Information Systems

WS 2005, JKU Linz
Course 5: Indexing and Hashing

Gábor Bodnár

URL: `http://www.risc.uni-linz.ac.at/education/courses/ws2005/is/`

Overview

- Indices (dense and sparse).
- B-trees.
- Hashes (extendible).

What are they good for?

Indices and hashes are files with special structure, helping the DBMS to navigate faster in the database.

Goals:

- To accelerate data retrieval from the database.
- To achieve as little administrative overhead as possible at database modifications.

To emphasize that in this discussion we consider the database on the physical level, we will talk about *files* and *records* instead of tables and rows.

Generalities on Indices and Hashes

The set of attributes on which an index (or hash) file is created is called the *search key* for the index (or hash).

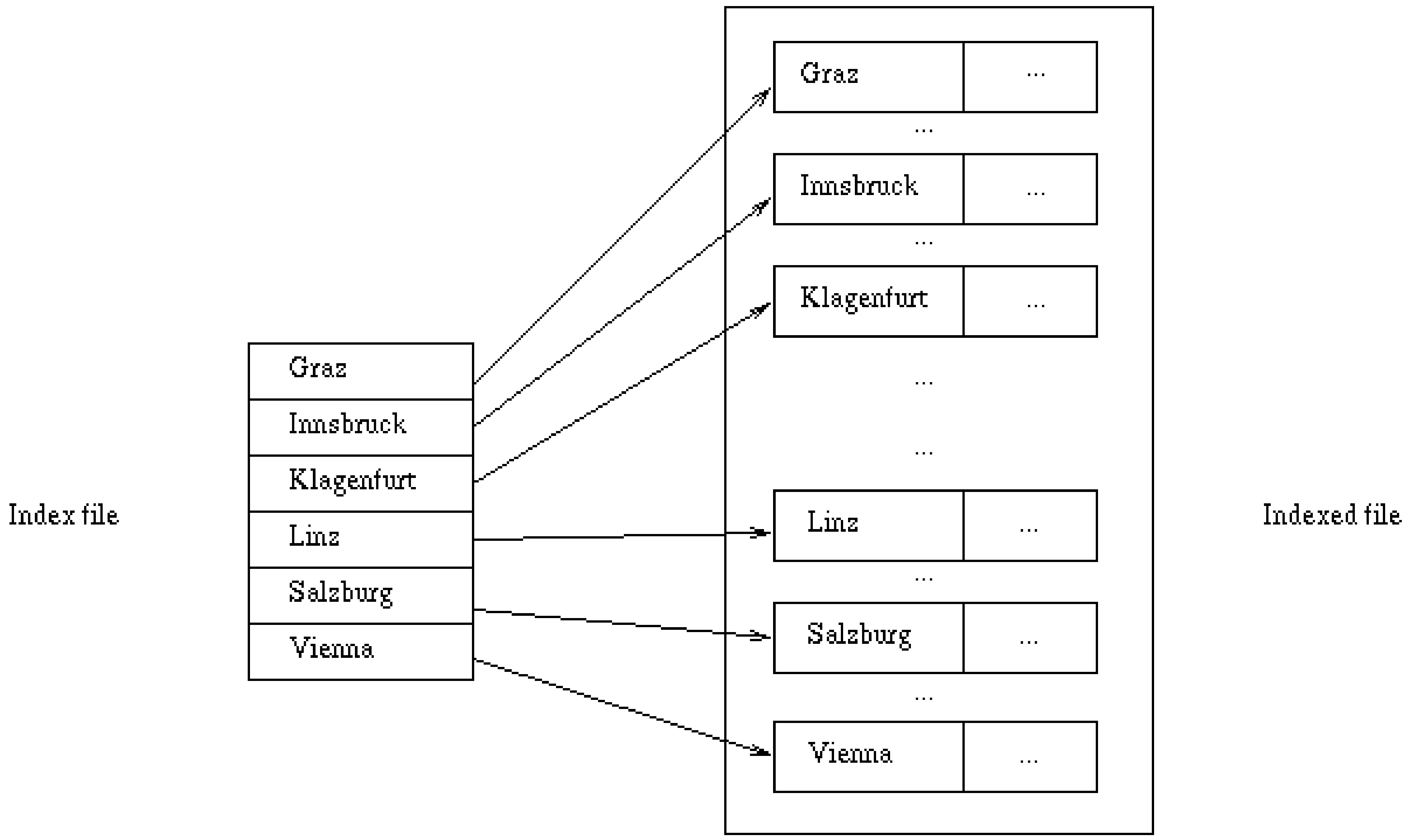
Indices work like the catalog in the library, containing search key values together with pointers to the corresponding records in the indexed file.

Indices are useful to accelerate queries involving attributes with ordered domains.

There are several aspects to evaluate indexing methods (access/insert/delete times, space overhead) and there is not a single best performer.

Relations of the database can have multiple indices and hashes assigned to them.

Example


Index file
Graz
Innsbruck
Klagenfurt
Linz
Salzburg
Vienna
Indexed file
Graz
...
Innsbruck
...
Klagenfurt
...
Linz
...
Salzburg
...
Vienna
...

Primary and Secondary Indices

Given a file of the database whose records are ordered sequentially according to a set of attributes.

An index whose search key is such a set of attributes is called a *primary index*.

Such a search key is usually (but not necessarily) the primary key of the relation stored in the file.

Indices that define not the sequential order of the file are called *secondary indices*.

Dense and Sparse Indices

An index is called *dense* if an index record appears (in the index file) for every search key value in the indexed file.

Dense indices allow fast data retrieval but they can require large amount of disk space.

When only some of the search key values appear in the index file, the index is called *sparse*.

Primary indices can be sparse.

In general it requires first searching the index file and then searching the data file to locate the requested record.

Secondary indices have to be dense.

Example

Dense primary index

Search key: `sid`	Pointer
0245654	1
0251563	2
0252375	4
0252483	5

Dense secondary index

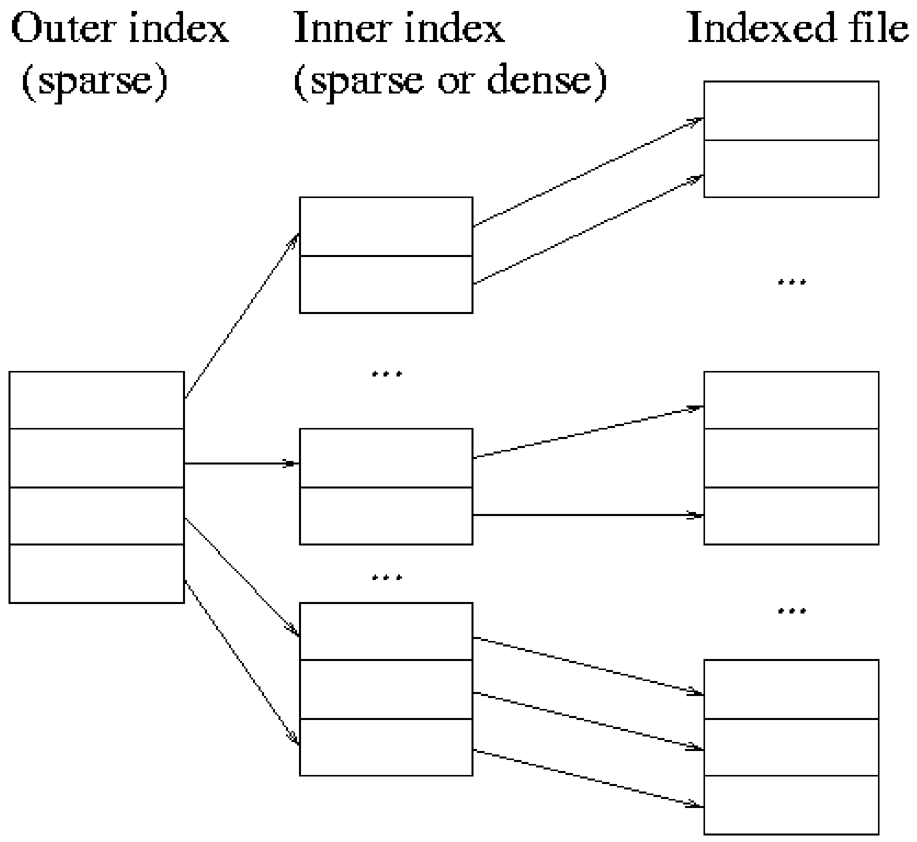
Search key: {`sid`,`cid`}		Pointer
0245654	327456	1
0251563	327456	3
0251563	327564	2
0252375	327456	4
0252483	327564	5

Indexed file (sorted by the attribute `sid`)

Address	`sid`	`fname`	`lname`	`cid`	`title`
1	0245654	Andrea	Ritter	327456	Analysis I
2	0251563	Werner	Schmidt	327564	Algebra I
3	0251563	Werner	Schmidt	327456	Analysis I
4	0252375	Stefan	Braun	327456	Analysis I
5	0252483	Franz	Lander	327564	Algebra I

Index Organization

For large indexed files one can apply multi-level indices.



Index Organization

If the search key is not a candidate key in the indexed file, pointer buckets can also be realized by the inner level index of a two level indexing scheme.

We search for the given value on the outer level, which points to the appropriate part of the inner level that stores pointers to the records of the data file with the given search key values.

It is also advantageous to fit indices with physical storage units, for instance to organize the multi-level index by

disks $\rightarrow$ cylinders $\rightarrow$ blocks.

Indices and Database Modifications

If record is deleted from the indexed file, we have to check for each index file (on this data file) whether the deleted record was the last with the given search key value.

If yes the index record for that value has to be deleted from the index file.

For sparse indices deleting a search key value can be substituted with updating it to the next larger value in the data file, if it is not already in the index file.

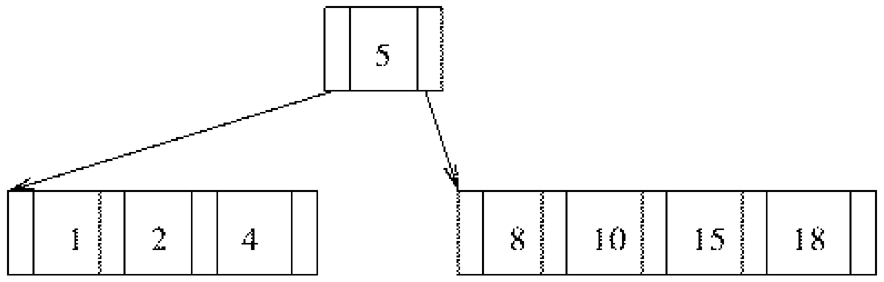
Insertion implies the opposite actions as deletion.

B-trees

The most popular data structures for indexing are B- and B^+-trees.

The most important feature of B-trees is that they are always balanced, hence the depth of the tree grows only logarithmically in the growth of the indexed records.

B-trees contain two kinds of data: search key values (with pointer buckets to the indexed file) and pointers to subtrees. One node contains many key values and pointers.



Rules for B-trees

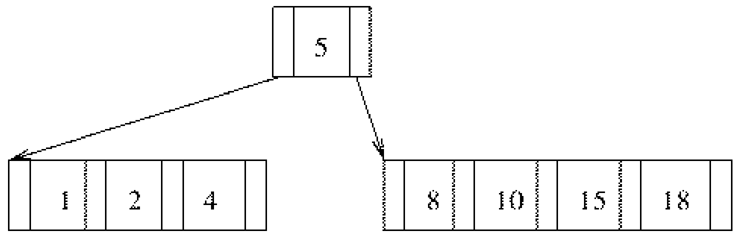
To construct a B-tree we fix an even integer n and require that

- the root stores minimum 1 and maximum n key values,
- each node of the tree, with the exception of the root, stores minimum $n/2$ and maximum n key values,
- leaf nodes are on the same level.

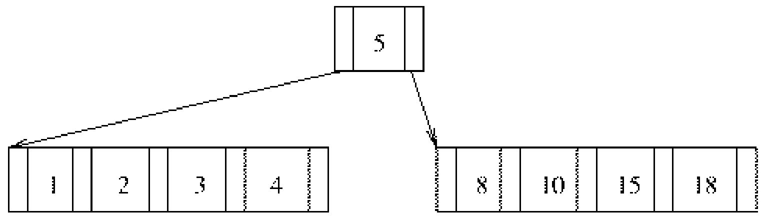
The subtrees have the property that each value stored in them lie between the two values neighboring the pointer that links the subtree to the node.

Insertion into B-trees (Example)

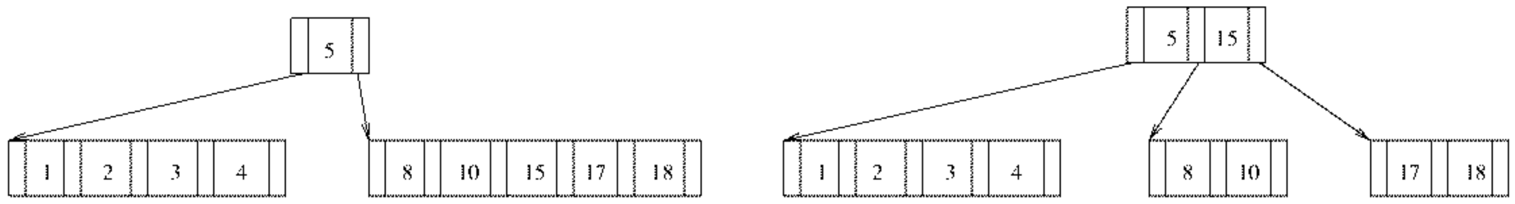
We have chosen $n = 4$.



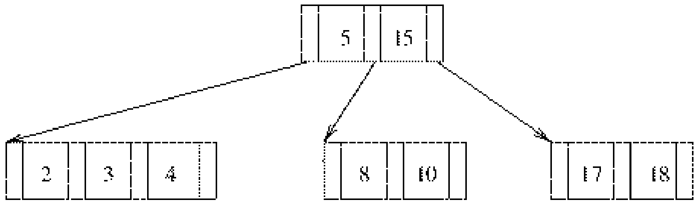
Inserting 3

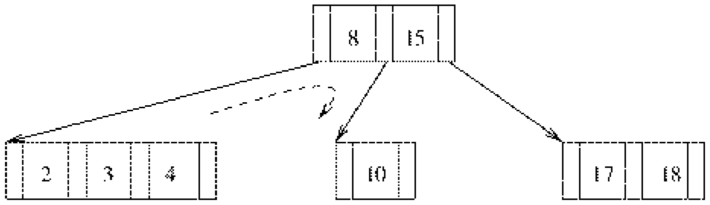


Inserting 17

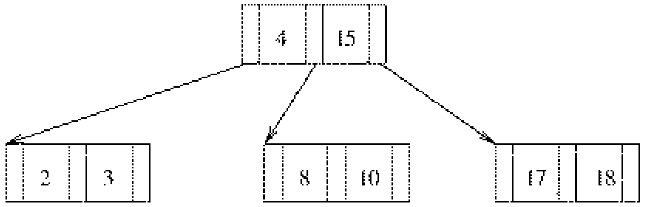


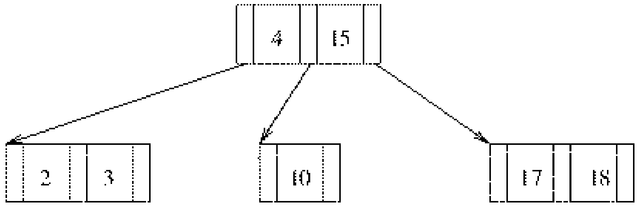
Deletion from B-trees (Example)



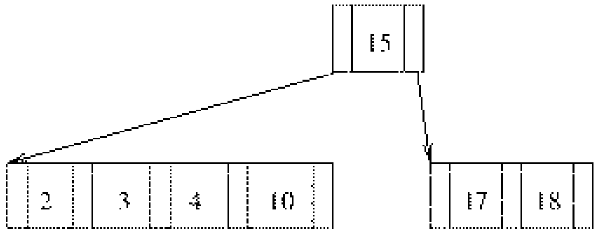


Deleting 5 and pulling over elements from the left sibling via the parent





Deleting 8 and unifying 4 with the first two siblings



Hashes

In hashing we prescribe an address space to which search key values have to be mapped.

A *hash function* h maps from the domain of search key values to the address space we fixed.

A hash function is by no means injective; at each address we have to provide storage for a list of pointers (identifying the records whose search key values hash to this address).

Such a storage is called a *bucket*.

To search records with a given search key K, we look through the records identified by the pointers in the bucket with address $h(K)$.

Insertion and deletion is straightforward.

Good Hash Functions

A good hash function:

- should be rapidly computable,
- should distribute the search key values as uniformly as possible between the addresses.

Problems:

- The search keys can come from various domains and can have hidden patterns.
- Bucket overflows are unavoidable in practice.

Extendible Hashing

Dynamic hashing copes with the problems of bucket overflow by allowing dynamic splitting and merging of the buckets.

Extendible hashing is a specific type of dynamic hashing which results fairly low performance overhead.

The idea of extendible hashing is comparable to the one of B-trees.

The hash function h has to map search key values to b-bit unsigned binary integers (e.g. $b = 32$).

However, we use only the i initial bits of the hash values to determine the bucket address, where $1 \leq i \leq b$ initially can be set to 1.

Extendible Hashing (continued)

The bucket with index j is addressed by the first i_j bits of the first i bits of the hash values, where $i_j \leq i$.

This means that 2^{i-i_j} rows of the bucket address table points to the jth bucket (the table has 2^i rows altogether).

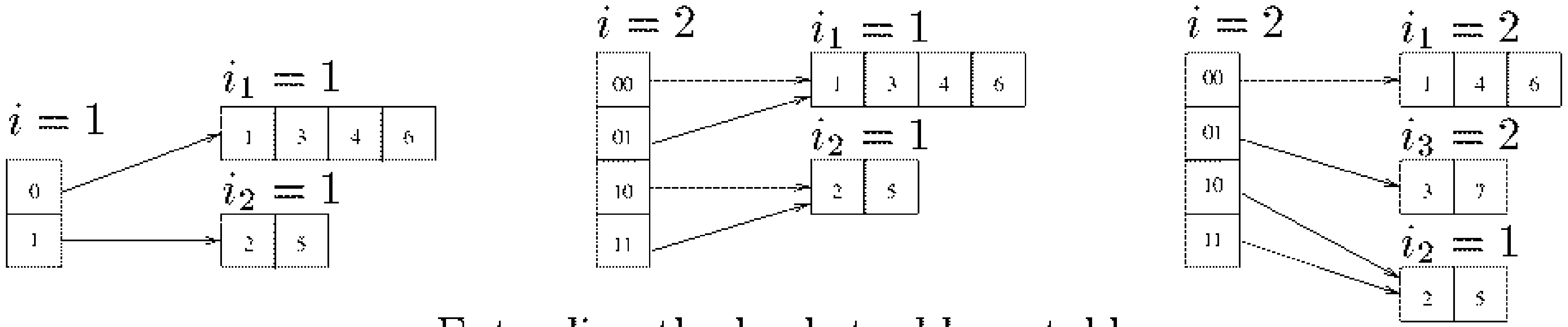
Whenever we insert a pointer to a record with a search key value K into the hashing scheme, we compute $h(K)$, take its first i bits and try to add it to the bucket whose address we obtain.

In case of an overflow, the bucket is split and the corresponding i_j value is increased.

If we had $i_j = i$, i also had to be increased.

Insertion in Extendible Hashing (Example)

We have chosen the bucket size to be 4.



Extending the bucket address table

Inserting record 7 with $h(K_7) = 01...$

Splitting up bucket 1, redistributing and inserting 7

$$h(K_1) = 00,\ h(K_2) = 10,\ h(K_3) = 01,\ h(K_4) = 00,\ h(K_5) = 11,\ h(K_6) = 00$$

Deletion implies the opposite actions (bucket merging and reducing i_js and i).

Comparison of Indexing and Hashing

Indices are useful in queries for ranges of search key values

(after the start of the range is found the index can be traversed to retrieve consecutive records).

Therefore indices provide better “worst case timing”.

Hashing, on the other hand, is more useful in accessing particular (individual) records.

Therefore hashes provide better “average timing”.

Questions to be Regarded

- If the periodic re-organization of index and hash file structures pays off.
- What is the relative frequency of insertion and deletion?
- Do we want better average access times or worst case access times?
- The type of queries we expect on the database.

Summary

- Purpose of indices and hashes
- Primary/secondary and dense/sparse indices
- Index organization
- B-trees, insertion/deletion
- Extendible hashes, insertion/deletion
- Comparison of indexing and hashing