CSCC43H: Introduction to Databases

Lecture 11

Wael Aboulsaadat

Acknowledgment: these slides are partially based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

XPath Syntax

- Path wildcards
  - // = descendant at any level (or self)
  - * = any (single) tag
  - Example: **/booklist//lastname**
- Query attributes and attribute content
  - Use “@”
  - Examples: **/booklist//book[@format=“Paperback”], /booklist//book/@genre**
- Branching predicates: A[*pred*]
  - Predicate on A’s subtree using *logical connectives* (and, or, etc.), *path expressions*, *built-in functions* (e.g., contains()), etc.
  - Example: **//author[contains(./lastname, “Fey”)]**

XQuery FLWOR Expressions

- FOR-LET-WHERE-ORDERBY-RETURN = FLWOR

$$\begin{tikzcd}
\boxed{\text{FOR / LET Clauses}} \arrow[d, "\text{List of tuples}"] \\
\boxed{\text{WHERE Clause}} \arrow[d, "\text{List of tuples}"] \\
\boxed{\text{ORDERBY/RETURN Clause}}
\end{tikzcd}$$

Instance of XQuery data model

XQUERY - examples

```
<bookstore>
        <book category="COOKING">
                <title lang="en">Everyday Italian</title>
                <author>Giada De Laurentiis</author>
                <year>2005</year>
                <price>30.00</price>
        </book>
        <book category="CHILDREN">
                <title lang="en">Harry Potter</title>
                <author>J K. Rowling</author>
                <year>2005</year>
                <price>29.99</price>
        </book>
        <book category="WEB">
                <title lang="en">XQuery Kick Start</title>
                <author>James McGovern</author>
                <author>Per Bothner</author>
                <author>Kurt Cagle</author>
                <author>James Linn</author>
                <author>Vaidyanathan Nagarajan</author>
                <year>2003</year>
                <price>49.99</price>
        </book>
        <book category="WEB">
                <title lang="en">Learning XML</title>
                <author>Erik T. Ray</author>
                <year>2003</year>
                <price>39.95</price>
        </book>
</bookstore>
```

- **Query: doc("books.xml")/bookstore/book/title**
- **Result:**

```
<title lang="en">Everyday Italian</title>
<title lang="en">Harry Potter</title>
<title lang="en">XQuery Kick Start</title>
<title lang="en">Learning XML</title>
```

XQUERY - examples

```
<bookstore>
        <book category="COOKING">
                <title lang="en">Everyday Italian</title>
                <author>Giada De Laurentiis</author>
                <year>2005</year>
                <price>30.00</price>
        </book>
        <book category="CHILDREN">
                <title lang="en">Harry Potter</title>
                <author>J K. Rowling</author>
                <year>2005</year>
                <price>29.99</price>
        </book>
        <book category="WEB">
                <title lang="en">XQuery Kick Start</title>
                <author>James McGovern</author>
                <author>Per Bothner</author>
                <author>Kurt Cagle</author>
                <author>James Linn</author>
                <author>Vaidyanathan Nagarajan</author>
                <year>2003</year>
                <price>49.99</price>
        </book>
        <book category="WEB">
                <title lang="en">Learning XML</title>
                <author>Erik T. Ray</author>
                <year>2003</year>
                <price>39.95</price>
        </book>
</bookstore>
```

- **Query:**
  ```
  doc("books.xml")/bookstore/
                  book[price<30]
  ```
- **Result:**
  ```
  <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
  </book>
  ```

XQUERY - examples

```
<bookstore>
        <book category="COOKING">
                <title lang="en">Everyday Italian</title>
                <author>Giada De Laurentiis</author>
                <year>2005</year>
                <price>30.00</price>
        </book>
        <book category="CHILDREN">
                <title lang="en">Harry Potter</title>
                <author>J K. Rowling</author>
                <year>2005</year>
                <price>29.99</price>
        </book>
        <book category="WEB">
                <title lang="en">XQuery Kick Start</title>
                <author>James McGovern</author>
                <author>Per Bothner</author>
                <author>Kurt Cagle</author>
                <author>James Linn</author>
                <author>Vaidyanathan Nagarajan</author>
                <year>2003</year>
                <price>49.99</price>
        </book>
        <book category="WEB">
                <title lang="en">Learning XML</title>
                <author>Erik T. Ray</author>
                <year>2003</year>
                <price>39.95</price>
        </book>
</bookstore>
```

- **Query:**
  for $x in doc("books.xml")/bookstore
  /book where $x/price>30 order
  by $x/title return $x/title
- **Result:**
  <title lang="en">Learning XML</title>
  <title lang="en">XQuery Kick Start</title>

FOR vs. LET

- FOR $x IN path-expression
  - Binds $x in turn to each element in the expression

- LET $x := path-expression
  - Binds $x to the *entire list of elements* in the expression
  - Useful for common sub-expressions and for aggregations

FOR vs. LET: Example

```
FOR $x IN document("bib.xml")/bib/book
          RETURN <result> $x </result>
```

Returns:

```
<result> <book>...</book></result>
<result> <book>...</book></result>
<result> <book>...</book></result>
...
```

Notice that result has several elements

```
LET $x := document("bib.xml")/bib/book
          RETURN <result> $x </result>
```

Returns:

```
<result> <book>...</book>
         <book>...</book>
         <book>...</book>
          ...
</result>
```

Notice that result has exactly one element

XQuery Example 1

Find all book titles published after 1995:

```
FOR $x IN document("bib.xml")/bib/book
WHERE $x/year > 1995
RETURN $x/title
```

Result:

```
<title> abc </title>
<title> def </title>
<title> ghi </title>
```

XQuery Example 2

For each author of a book by Morgan Kaufmann, list all books she published:

```
FOR $a IN distinct(
 document("bib.xml"/bib/book[publisher="Morgan Kaufmann"]/author))

RETURN <result>

                $a,

                FOR $t IN /bib/book[author=$a]/title

                RETURN $t

            </result>
```

distinct = a function that eliminates duplicates (after converting inputs to atomic values)

Results for Example 2

```
<result>
        <author>Jones</author>
        <title> abc </title>
        <title> def </title>
</result>
<result>
         <author> Smith </author>
         <title> ghi </title>
</result>
```

Observe how nested structure of result elements is determined by the nested structure of the query.

XQuery Example 3

```
<big_publishers>
    FOR $p IN distinct(document("bib.xml")//publisher)
    LET $b := document("bib.xml")/book[publisher = $p]
    WHERE count($b) > 100
    RETURN $p
</big_publishers>
```

For each publisher p

- Let the list of books published by p be b

Count the # books in b, and return p if b > 100

count = (aggregate) function that returns the number of elements

XQuery Example 4

Find books whose price is larger than average:

```
LET $a := avg(document("bib.xml")/bib/book/price)

FOR $b in document("bib.xml")/bib/book

WHERE $b/price > $a

RETURN $b
```

Collections in XQuery

Not covered in class

- Ordered and unordered collections
  - /bib/book/author = an ordered collection
  - Distinct(/bib/book/author) = an unordered collection
- Examples:
  - LET $a = /bib/book → $a is a collection
  - $b/author → also a collection (several authors...)

However:

RETURN <result> $b/author </result>

Returns a single collection!

```
<result> <author>...</author>
         <author>...</author>
         <author>...</author>
         ...
</result>
```

Collections in XQuery

Not covered in class

What about collections in expressions ?

- $b/price → list of n prices

- $b/price * 0.7 → list of n numbers??
- $b/price * $b/quantity → list of n x m numbers ??
  - Valid only if the two sequences have at most one element
  - Atomization

- $book1/author eq "Kennedy" - Value Comparison
- $book1/author = "Kennedy" - General Comparison

Sorting in XQuery

Not covered in class

```
<publisher_list>
    FOR $p IN distinct(document("bib.xml")//publisher)
    ORDERBY  $p
    RETURN <publisher> <name> $p/text() </name> ,
                        FOR $b IN document("bib.xml")//book[publisher = $p]
                        ORDERBY $b/price DESCENDING
                        RETURN <book>
                                    $b/title ,
                                    $b/price
                               </book>
                  </publisher>
</publisher_list>
```

Conditional Expressions: If-Then-Else

Not covered in class

```
FOR $h IN //holding
ORDERBY $h/title
RETURN <holding>

                    $h/title,

                    IF $h/@type = "Journal"

                            THEN $h/editor

                    ELSE $h/author

              </holding>
```

XML & PostgreSQL

- Store XML documents as text BLOBs (Binary Large Objects) inside text-valued columns

- Load XML *in-memory* and use external User- Defined Functions (UDFs) to process XPath expressions
  - xpath_bool(xml_text_col, “xpath_query_string”)
    - False/true if element set discovered is empty/nonempty
  - xpath_nodeset(xml_text_col, “xpath_query_string”)
    - Text result = concatenation of element subtrees

- No support for full-fledged XQuery
  - Some support for XSLT transformations -- won’t discuss here…

- Pros/Cons??

Storage and Indexing

Motivation

- DBMS stores vast quantities of data
- Data is stored on external storage devices and fetched into main memory as needed for processing
- Page is unit of information read from or written to disk. (in DBMS, a page may have size 8KB or more).
- Data on external storage devices :
  - Disks: Can retrieve random page at fixed cost

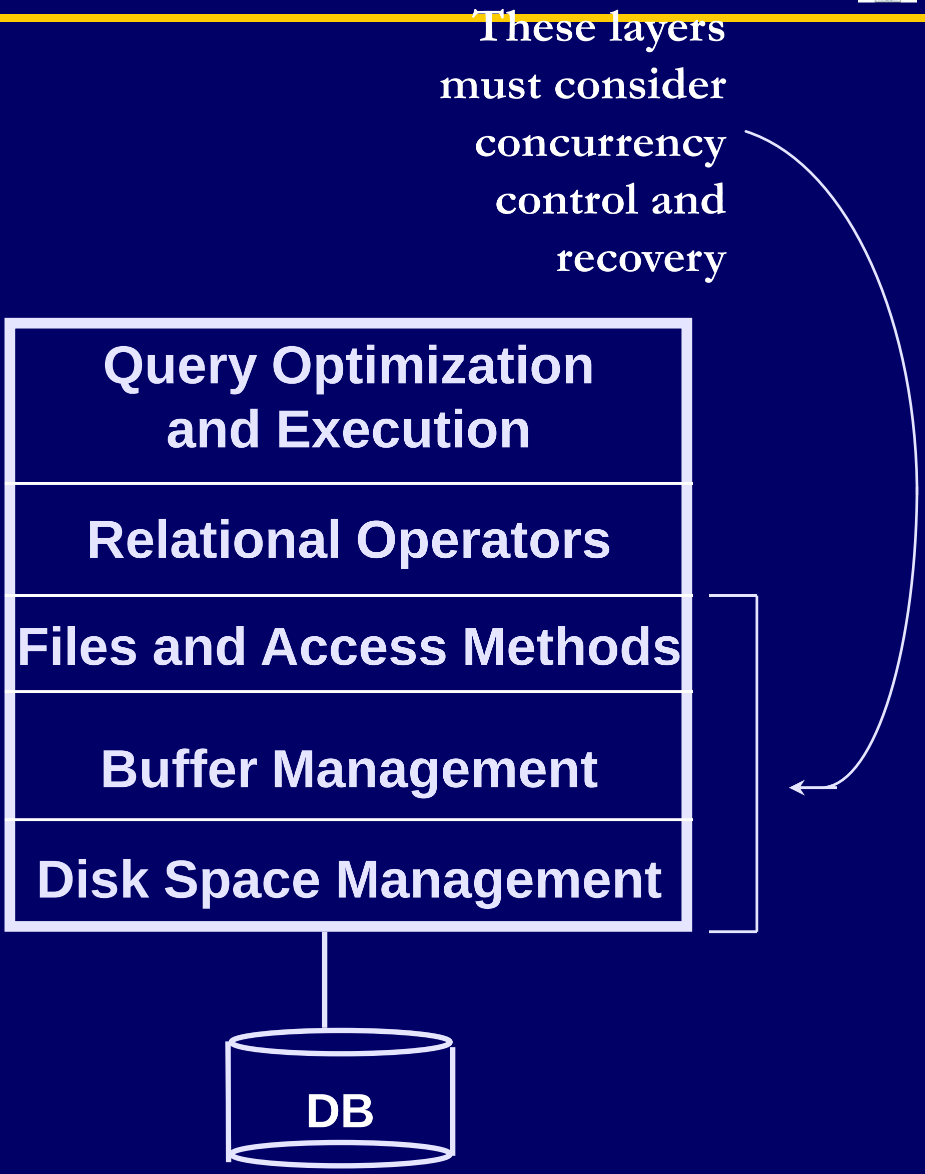
    But reading several consecutive pages is much cheaper than reading them in random order
  - Tapes: Can only read pages in sequence

    Cheaper than disks; used for archival storage

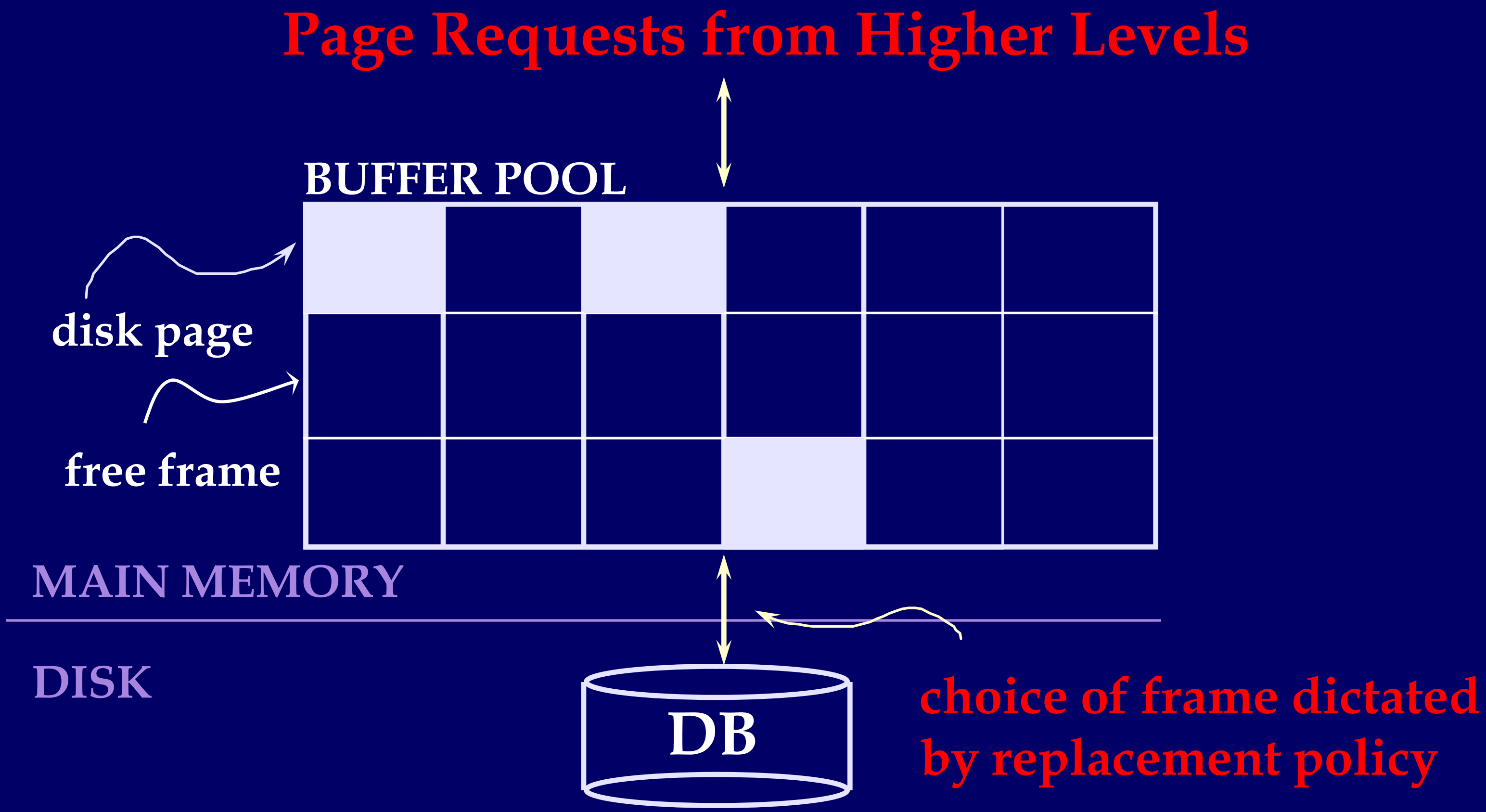
- Cost of page I/O dominates cost of typical database operations

Structure of a DBMS: Layered Architecture

- external storage access
  - Disk space manager manages persistent data
  - Buffer manager stages pages from external storage to main memory buffer pool.
  - File and index layers make calls to buffer manager.


These layers must consider concurrency control and recovery
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
DB

Buffer Manager


Page Requests from Higher Levels
BUFFER POOL
disk page
free frame
MAIN MEMORY
DISK
DB
choice of frame dictated by replacement policy

- *Data must be in RAM for DBMS to operate on it!*
- *Buffer Mgr hides the fact that not all data is in RAM*

Buffer Manager

- Similar to *virtual memory manager*
- Buffer replacement policies
  - What page to evict ?
  - LRU: Least Recently Used
    - Throw out the page that was not used in a long time
  - MRU: Most Recently Used
    - The opposite
    - Why ?

Buffer Manager

- *Pinning* a block
  - Not allowed to write back to the disk
- *Force-output (force-write)*
  - Force the contents of a block to be written to disk
- *Order the writes*
  - This block must be written to disk before this block
- Critical for fault tolerant guarantees
  - Otherwise the database has no control over whats on disk and whats not on disk

File Organization for DB ?

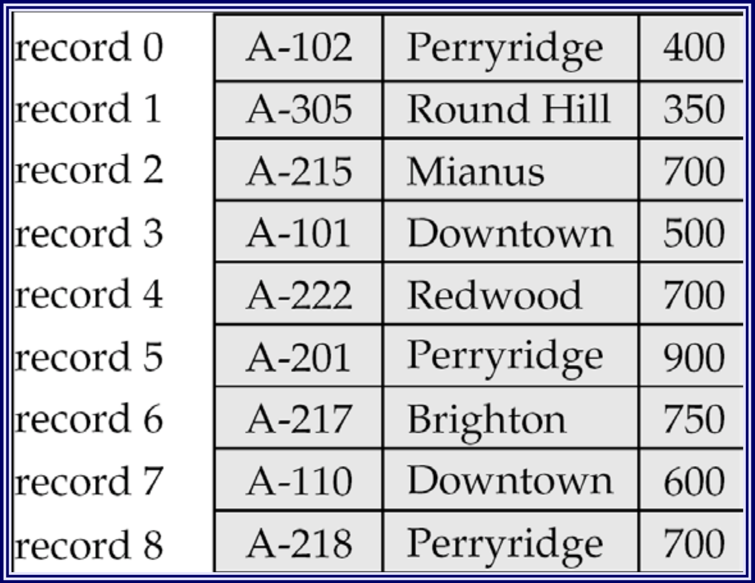
- How are the relations mapped to the disk blocks ?
  - Use a standard file system ?
    - High-end systems have their own OS/file systems (e.g. Oracle)
    - OS interferes more than helps in many cases
  - Mapping of relations to file ?
    - One-to-one ?
    - Advantages in storing multiple relations clustered together
  - A *file* is essentially a *collection of disk blocks*
    - How are the tuples mapped to the disk blocks ?
    - How are they stored within each block

File Organization for DB

- Goals:
  - Allow insertion/deletions of tuples/records
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123) ?
- Simplest case
  - Each relation is mapped to a file
  - A file contains a sequence of records
  - Each record corresponds to a logical tuple
- Next:
  - How are tuples/records stored within a block ?

Fixed Length Records

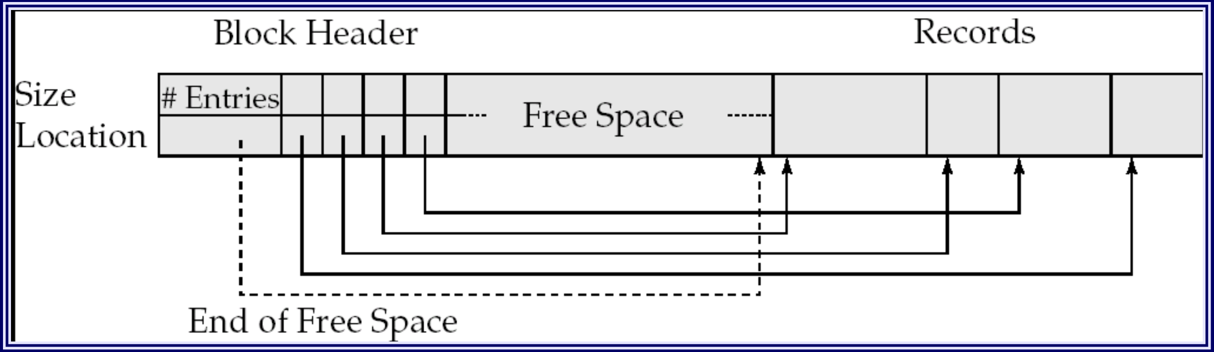
- n = number of bytes per record
- Store record *i* at position:
  - n * (i – 1)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn't happen
- Inserting a tuple ?
  - Depends on the policy used
  - One option: Simply append at the end of the record
- Deletions ?
  - Option 1: Rearrange
  - Option 2: Keep a *free list* and use for next insert



record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Variable-length Records

Slotted page structure



- *Indirection:*
  - The records may move inside the page, but the outside world is oblivious to it
  - Why ?
    - The headers are used as a indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

Sequential File Organization

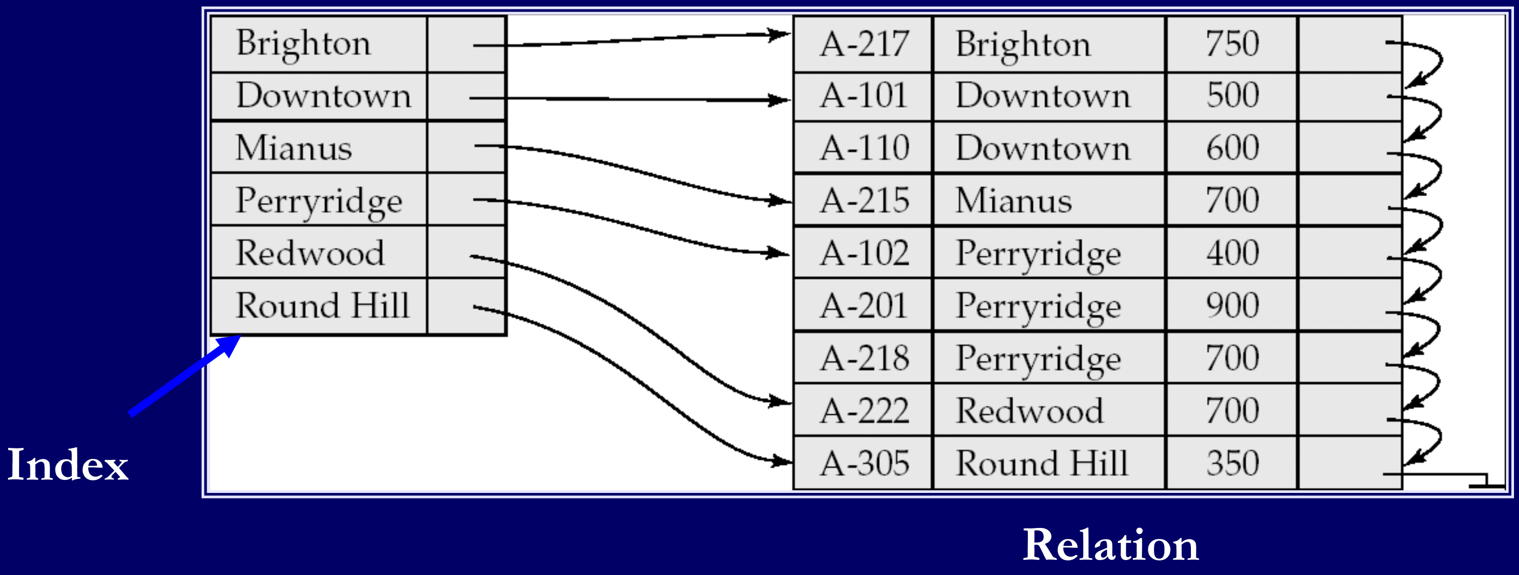
- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become *fragmented*
  - Must reorganize once in a while
- What if we want to find a particular record by value ?

Indexed File Organization

- A data structure for efficient search through large databases
- Two key ideas:
  1) The records are mapped to the disk blocks in specific ways
     - Sorted, or hash-based
  2) Auxiliary data structures are maintained that allow quick search
- Think library index/catalogue
- Search key:
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a persons table
- Two types of indexes
  - Ordered indexes
  - Hash-based indexes

Ordered Indexes

- Primary index
  - The relation is sorted on the key of the index
- Secondary index
  - It is not
- Can have only one primary index on a relation

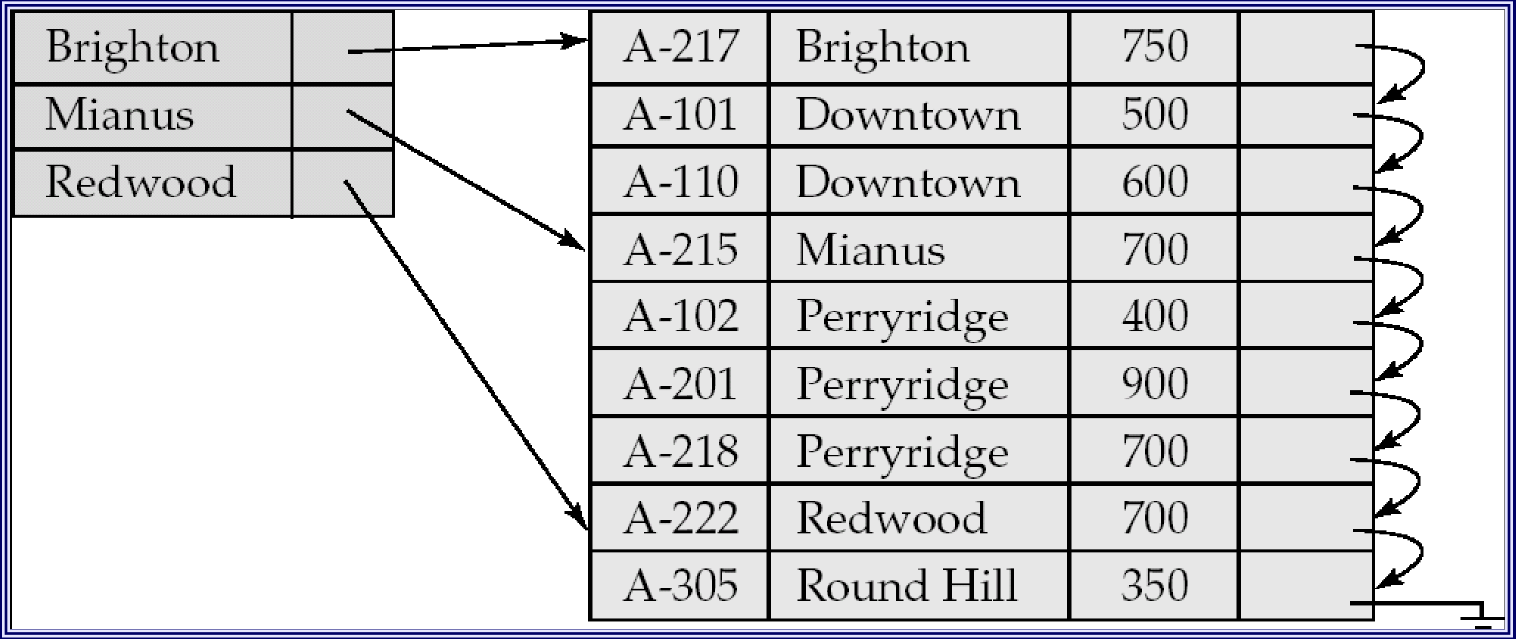


Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Primary *Sparse* Index

- Every key doesn't have to appear in the index
- Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoff: Must access the relation file even if the record is not present

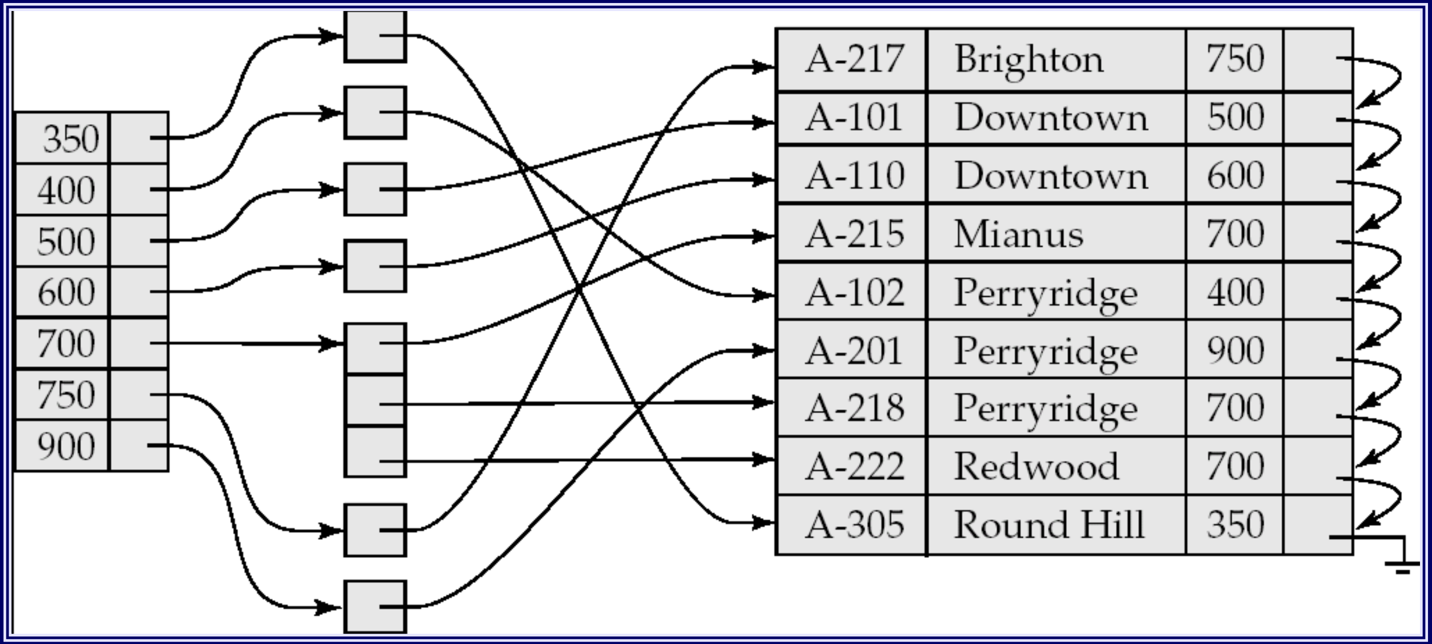


Brighton	
Mianus	
Redwood	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Secondary Index

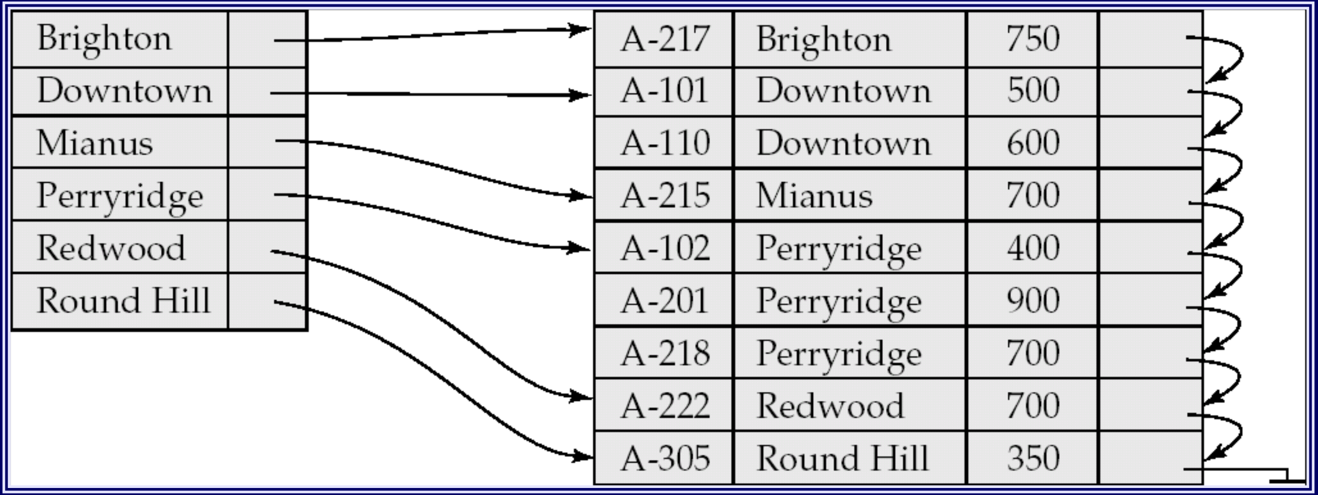
- E.g.
  - Relation sorted on *branch* but we want an index on *balance*
- Must be dense
  - Every search key must appear in the index



A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Question?

- What if index is too large to search sequentially?

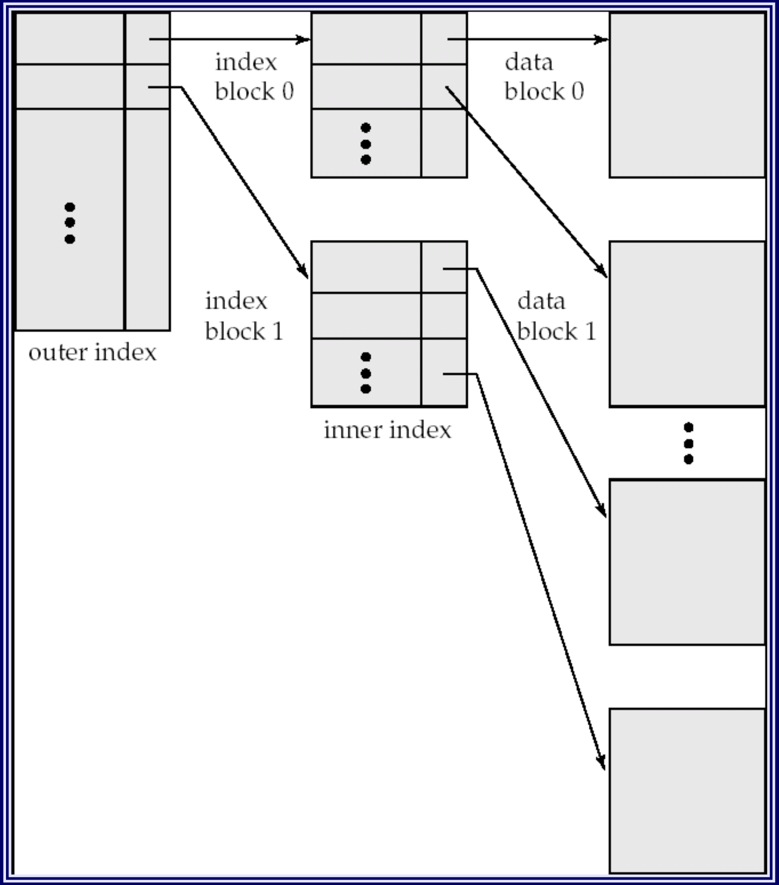


Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Multi-level Indexes

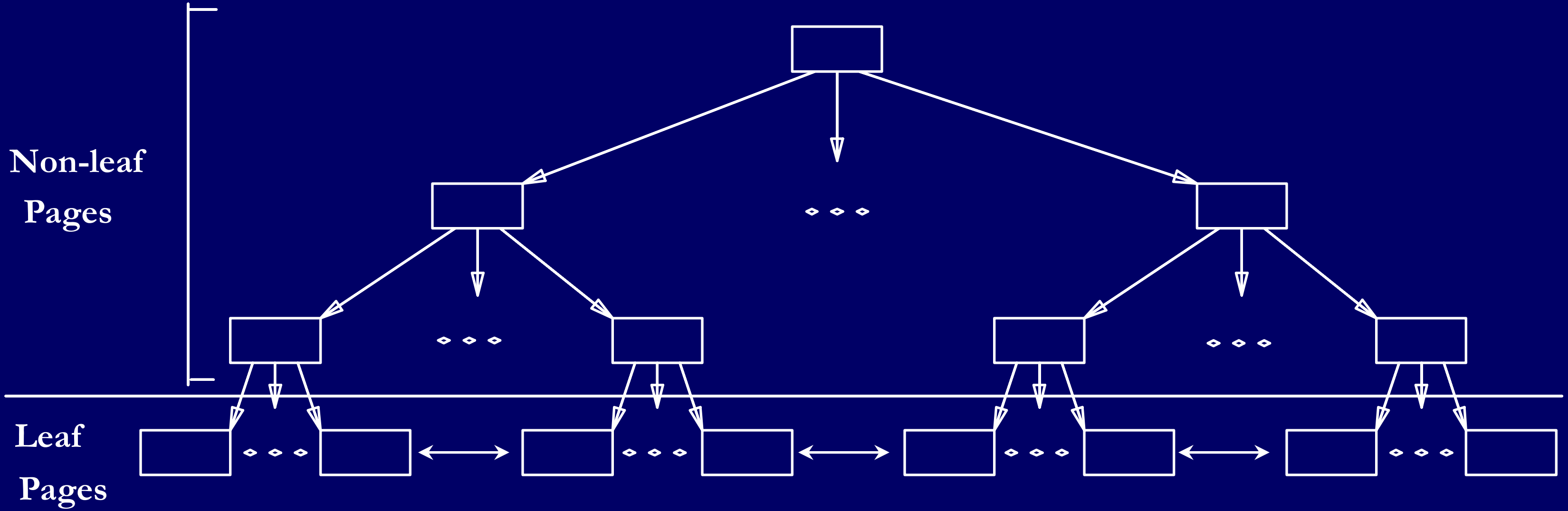
- What if the index itself is too big for memory ?
- Relation size = n = 1,000,000,000
- Block size = 100 tuples per block
- So, number of pages = 10,000,000
- Keeping one entry per page takes too much space
- Solution
  - Build an index on the index itself



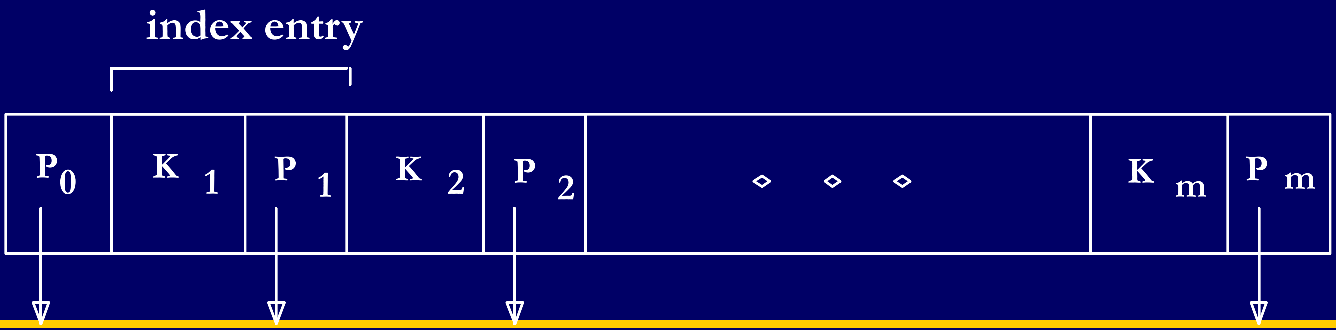
Question?

- What is the best data structure to build indexes of data?

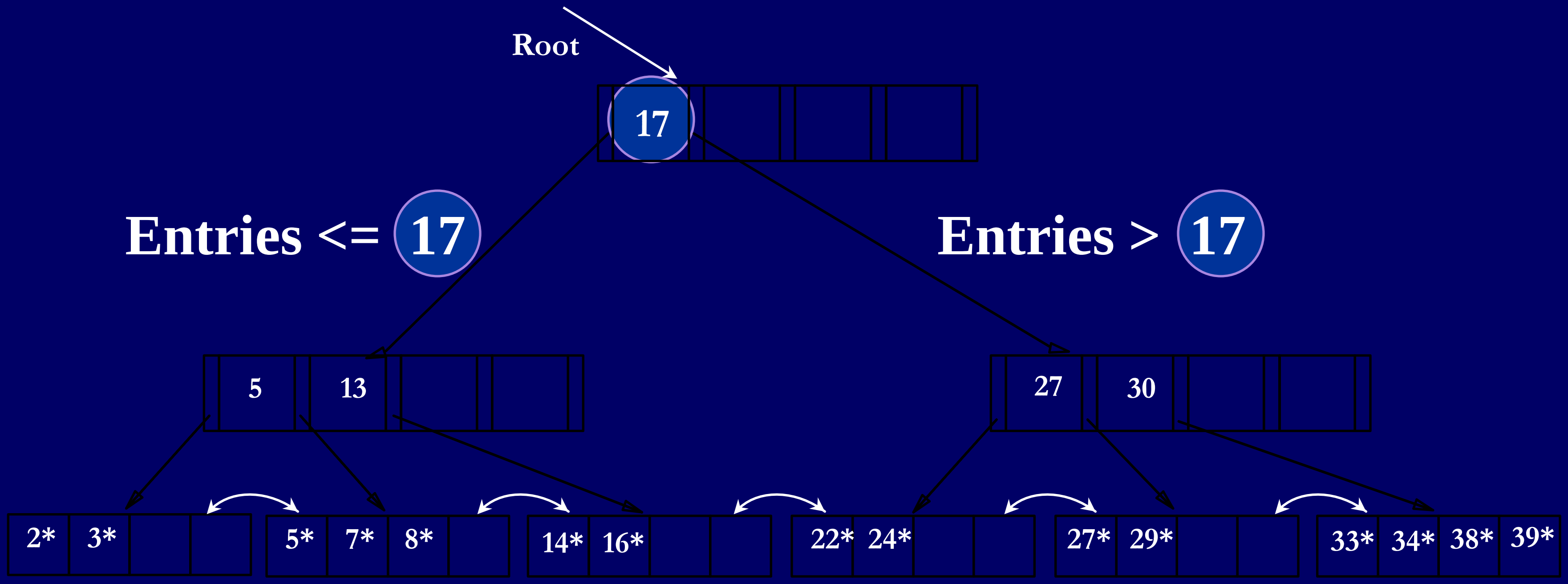
B+ Tree Indexes



- **Leaf** pages contain *data entries*, and are **chained** (prev & next
- **Non-leaf** pages contain *index entries* and direct searches:



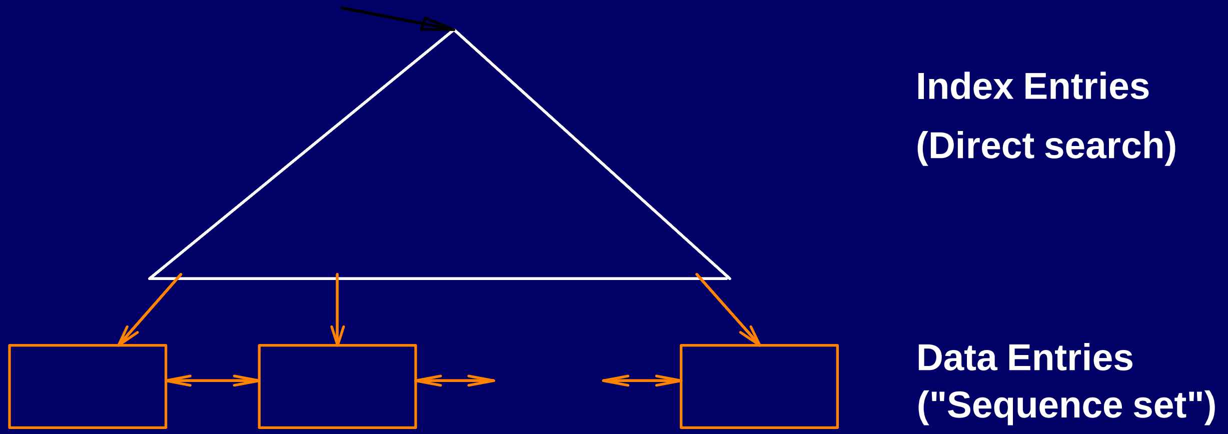
Example B+ Tree



B+ Tree Structure

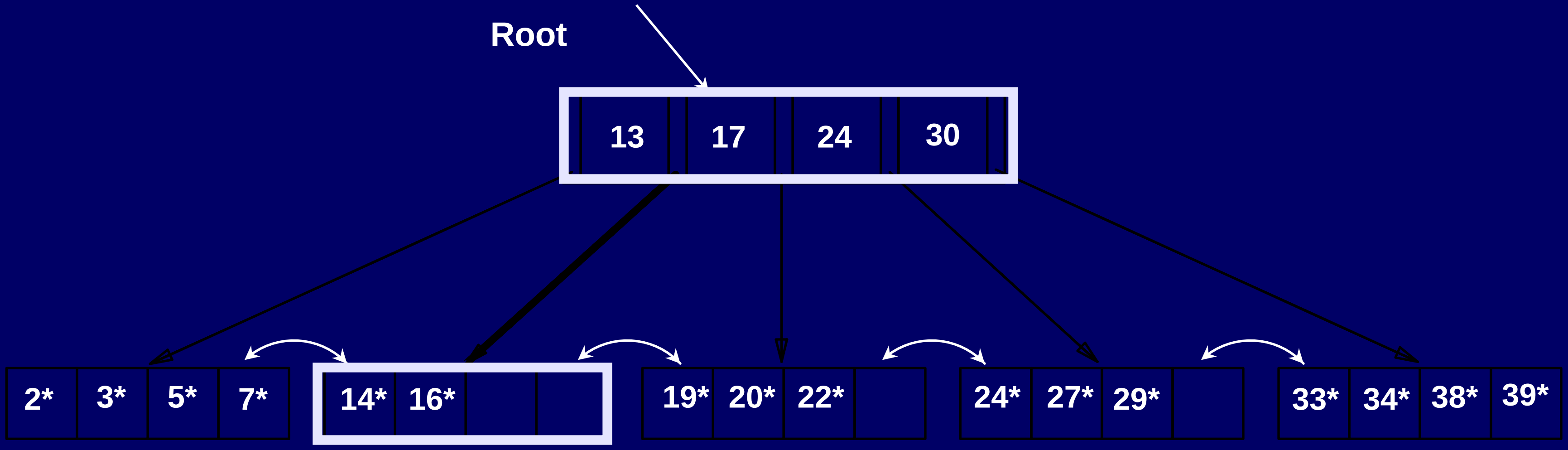
- Keep tree *height-balanced*
  - Supports equality and range-searches efficiently.

- Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.



B+ Tree Equality Search

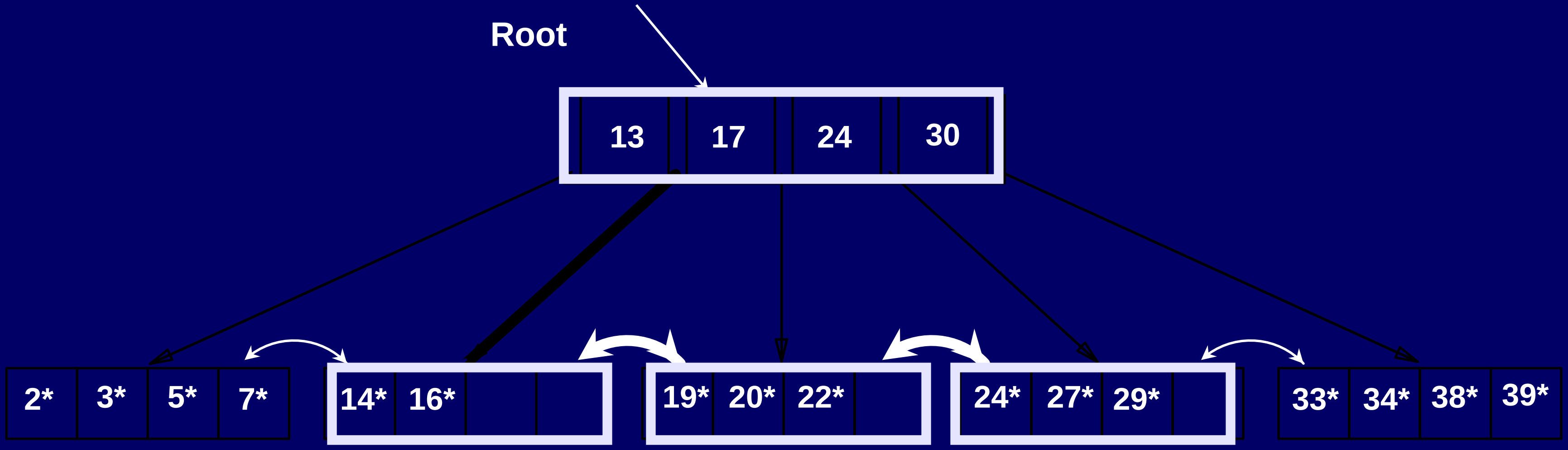
- Search begins at root, and key comparisons direct it to a leaf.
- Search for 15*…



✉ *Based on the search for 15*, we know it is not in the tree!*

B+ Tree Range Search

- Search all records whose ages are in [15,28].
  - Equality search 15*.
  - Follow sibling pointers.



Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.
- Consider the most important queries in turn.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Examples of Indexes

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- Equality queries and duplicates:
  - Indexing on *E.hobby* helps!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Index-Only Queries

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

<E.dno>

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno
```

<E.dno,E.sal>

Tree index!

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno
```

<E. age,E.sal>
or
<E.sal, E.age>

Tree index!

```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
E.sal BETWEEN 3000 AND 5000
```

How to create an index in SQL ?

- Syntax

```
CREATE INDEX  Index-Name on Table-Name(Columns…)
```

- Example:

```
TABLE Customer      (First_Name char(50),
                           Last_Name char(50),
                           Address char(50),
                           City char(50),
                           Country char(25),
                           Birth_Date date)

CREATE INDEX  IDX_CUSTOMER_LAST_NAME on CUSTOMER (Last_Name)

CREATE INDEX IDX_CUSTOMER_LOCATION on CUSTOMER (City, Country)
```