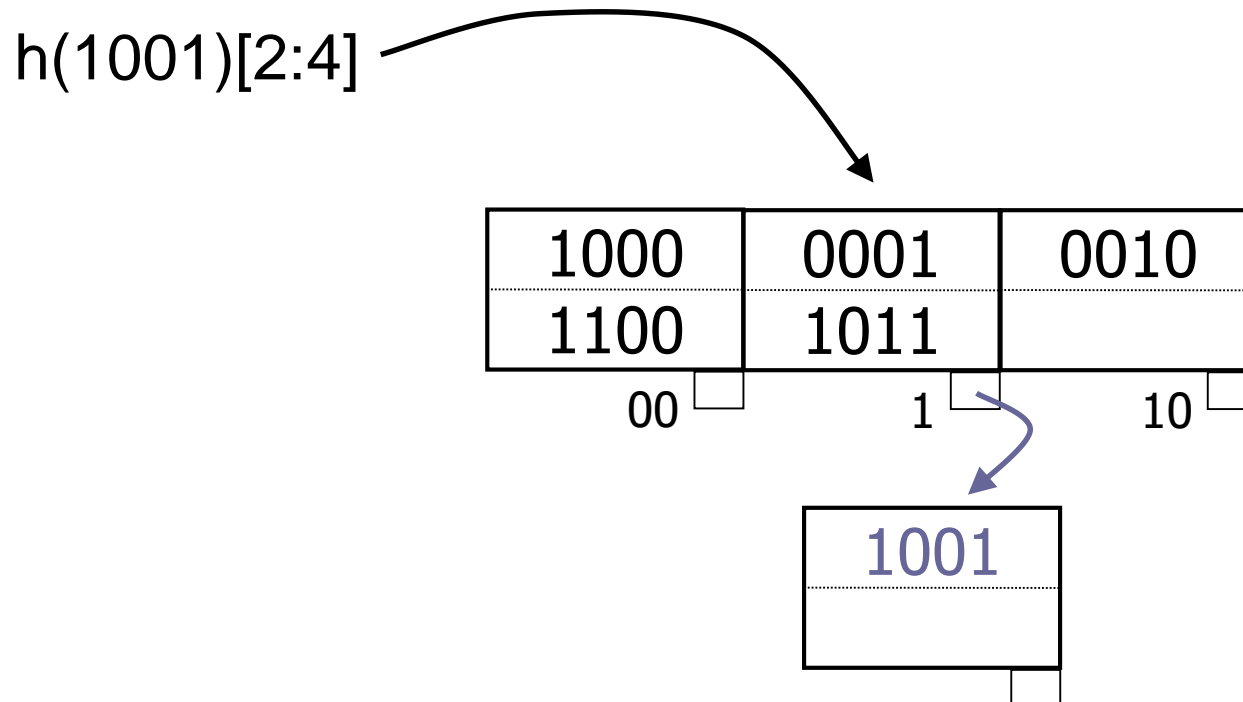


# Linear Hashing: Insertion

## ■ Insert 1001

□  $h(1001)[2:4] = 01$

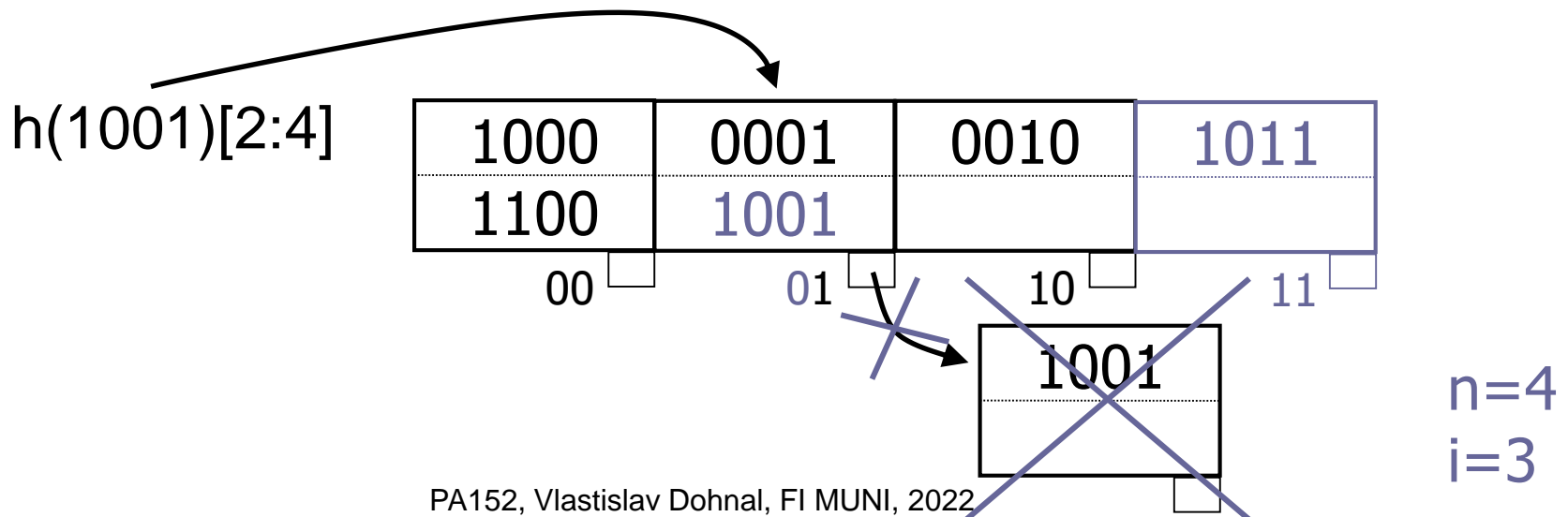
□ No room → create overflow bucket



# Linear Hashing: Bucket Splitting

## ■ Keep track of bucket utilization

- $>80\%$ , then split a bucket (its address is  $0abcd\dots z$ )
  - Add new bucket  $\rightarrow$  address is  $1abcd\dots z$
  - Distribute keys from bucket  $0abcd\dots z$  and its overflow to buckets  $[01]abcd\dots z$
- This address  $(n - 2^{i-1})$  is always split, i.e.,  $3 - 2^1 = 1$



# Linear Hashing

## ■ Inserting new keys

- May lead to overflow buckets

- Utilization exceeding 80%

  - Do the bucket split

  - The bucket being split may differ from the overflowing one!

- After allocating  $2^i$ -th bucket, increment  $i$

  - i.e., the bucket count exceeds  $2^i - 1$

# Linear Hashing: Summary

## ■ Advantages

- Handles growing files
- Less wasted space (than static hashing)
- Local reorganization
- No indirection like extendible hashing

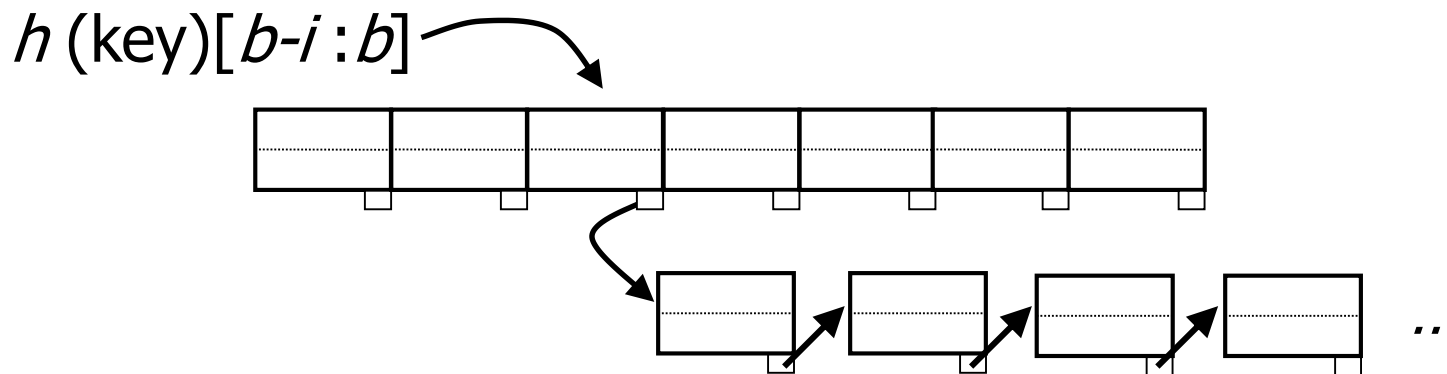
## ■ Disadvantages

- Overflow chains

# Linear Hashing: Example

- „Bad“ data

- Last  $x$  bits do not distinguish keys



- One bucket “stores” all data; others are empty
  - Utilization is decreased by split
    - → need to track the length of overflow blocks!

# Hashing vs. Indexing

## ■ Hashing

- Good for exact match

```
SELECT ... WHERE a=5
```

## ■ Indexing

- Good for range search

```
SELECT ... WHERE a>5
```

## ■ What if cardinality of ``a'' is low?

# Bitmap (raster) index

- Attribute  $X$  has got  $h$  unique values
- Collection of  $h$  bit arrays
  - One array (vector) per value
  - Array length is equal to # records ( $n$ )
  - = inverted file
- Good for attributes with „few“ values
  - Up to 10% of records
- Queries:
  - WHERE  $a = 5$  OR  $a = 7$



# Bitmap index: Example

- Relation  $R(F, G)$

$F$	$G$
30	foo
30	bar
40	baz
50	foo
40	bar
30	baz
34	foo

- Bitmap index on  $G$

<i>Value</i>	<i>Vector</i>
foo	1001001
bar	0100100
baz	0010010

# Bitmap index: Summary

## ■ Disadvantages

### □ Storage demands

- If *key* is the primary key, then

$$n(n + \log_2 n) \text{ bits}$$

- Many records with the same value (many 1's)  
or too many records in general

- Compress to the bit vector of blocks (instead of records)

### □ Record updates

- New value → add new array
- New record → extend all arrays

# Bitmap index: Summary

## ■ Advantages

- Fast bit operations (AND, OR)
- Applicable to ``range'' queries
- Easy to combine multiple indexes together
  - which are typically single-attribute indexes

# Bitmap index: Compression

- Reduce individual arrays
  - Few 1's, many 0's
  - ends with one
  - trailing zeros are omitted
    - Can be filled up to # records
- Run-Length Encoding (RLE)
  - Split to blocks
    - Sequence of "*i*" zeros followed by *one*
  - Encode "*i*" binary
  - Code for "*i*" :  
„Binary number length + number itself“

# Bitmap index: RLE

- Example of compression – 24 bits
  - Sequence: 0000 0000 0000 0110 0010 0000
    - 13x zeros, 1x one
    - 0x zero, 1x one
    - 3x zero, 1x one
    - 5x zero followed by nothing... → ignore
  - Binary:
    - $13_d \rightarrow 1101_b$
    - $0_d \rightarrow 0_b$
    - $3_d \rightarrow 11_b$
  - RLE code:
    - Sequence of „blocks“:
      - Binary number length in prefix code, binary number itself
    - Code: 11101101001011

# Bitmap index: RLE

- Example of decompression
  - Code 11101101001011
  - Split into block and decompress...
    - Binary number length:
      - bit count (ones followed by one zero)
    - 11101101001011
  - Decoding parts
    - 11101101 → 0000 0000 0000 01
    - 00 → 1
    - 1011 → 0001
  - Resulting sequence:
    - 0000 0000 0000 0110 001
    - Missing trailing zeros filled up to # records

# Bitmap index: Operations

- Bit operations

- AND, OR

- RLE strings

- Decompression is easy

- No need to decompress

- More complex implementation, but possible

- AND: sum of “*i*”s in codes must match

- OR: by analogy...

- 11101101001011 OR/AND 1101011101001010

# Bitmap index: Implementation

## ■ Issues for efficient use:

1. Locate bit arrays for the passed key value
2. Having the array, how to get records?
3. Updating records, how to update index?



# Bitmap index: Solutions

- Ad 1: (Locate bit arrays for the passed key value)
  - We have a bit array for each key value
    - B<sup>+</sup>-tree for keys
    - Pointers to arrays in leaves
  - Storing bit arrays
    - records of variable length
- Ad 2: (Having the array, how to get records?)
  - Get record  $r$  (ordinal number of record)
    - Secondary index for record numbers
    - Array of pointers to records (replacement of bit arrays)
    - List of block occupations in the number of records

# Bitmap index: Solutions

- Ad 3: (Updating records, how to update index?)
  - Record numbers fixed (not seq. file / sec. index)
    - Delete record
      - tombstone in file and
      - clear a bit in the corresponding array
    - Insert record
      - append to end of file (a new record number)
      - append bit 1 to the corresponding bit array
        - if not existing, create new one

# Bitmap index: Solutions

- Ad 3: (Updating records, how to update index?)
  - Record numbers are not fixed
    - Reorganize all arrays (delete 1 bit)
    - Update array of pointers to records
  - Rarely used

# Bitmaps – data (TPC-H)

## Settings:

```
lineitem ( L ORDERKEY, L PARTKEY , L SUPPKEY, L LINENUMBER,  
L_QUANTITY, L_EXTENDEDPRICE ,  
L_DISCOUNT, L_TAX , L_RETURNFLAG, L_LINESTATUS ,  
L_SHIPDATE, L_COMMITDATE,  
L_RECEIPTDATE, L_SHIPINSTRUCT ,  
L_SHIPMODE , L_COMMENT );
```

```
create bitmap index b_lin_2 on lineitem(l_returnflag);
```

```
create bitmap index b_lin_3 on lineitem(l_linestatus);
```

```
create bitmap index b_lin_4 on lineitem(l_linenum);
```

- 100000 rows ; cold buffer
- Dual Pentium II (450MHz, 512KB), 512 MB RAM, 3x18GB drives (10000RPM), Windows 2000.

# Bitmaps -- queries

## Queries:

### □ 1 attribute

```
select count(*) from lineitem where l_returnflag = 'N';
```

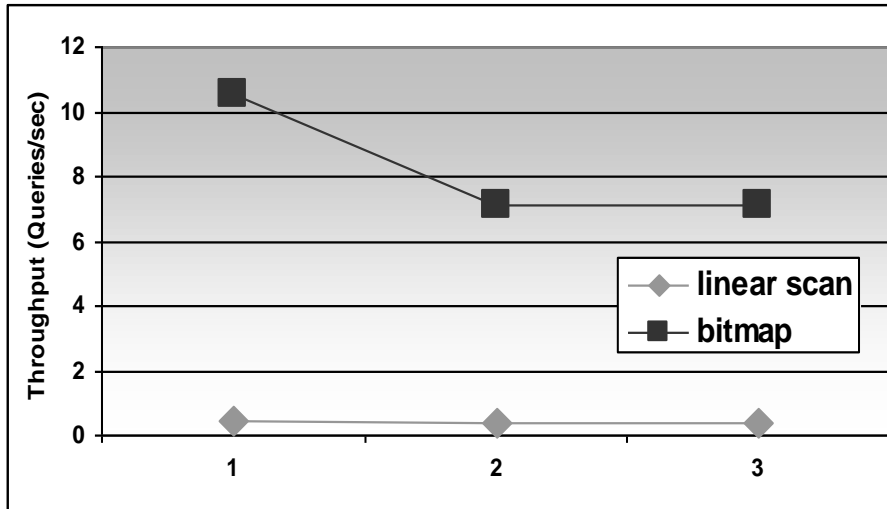
### □ 2 attributes

```
select count(*) from lineitem where l_returnflag = 'N' and  
l_linenumber > 3;
```

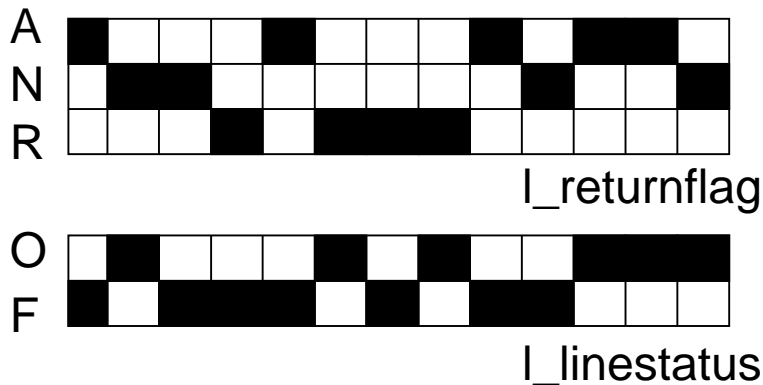
### □ 3 attributes

```
select count(*) from lineitem where l_returnflag =  
'N' and l_linenumber > 3 and l_linestatus = 'F';
```

# Bitmaps



- Order of magnitude improvement compared to table scan.
- Bitmaps are best suited for multiple conditions on several attributes, each having a low selectivity.

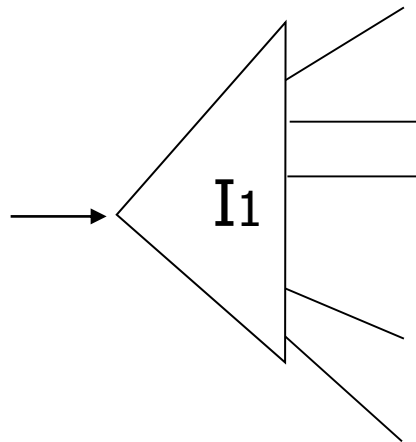


# Multi-key / Composite index

- Index on multiple attributes
- Motivation:  
SELECT name, salary FROM emp  
WHERE dept='Toys' AND salary < 10000
- Alternatives:
  - a) Index on one attribute + filtering
  - b) Combine two indexes + intersection of candidate record pointers
  - c) Index in index
  - d) Concatenation of key values into one

# Index on one attribute

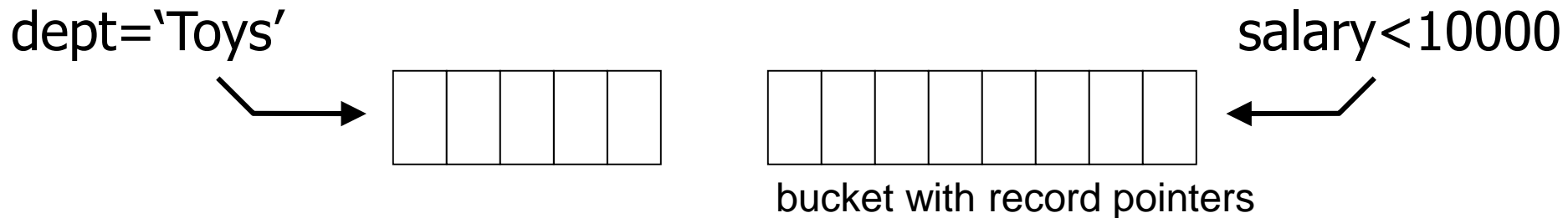
- SELECT name, salary FROM emp  
WHERE dept='Toys' AND salary < 10000
- Index on *dept*
  - Filter found records using *salary* < 10000





# Combining Indexes

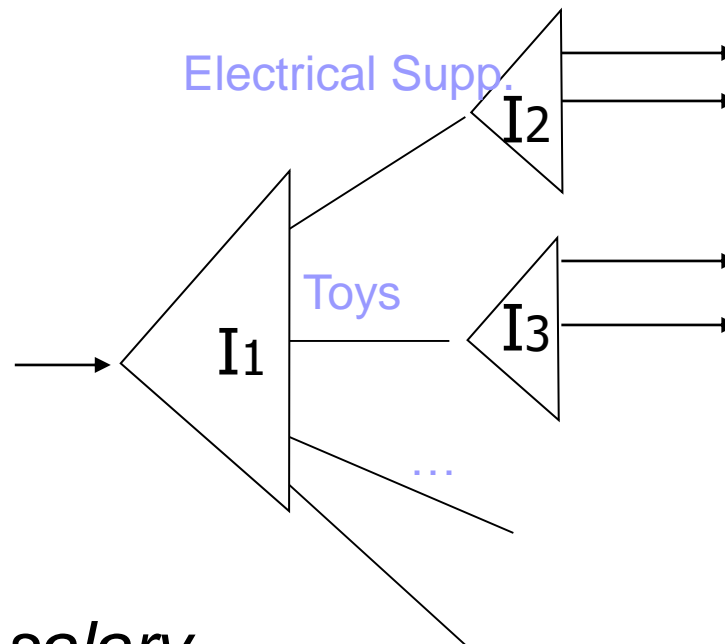
- Index on *dept*
- Index on *salary*
- Each index returns a list of candidate records
  - Intersection of lists → query result



# Index in index

- Index on the first attribute

- In leaves, pointers to embedded indexes on 2<sup>nd</sup> attribute



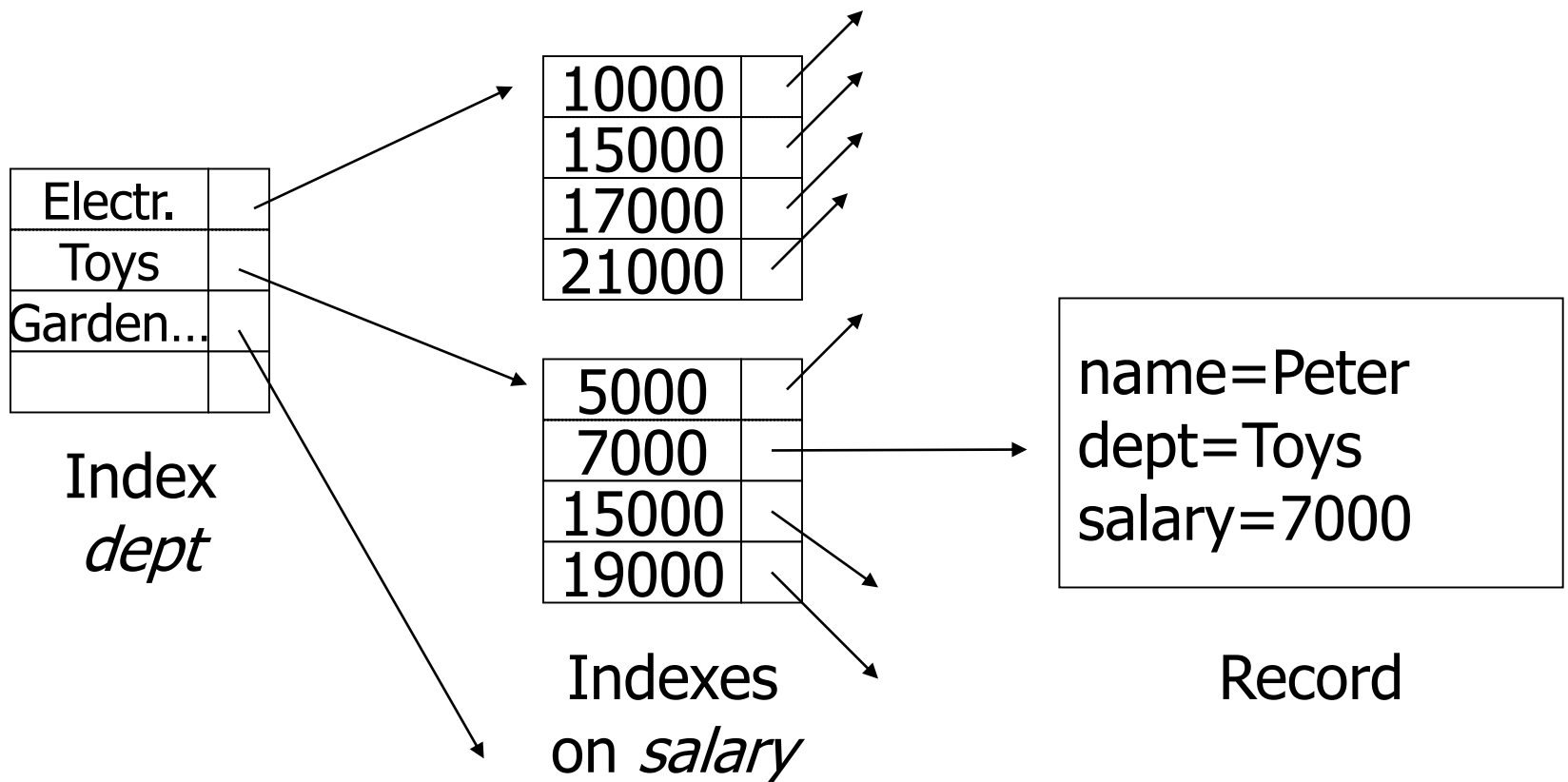
- $I_1$  on *dept*

- $I_x$ ,  $x=2..k$  on *salary*

- $I_2$  contains only records having the same dept. value (Electrical Supp.)

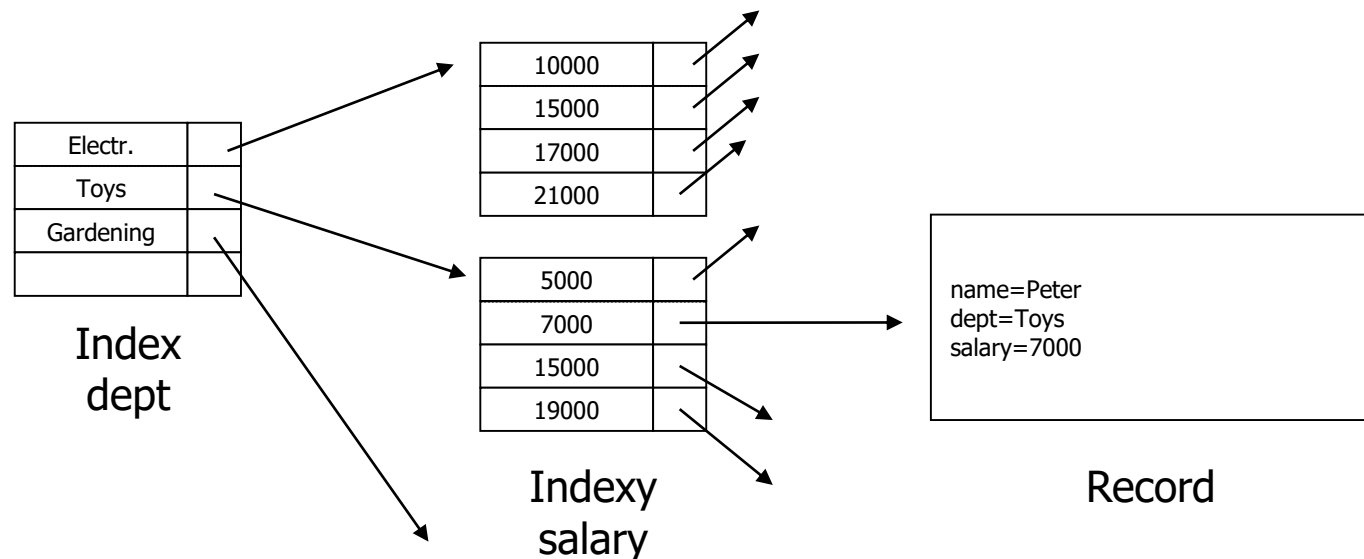
# Index in index: Example

- `SELECT name, salary FROM emp WHERE dept='Toys' AND salary < 10000`



# Index in index

- What queries can use this solution?
  - SELECT name, salary FROM emp WHERE
    - a) dept = 'Toys' AND salary  $\geq$  10000
    - b) dept = 'Toys'
    - c) salary = 10000

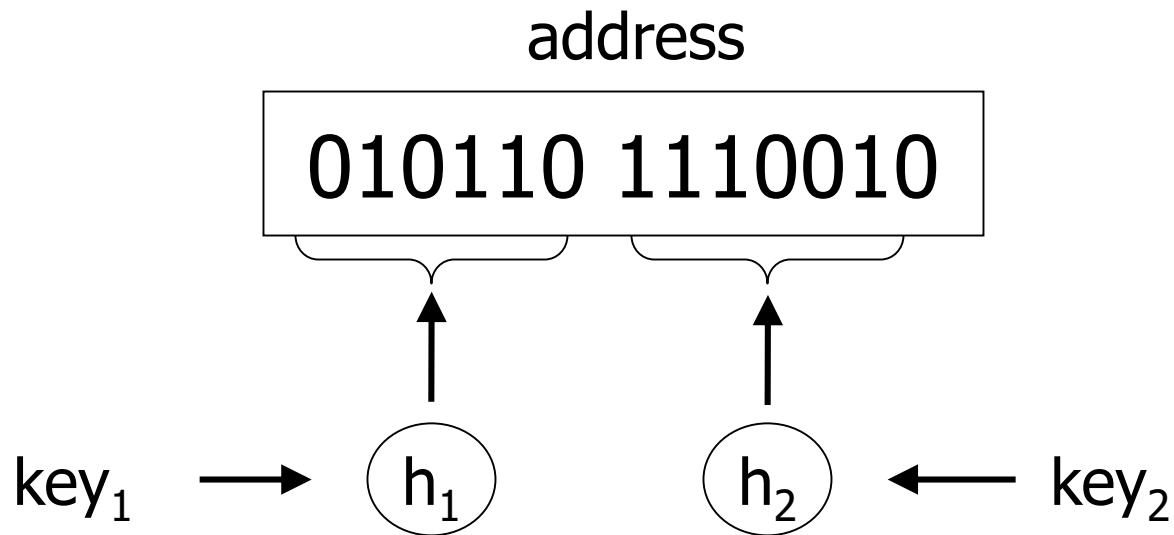


# Concatenation of key values

- Analogous to an index on one attribute
  - Key values are merged
    - String concatenation, number multiplication, ...
- Indexing
  - rarely used unless (string) keys are compressed
- Hashing
  - Partitioned hash function

# Partitioned hash function

- Idea:
  - Two keys
  - Two hash functions
  - One address



# Partitioned hash function: Example

## ■ Dept

$h_1(\text{Electr.})=0$

$h_1(\text{Toys}) =1$

$h_1(\text{Gardening}) =1$

## ■ Salary

$h_2(10000) =01$

$h_2(20000) =11$

$h_2(30000) =10$

$h_2(40000) =00$

## ■ Records to insert

<Peter,Electr.,10000>

<Jan,Toys,10000>

<Alice,Gardening,30000>

000	
001	<Peter,Electr.,10000>
010	
011	
100	
101	<Jan,Toys,10000>
110	<Alice,Gardening,30000>
111	

# Partitioned hash function: Example

## ■ Dept

$h_1(\text{Electr.})=0$

$h_1(\text{Toys}) = 1$

$h_1(\text{Gardening}) = 1$

## ■ Salary

$h_2(10000) = 01$

$h_2(20000) = 11$

$h_2(30000) = 10$

$h_2(40000) = 00$

000	<Pavel,...> <Lukáš,...>
001	<Peter,...>
010	<Marie,...>
011	
100	<Anna,...>
101	<Jan,...>
110	<Alice,...>
111	<Veronika,...>

## ■ Find

employess in Toys dept. with salary of 40000.



# Partitioned hash function: Example

## ■ Dept

$h_1(\text{Electr.})=0$

$h_1(\text{Toys}) =1$

$h_1(\text{Gardening}) =1$

## ■ Salary

$h_2(10000) =01$

$h_2(20000) =11$

$h_2(30000) =10$

$h_2(40000) =00$

000	<Pavel,...>	<Lukáš,...>
001	<Peter,...>	
010	<Marie,...>	
011		
100	<Anna,...>	
101	<Jan,...>	
110	<Alice,...>	
111	<Veronika,...>	

## ■ Find

employees with salary 30000

# Partitioned hash function: Example

## ■ Dept

$h_1(\text{Electr.})=0$

$h_1(\text{Toys}) = 1$

$h_1(\text{Gardening}) = 1$

## ■ Salary

$h_2(10000) = 01$

$h_2(20000) = 11$

$h_2(30000) = 10$

$h_2(40000) = 00$

## ■ Find

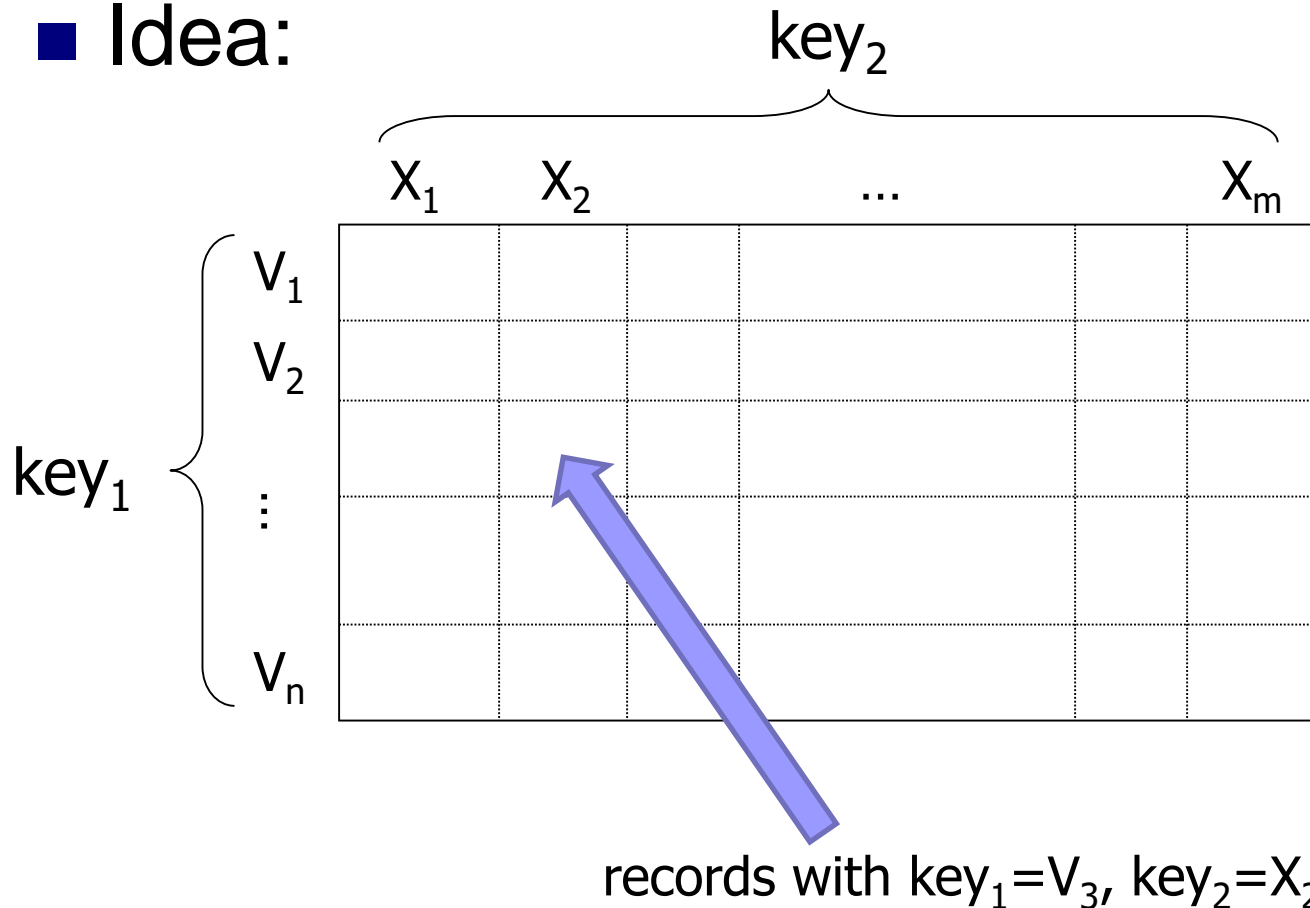
employes in Toys dept.

000	<Pavel,...> <Lukáš,...>
001	<Peter,...>
010	<Marie,...>
011	
100	<Anna,...>
101	<Jan,...>
110	<Alice,...>
111	<Veronika,...>

# Another Multi-key Index

- Grid index

- Idea:



# Grid Index: Properties

- Efficient for exact match queries

- $\text{key}_1 = V_i$  and  $\text{key}_2 = X_j$

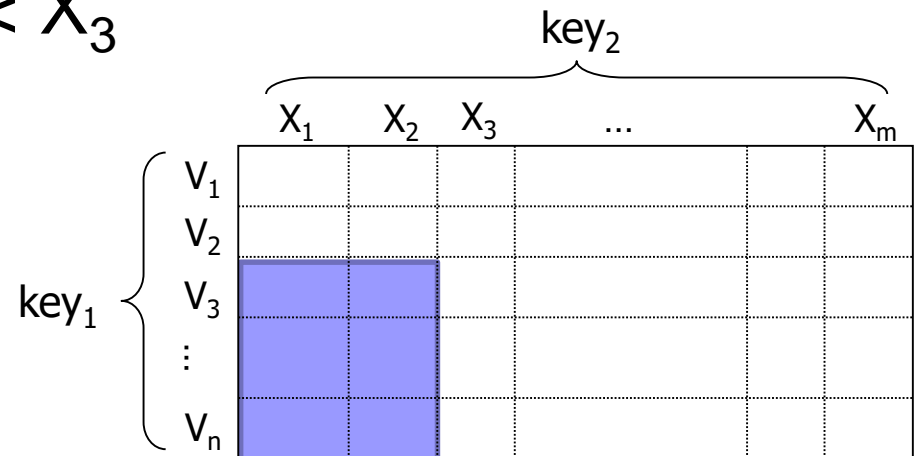
- $\text{key}_1 = V_i$

- $\text{key}_2 = X_j$

- Range queries

- $\text{key}_1 \geq V_3$  and  $\text{key}_2 < X_3$

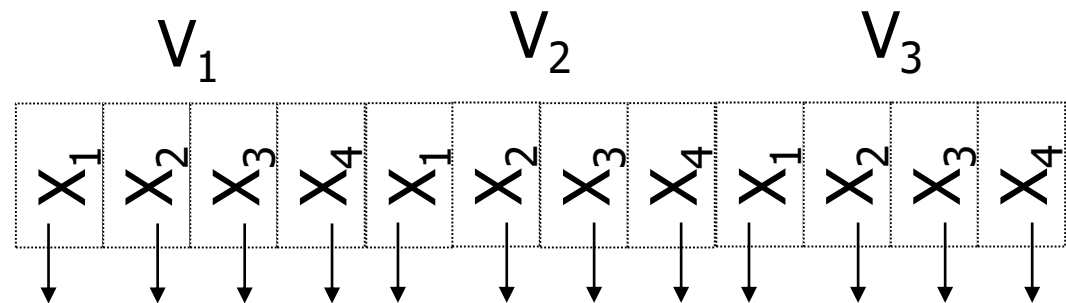
- Rectangular area



# Grid Index: Implementation

- How to store grid on disk?

- one-dimensional array



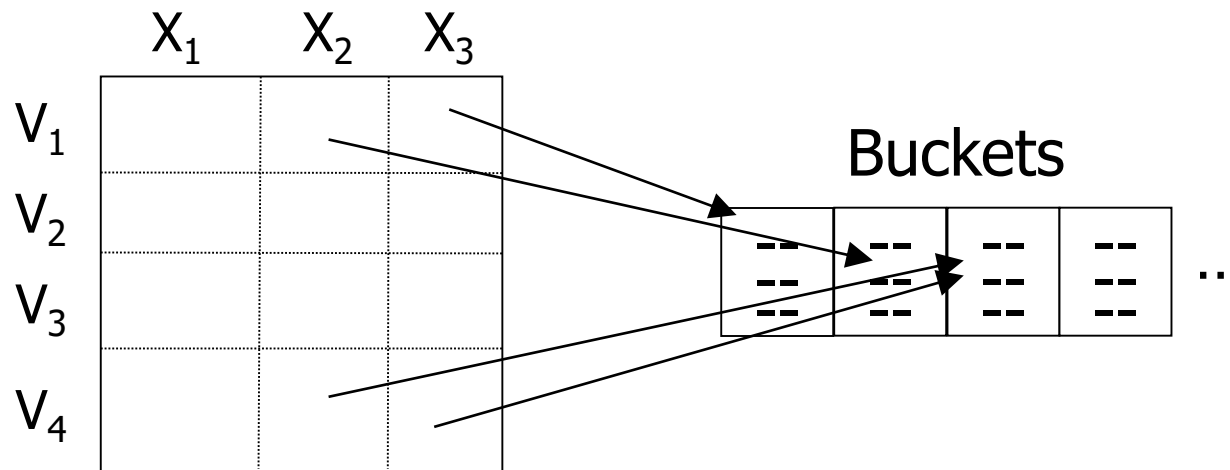
- Tradeoff: grid dimensions vs. cell capacity

- Disadvantage

- Fixed grid dimensions needed to calculate position of cell  $\langle V_x, X_y \rangle$ .
  - Limited cell capacity

# Grid Index: Implementation

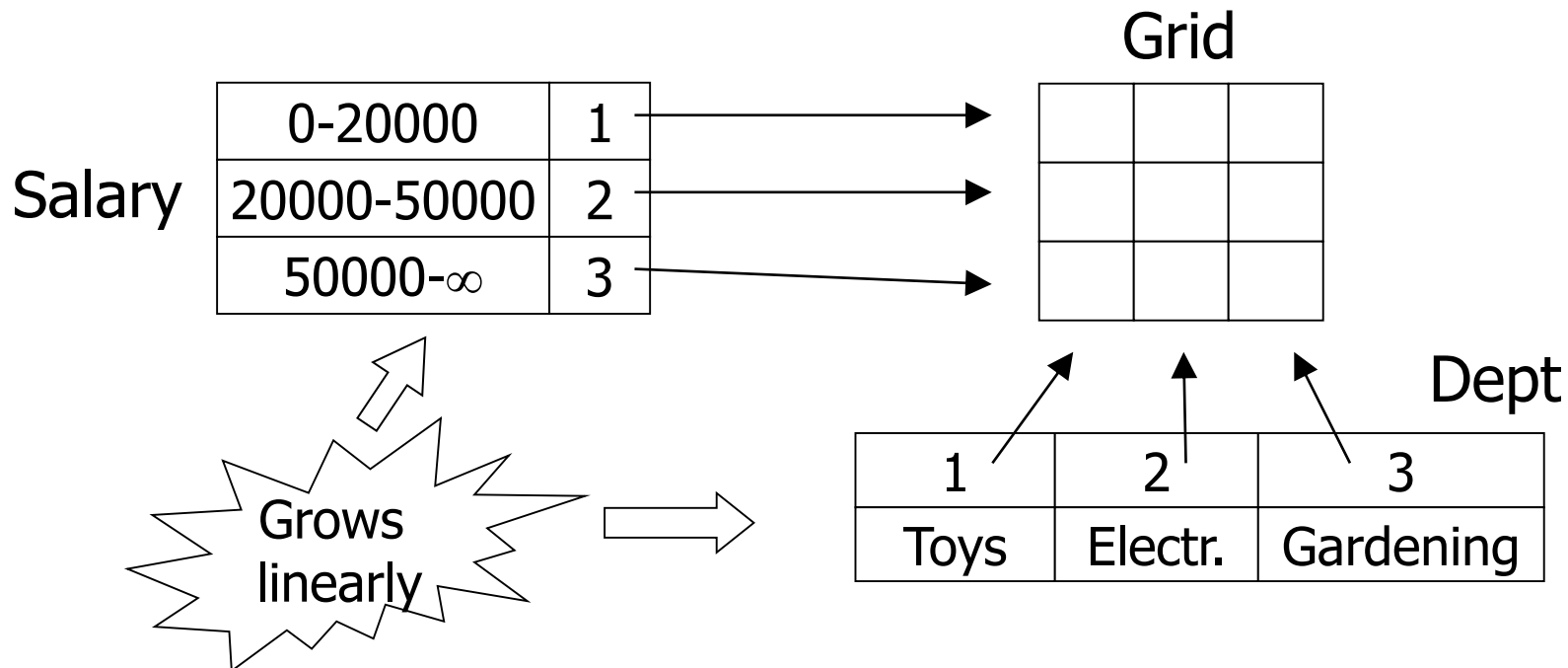
- Use buckets, i.e., indirection
  - Grid cell points to a bucket



- Grid is fixed-sized
  - Overhead with indirection
  - Allows allocating buckets linearly.

# Grid Index: Defining Grid

- Analysis of data and query types
  - Find out dimensions of the grid
  - Value on an axis can be an interval → binning
    - E.g., numeric domains



# Grid Index: Summary

## ■ Advantages

- Good for multikey indexing

## ■ Disadvantages

- Grid size is fixed; may waste space
  - Alternative is a hierarchical grid
- Choice of grid intervals (quantization)  
→ uniform data distribution



# Index Design Guidelines

- Good selectivity (i.e., low selectivity)
- “Short” attribute values
  - Increase tree fan-out
- On “join” attributes
- Prefer more single-attribute indexes for one multi-attribute one
- Few indexes for highly-updated tables
- No indexes for tiny tables

# Lecture's Takeaways

- Techniques of hashing
- Recall handling duplicate keys
  - i.e. multiple records with the same key value.
- Alternative indexes
  - Bitmap index
    - do not mix up with bitmap scan in Pg's explain plan
  - Grid index
- Further terms (follows...)

# Further Terms

- Clustered index
  - = index-sequential file / B<sup>+</sup>-file
  - Records are stored in leaf nodes and cannot be accessed in any other way (than by this index)
- Non-clustered index
  - = secondary index / B<sup>+</sup>-tree
- Covering index
  - Query can be fully answered using the index only
    - I.e. records are not accessed.
  - MS SQL Server: *Index with included columns*
    - Not a multi-attribute index, but leaf node values are enriched with values of the included cols.
- Indexed view
  - materialized view with clustered index
- Multidim. data and indexes (R-trees, Quad Trees, kd-trees)