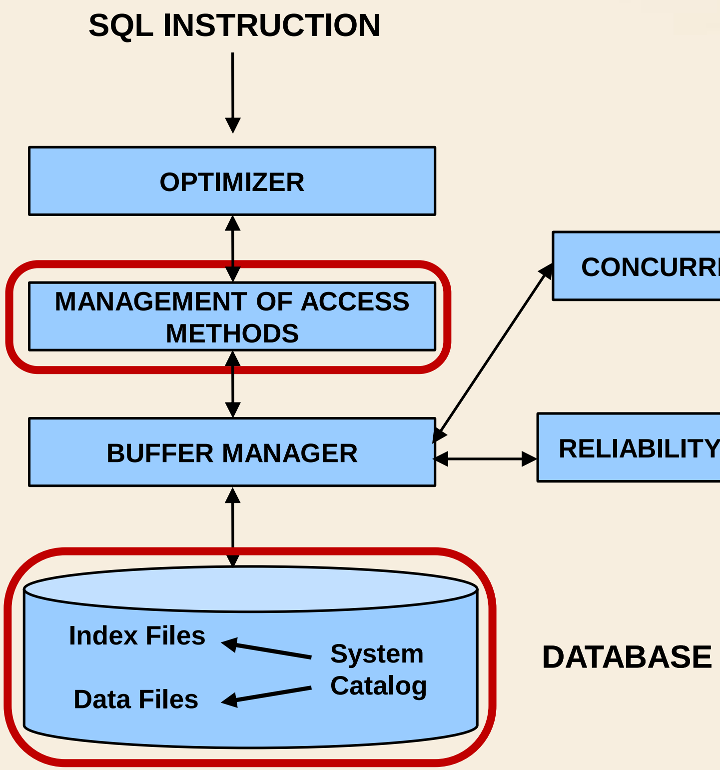
Database Management Systems

Physical Access to Data

DBMS Architecture



Physical Access Structures

- Data may be stored on disk in different formats to provide efficient query execution
  - Different formats are appropriate for different query needs
- Physical access structures describe how data is stored on disk

Access Method Manager

- Transforms an access plan generated by the optimizer into a sequence of physical access requests to (database) disk pages
  - It exploits *access methods*
- An access method is a software module
  - It is specialized for a single physical data structure
  - It provides primitives for
    - reading data
    - writing data

Access method

- Selects the appropriate blocks of a file to be loaded in memory
- Requests them to the Buffer Manager
- Knows the organization of data into a page
  - can find specific tuples and values inside a page

Organization of a disk page

- Different for different access methods
  - Divided in
    - Space available for data
    - Space reserved for access method control information
    - Space reserved for file system control information

Remarks

- Tuples may have varying size
  - Varchar types
  - Presence of Null values
- A single tuple may span several pages
  - When its size is larger than a single page
    - e.g., for BLOB or CLOB data types

Database Management Systems

Physical Access Structures

Physical Access Structures

- Physical access structures describe how data is stored on disk to provide efficient query execution
  - SQL select, update, …
- In relational systems
  - Physical data storage
    - Sequential structures
    - Hash structures
  - Indexing to increase access efficiency
    - Tree structures (B-Tree, B^+-Tree)
    - Unclustered hash index
    - Bitmap index

Sequential Structures

- Tuples are stored in a given sequential order
- Different types of structures implement different ordering criteria
- Available sequential structures
  - Heap file (entry sequenced)
  - Ordered sequential structure

Heap file

- Tuples are sequenced in *insertion order*
  - insert is typically an *append* at the end of the file
- *All* the space in a block is completely exploited before starting a new block
- Delete or update may cause wasted space
  - Tuple deletion may leave unused space
  - Updated tuple may not fit if new values have larger size
- Sequential reading/writing is very efficient
- Frequently used in relational DBMS
  - jointly with unclustered (secondary) indices to support search and sort operations

Ordered sequential structures

- The order in which tuples are written depends on the value of a given key, called *sort key*
  - A sort key may contain one or more attributes
    - the sort key may be the primary key
- Appropriate for
  - Sort and group by operations on the sort key
  - Search operations on the sort key
  - Join operations on the sort key
    - when sorting is used for join

Ordered sequential structures

- Problem
  - preserving the sort order when inserting new tuples
    - it may also hold for update
- Solution
  - Leaving a percentage of free space in each block during table creation
    - On insertion, dynamic (re)sorting in main memory of tuples into a block
- Alternative solution
  - Overflow file containing tuples which do not fit into the correct block

Ordered sequential structures

- Typically used with B$^+$-Tree clustered (primary) indices
  - the index key is the sort key
- Used by the DBMS to store intermediate operation results

Tree structures

- Provide “direct” access to data based on the value of a key field
  - The key includes one or more attributes
- It does not constrain the physical position of tuples
- The most widespread in relational DBMS

General characteristics

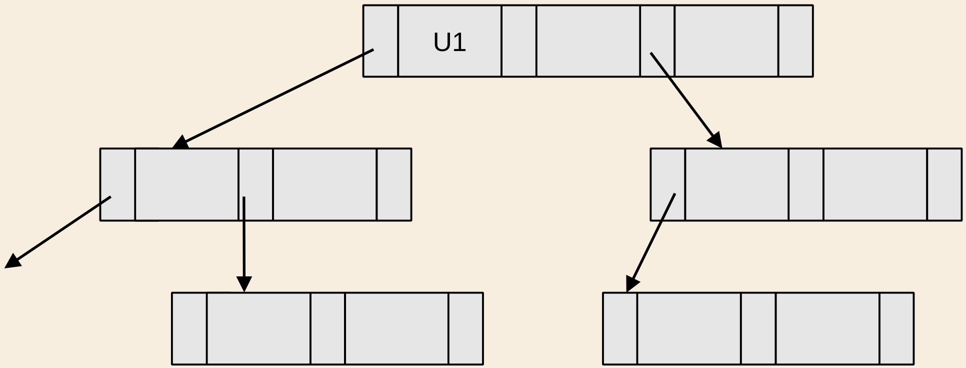
- One root node

Tree structure

General characteristics

- One root node
- Many intermediate nodes
- Nodes have a large fan-out
  - Each node has many children

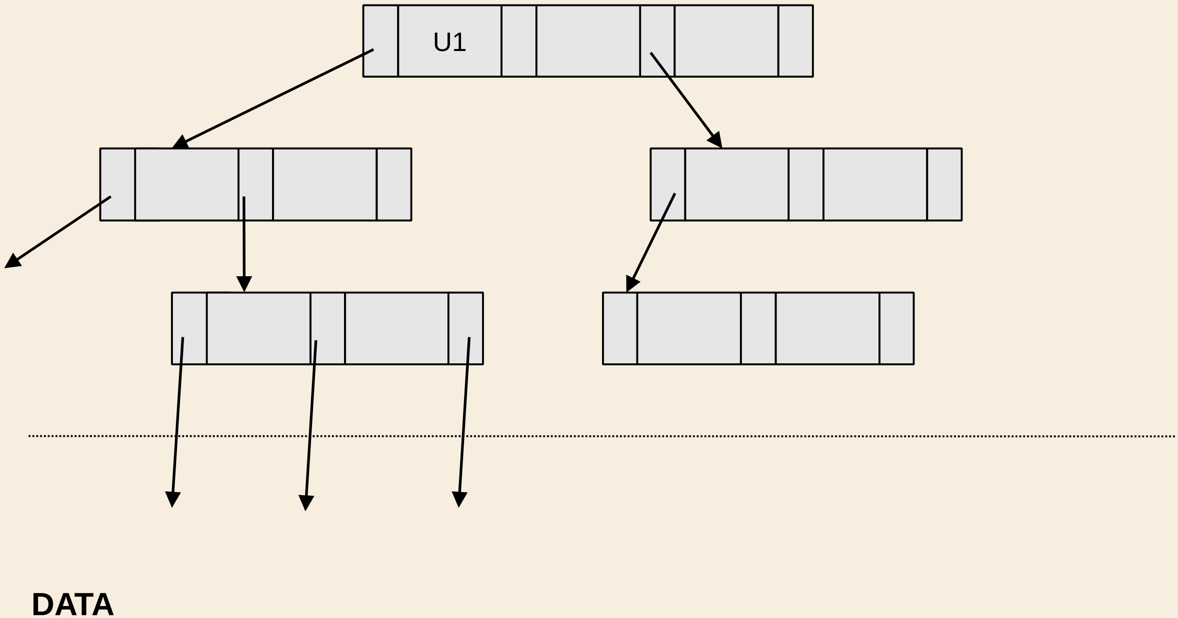
Tree structure



General characteristics

- One root node
- Many intermediate nodes
- Nodes have a large fan-out
  - Each node has many children
- Leaf nodes provide access to data
  - Clustered
  - Unclustered

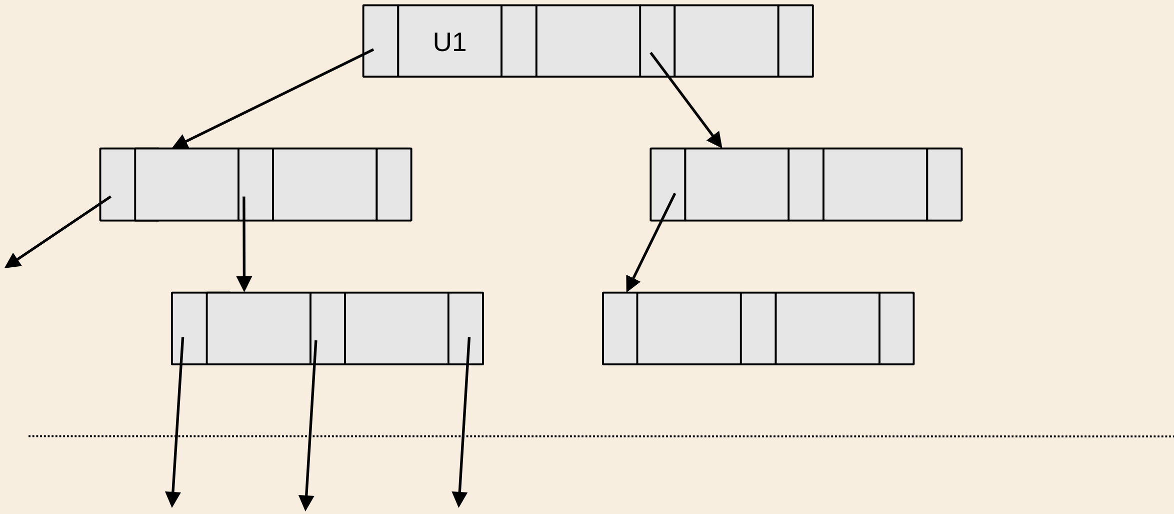
Tree structure



B-Tree and B^+-Tree

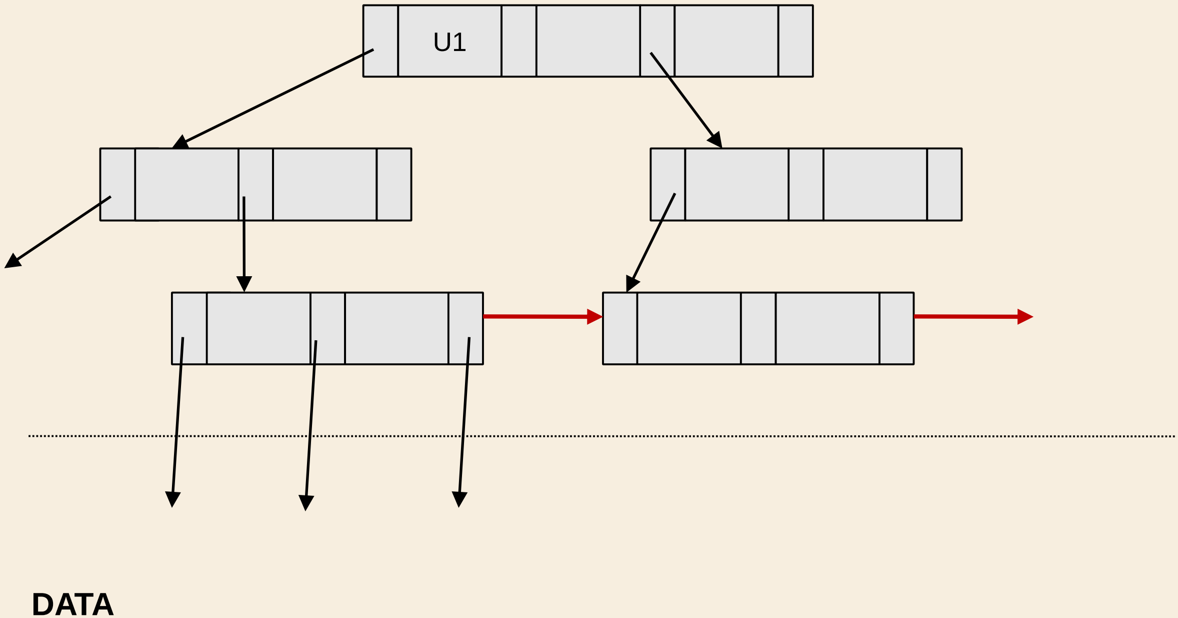
- Two different tree structures for indexing
  - B-Tree
    - Data pages are reached only through key values by visiting the tree
  - B^+-Tree
    - Provides a link structure allowing sequential access in the sort order of key values

B-Tree structure



DATA

B+-Tree structure



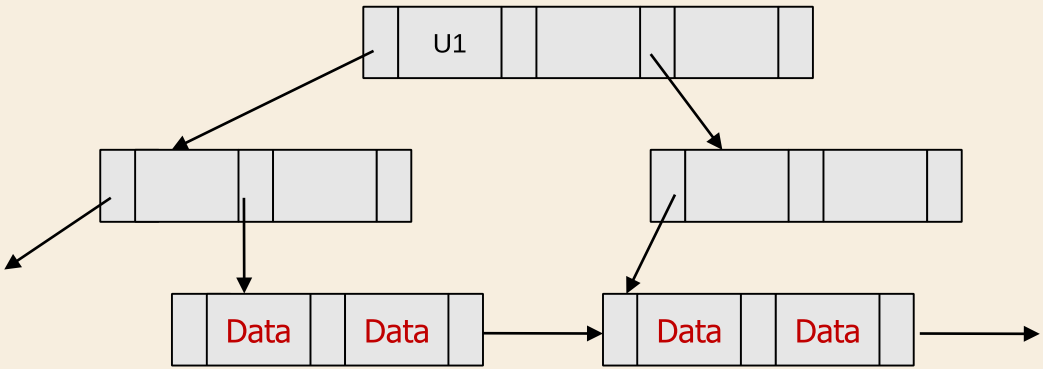
B-Tree and B⁺-Tree

- Two different tree structures for indexing
  - B-Tree
    - Data pages are reached only through key values by visiting the tree
  - B^+-Tree
    - Provides a link structure allowing sequential access on the sort order of key values
- B stands for *balanced*
  - Leaves are all at the same distance from the root
  - Access time is constant, regardless of the searched value

Clustered

- The tuple is contained into the leaf node
  - Constrains the physical position of tuples in a given leaf node
    - The position may be modified by splitting the node, when it is full
  - Also called key sequenced
- Typically used for primary key indexing

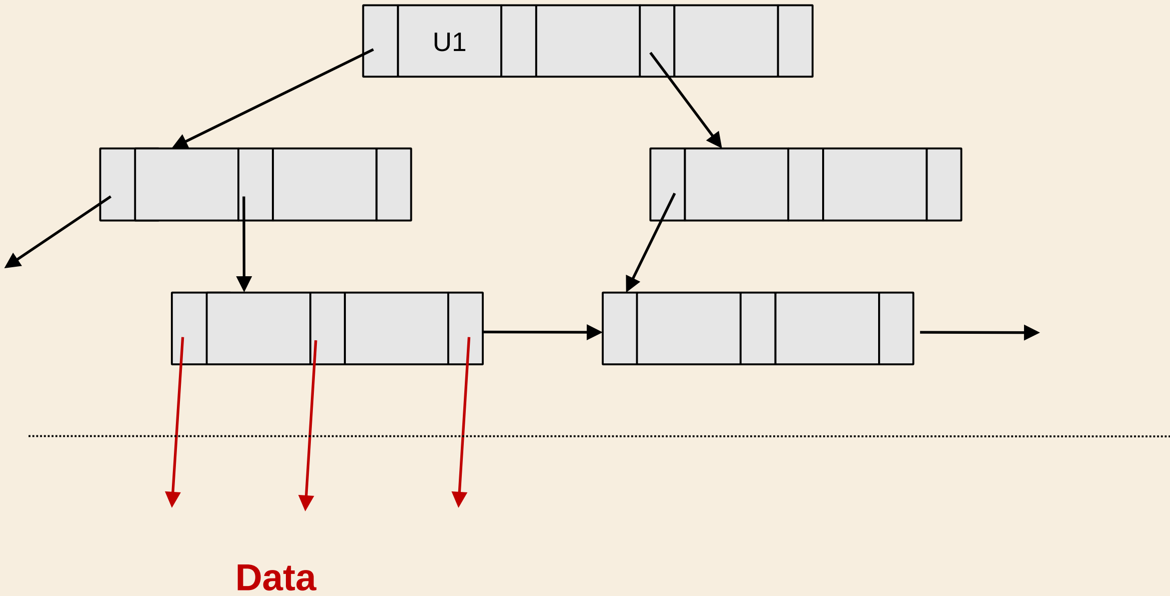
Clustered B^+-Tree index



Unclustered

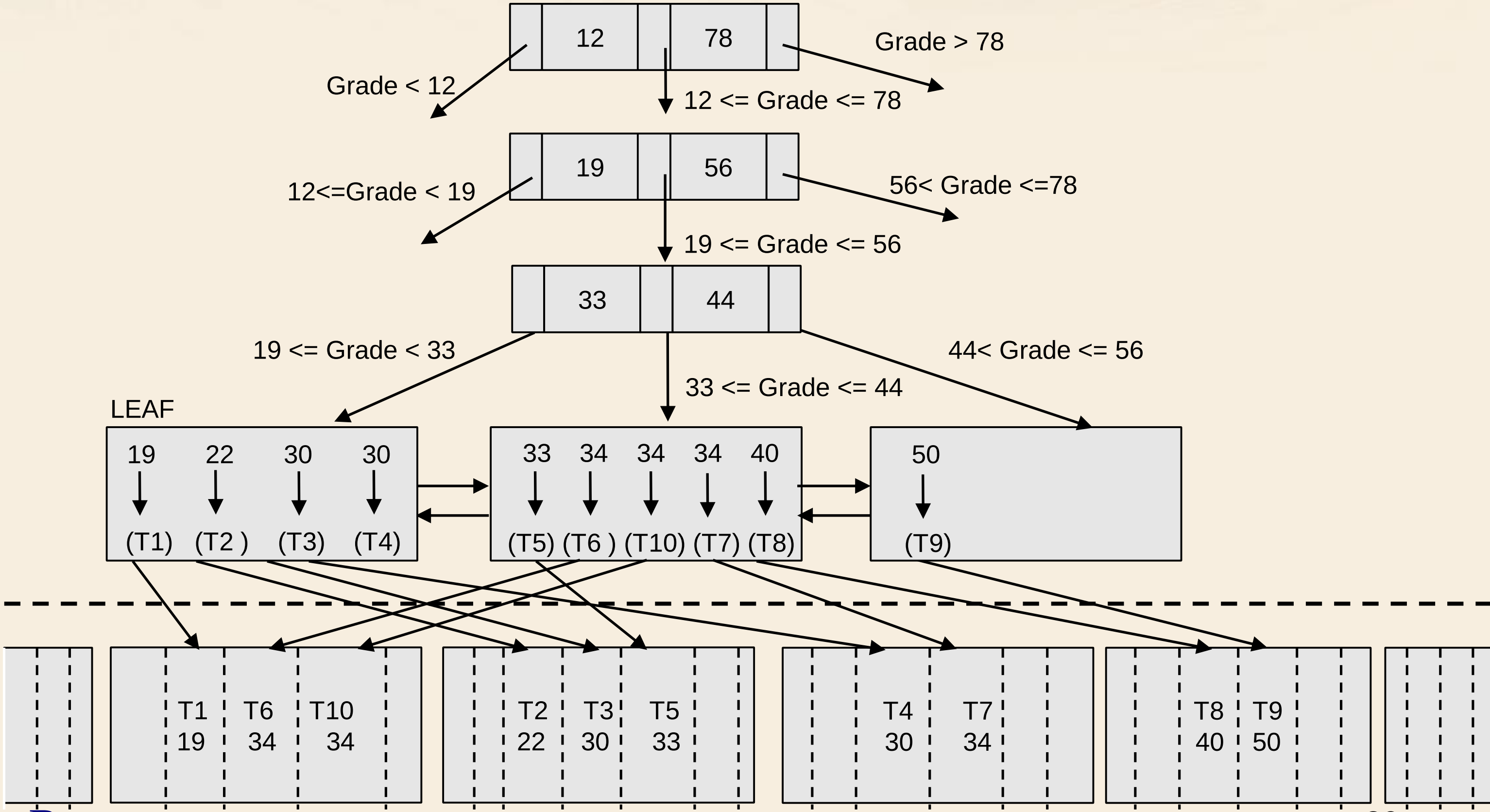
- The leaf contains physical pointers to actual data
  - The position of tuples in a file is totally unconstrained
  - Also called indirect
- Used for secondary indices

Unclustered B+-Tree index



Example: Unclustered B+-Tree index

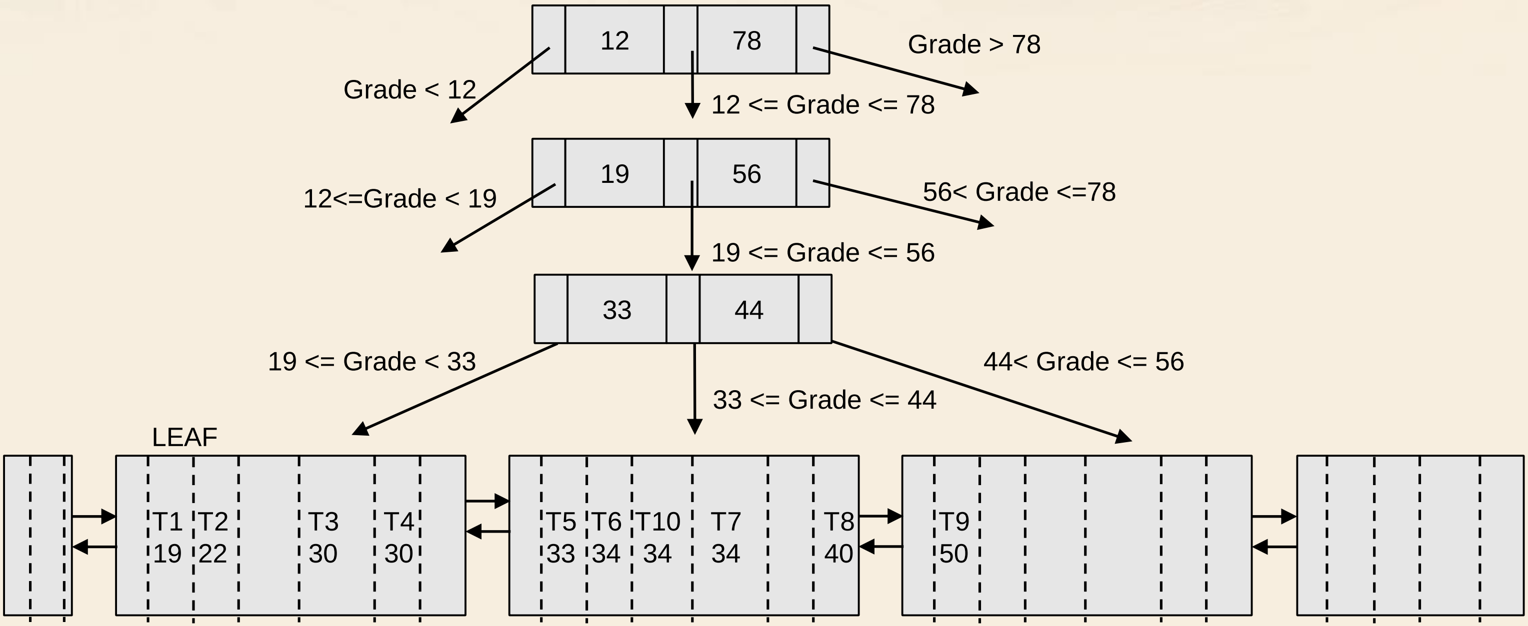
STUDENT (<u>StudentId</u>, Name, *Grade*)



DATA FILE FOR STUDENT TABLE

Example: Clustered B+-Tree index

STUDENT (<u>StudentId</u>, Name, *Grade*)



DATA FILE FOR STUDENT TABLE

Advantages and disadvantages

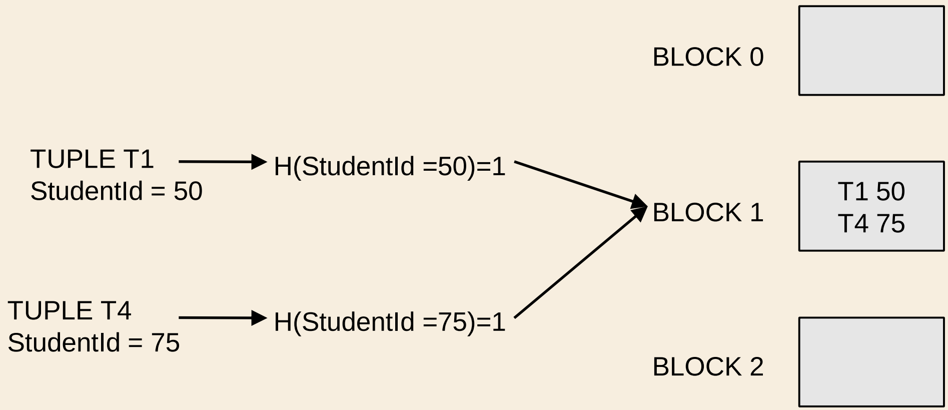
- Advantages
  - Very efficient for range queries
  - Appropriate for sequential scan in the order of the key field
    - Always for clustered, not guaranteed otherwise
- Disadvantages
  - Insertions may require a split of a leaf
    - possibly, also of intermediate nodes
    - computationally intensive
  - Deletions may require merging uncrowded nodes and re-balancing

Hash structure

- It guarantees direct and efficient access to data based on the value of a *key field*
  - The hash key may include one or more attributes
- Suppose the hash structure has B blocks
  - The hash function is applied to the key field value of a record
    - It returns a value between 0 and B-1 which defines the position of the record
  - Blocks should never be completely filled
    - To allow new data insertion

Example: hash index

STUDENT (*StudentId*, Name, Grade)



DATA FILE FOR STUDENT TABLE

Hash index

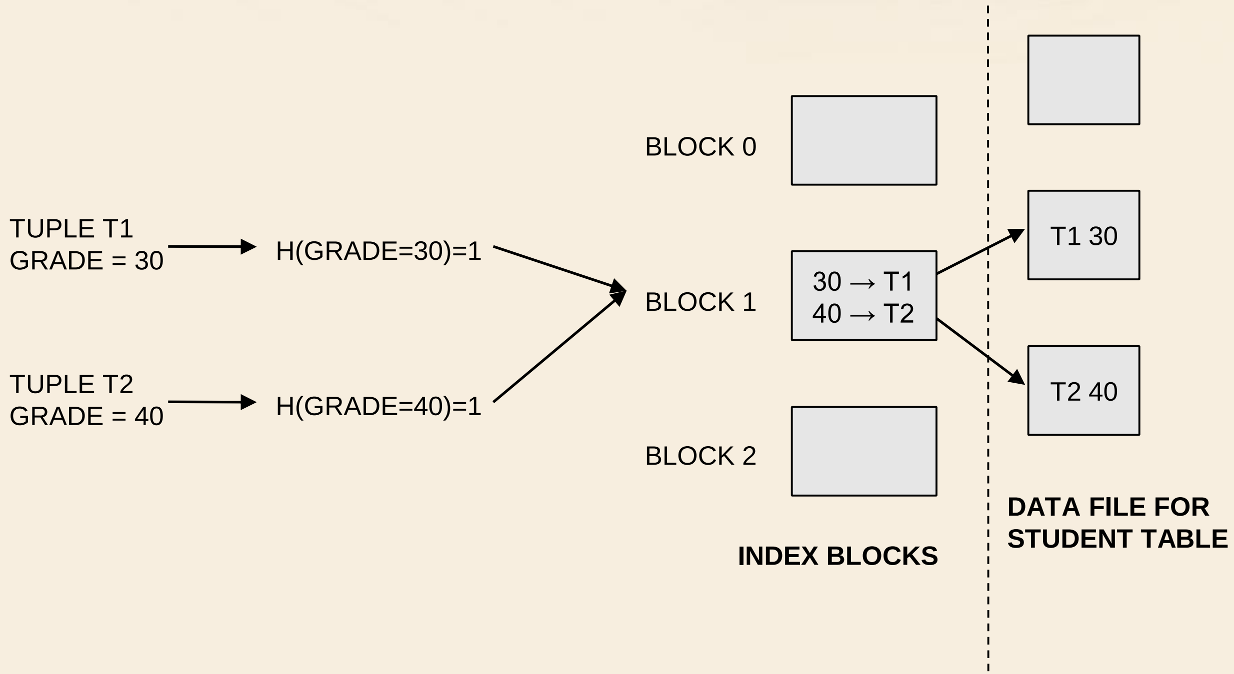
- Advantages
  - Very efficient for queries with equality predicate on the key
  - No sorting of disk blocks is required
- Disadvantages
  - Inefficient for range queries
  - Collisions may occur

Unclustered hash index

- It guarantees direct and efficient access to data based on the value of a *key field*
  - Similar to hash index
- Blocks contain pointers to data
  - Actual data is stored in a separate structure
  - Position of tuples is not constrained to a block
    - Different from hash index

Example: Unclustered hash index

STUDENT (<u>StudentId</u>, Name, *Grade*)



Bitmap index

- It guarantees direct and efficient access to data based on the value of a *key field*
  - It is based on a *bit matrix*
- The bit matrix references data rows by means of RIDs (Row IDentifiers)
  - Actual data is stored in a separate structure
  - Position of tuples is not constrained

Bitmap index

- The bit matrix has
  - One column for each different value of the indexed attribute
  - One row for each tuple
- Position (i, j) of the matrix is
  - 1 if tuple i takes value j
  - 0 otherwise

RID	Val_1	Val_2	…	Val_n
1	0	0	…	1
2	0	0	…	0
3	0	0	…	1
4	1	0	…	0
5	0	1	…	0

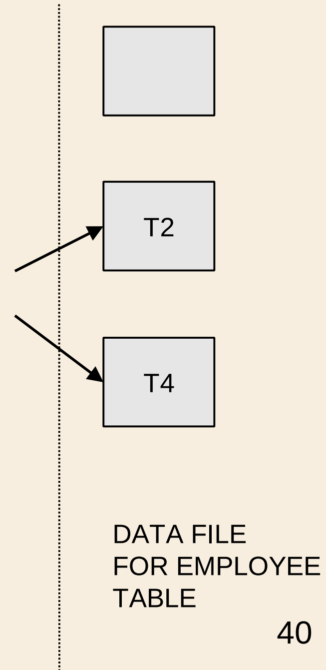
Example: Bitmap index

EMPLOYEE (<u>EmployeeId</u>, Name, *Job*)

Domain of Job attribute = {Engineer, Consultant, Manager, Programmer, Secretary, Accountant}

RID	Eng.	Cons.	Man.	Prog.	Secr.	Acc.
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	1	0
4	0	0	0	1	0	0
5	1	0	0	0	0	0

Prog.
0
1
0
1
0



Bitmap index

- Advantages
  - Very efficient for boolean expressions of predicates
    - Reduced to bit operations on bitmaps
  - Appropriate for attributes with limited domain cardinality
- Disadvantages
  - Not used for continuous attributes
  - Required space grows significantly with domain cardinality