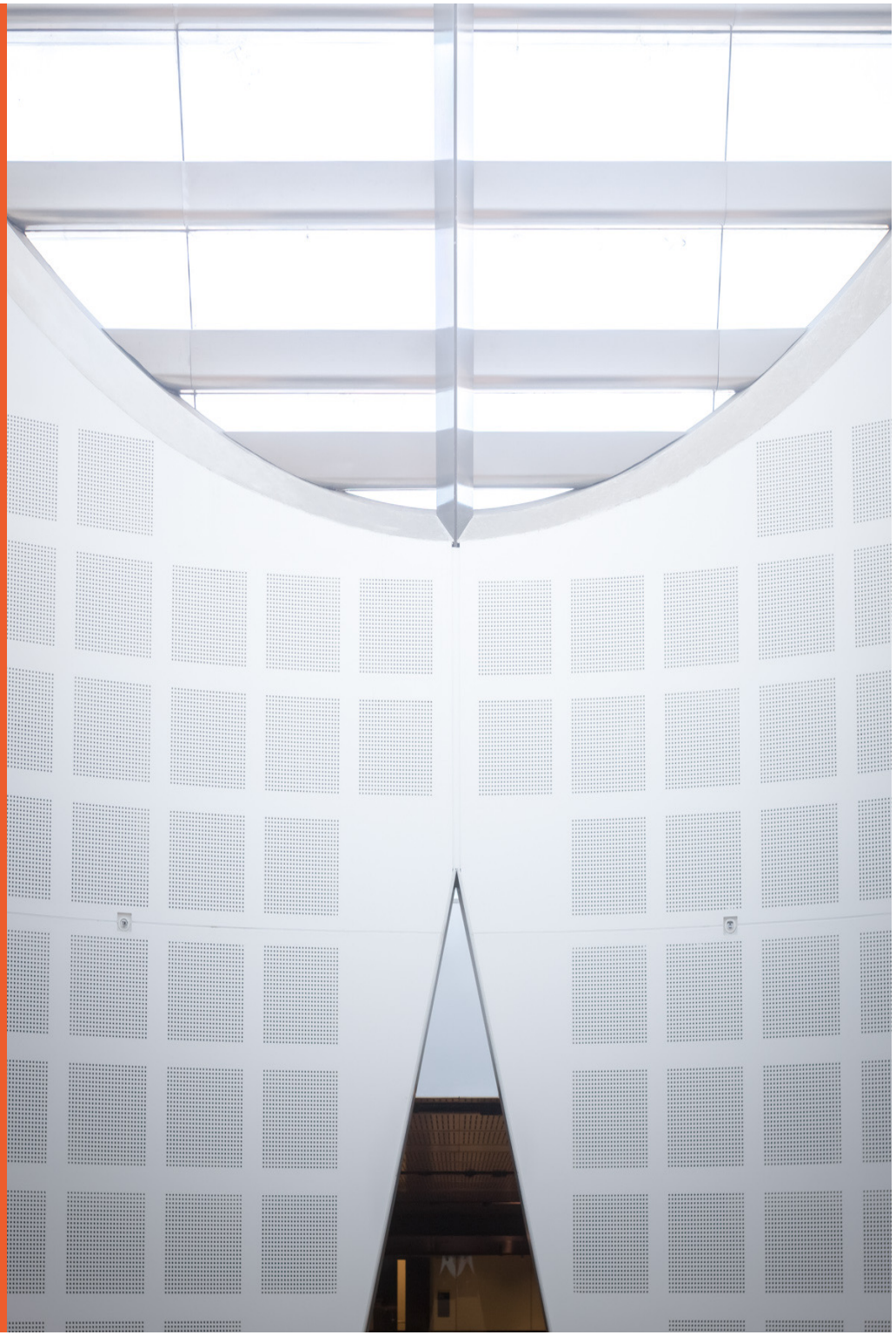
ISYS2120: Data & Information Management

Week 11: Indexing and database tuning

Alan Fekete

Based on slides from Kifer/Bernstein/Lewis (2006) "Database Systems"
and from Ramakrishnan/Gehrke (2003) "Database Management Systems",
and including material from Fekete, Roehm, Khushi



THE UNIVERSITY OF SYDNEY

Agenda

- Database Storage Layer: Physical Data Organisation
  - Motivation: Data stored on disks
  - Several physical design alternatives possible for same logical schema
- Indexing of Databases
  - Efficient data access based on search keys
  - Several design decisions…
- Database Tuning
  - How to suggest appropriate indexes for a given SQL workload
  - Awareness of the trade-off between query performance and indexing costs (updates)

Server structure

- Code that performs DBMS computations runs on the CPU
- CPU operations use data in a few registers, which are themselves loaded/stored from the program memory which is held in DRAM
  - DRAM is “volatile”: data is not still there after system crashes
  - Also OS overwrites this data after the process finishes running
- Files are held on disk (traditionally “hard disk drive” abbreviated “HDD”) which keep the data persistent

Understanding the hardware for storing the data

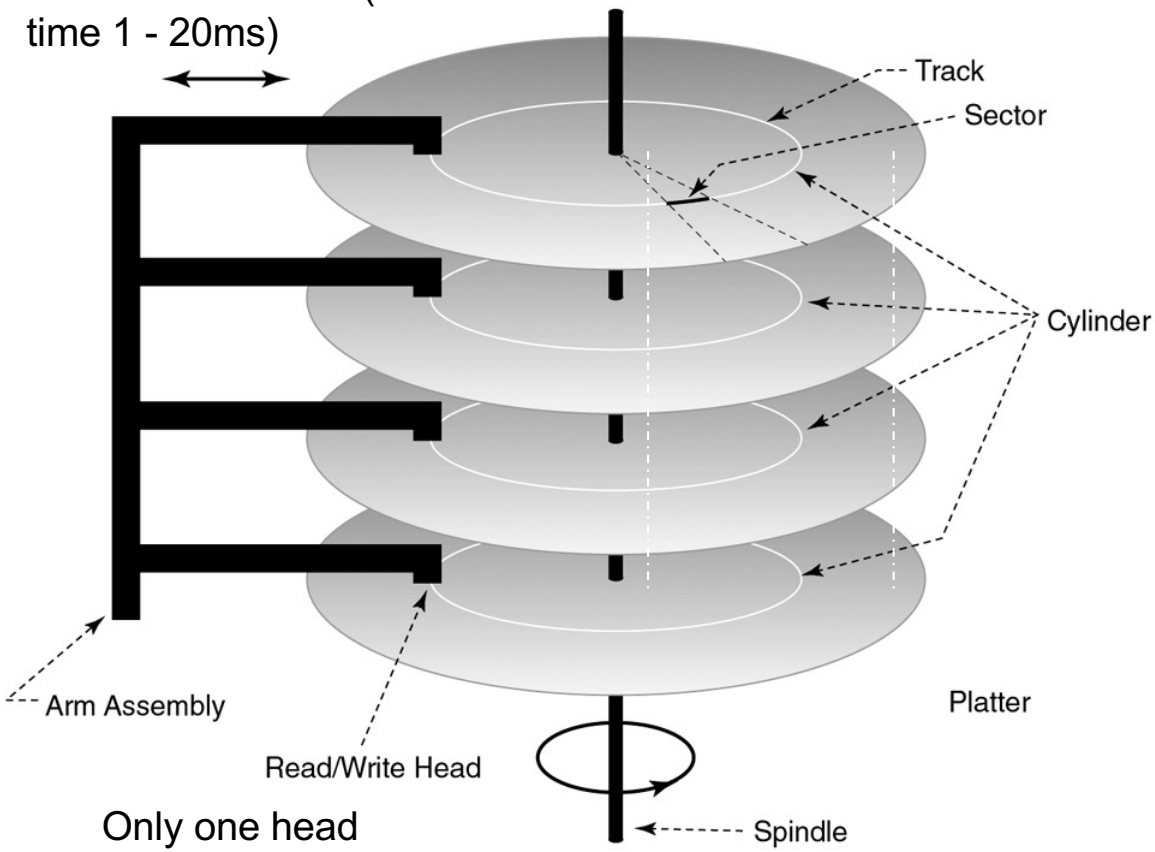
- Where will the DBMS put the data it holds?
  - *Main memory is accessed much faster than HDDs*
    - Order of 0.1μs vs 10ms [that is, 100,000 times faster!]
  - *Main memory (DRAM) is much more expensive (per Mb) than HDDs, so a system can't afford so much of it*
    - A very large dataset may not fit into the available DRAM
  - *Main memory is volatile*
    - We insist that data to be saved between runs, and despite crashes. (Obviously!)
- This has major implications for DBMS design!
  - Data is kept on disk (maybe managed through OS file system, but often the DBMS will deal directly with the disk hardware)
  - When it is used in a computation, data must be in main memory
  - DBMS needs to manage movement of data between disk and main memory
    - Moving between levels are high-cost operations, relative to in-memory operations, so must be planned carefully!
  - Indeed, overall performance is determined largely by the number of disk I/Os done

Physical Disk Structure

http://databacknow.com/harddisk_internals.html

Arm assembly moved in or out to position a head on a desired track. (Seek time 1 - 20ms)

Block size is a multiple of sector size, typically 4kB, 8kB
Transfer rate ~0.03 ms per 4kB block



Only one head reads/writes at any one time.

Platters spin ~ 7200 rpm

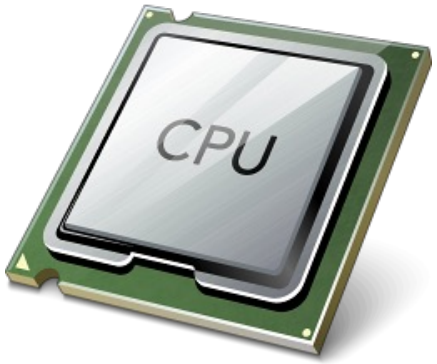
Traditional Storage Hierarchy

- **primary storage:** Fastest media but volatile (cache, DRAM).
- **secondary storage:** next level in hierarchy, non-volatile, intermediate access time
  - also called **on-line storage**
  - E.g.: hard disks, solid-state drives
- **tertiary storage:** lowest level in hierarchy, non-volatile, very slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage
- Typical storage hierarchy:
  - Main memory (DRAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

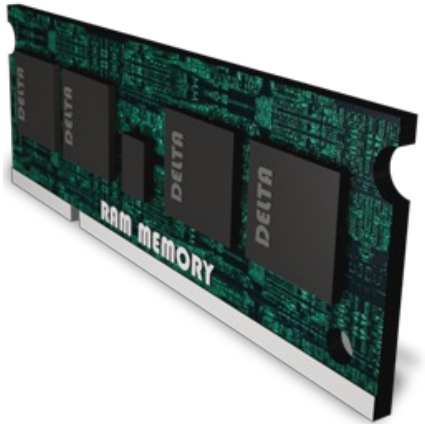
Where is Data Stored?

Main Memory (DRAM):
- Expensive
- Fast
- Volatile

Secondary Storage (HDD):
- Cheap
- Stable
- BIG



FAST



SLOW

Tertiary Storage (e.g. Tape
- Very cheap
- Very slow
- Stable

http://www.software182.com/2015/03/cpu.html

https://www.iconfinder.com/icons/104158/ram_icon

https://www.iconfinder.com/icons/18285/harddisk_hdd_icon

http://www-03.ibm.com/systems/storage/media/3592/index.html

Solid State Technology

- Newer technology stores data in the state of integrated circuits rather than by magnetising metal
  - that has some intermediate features between HDD and DRAM
- Data persists despite power failure
  - Only a limited (but fairly large) number of changes in any location
- Access speeds are often quite different for reads versus writes
- Some ("Persistent Memory" or "NVRAM") are manufactured to provide same interface as traditional DRAM; while others ("SSD") fit in to the system as replacement for HDD

- Active research happening on how best to redesign database management platforms to use this effectively

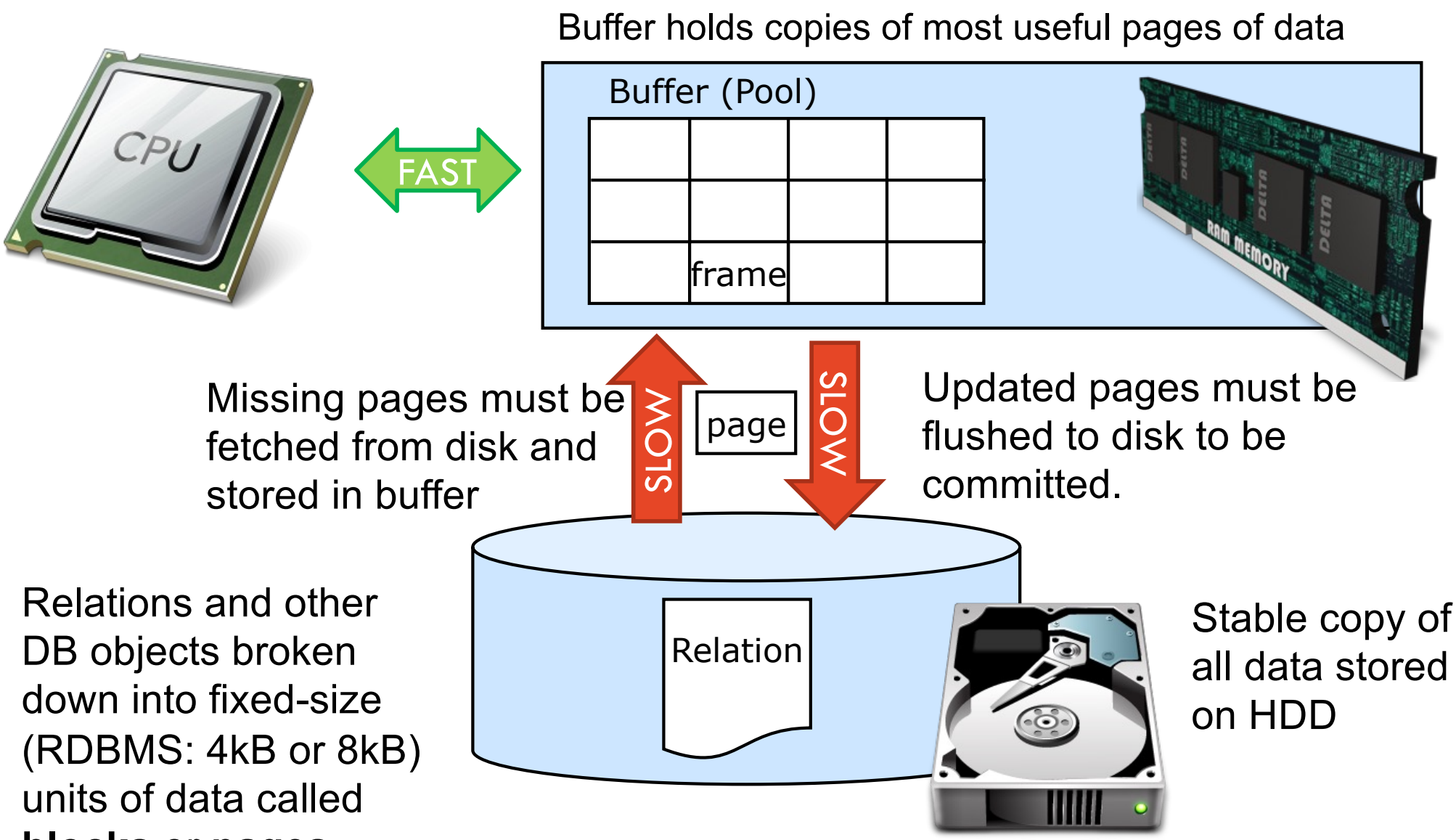
Accessing a Database Disk Page

- **Time to access (read/write) a disk block:**
  - seek time (moving arms to position disk head on track)
  - rotational delay (waiting for block to rotate under head)
  - transfer time (actually moving data to/from disk surface)
  - **Seek time and rotational delay dominate.**

- **Key to lower I/O cost: reduce the number of movements!**
  - Aim where possible that data one needs to access is on a page that was already brought into memory
  - Aim where possible to reduce the number of data items one must examine

Using a Buffer to Hide Access Gap

(as much as possible)



How to Store a Database?

- Logical Database Level:
  - A database is a collection of ***relations***. Each relation is a set of ***records*** (or *tuples*). *A* record is a sequence of ***fields*** *(or attributes*).
  - Example:

```
CREATE TABLE Student (
   id         INTEGER      PRIMARY KEY,
        name       VARCHAR(40) UNIQUE,
   address    VARCHAR(255),
   gender     CHAR(1),
   birthdate  DATE
);
```

- Physical Database Level:
  - How to represent tuples with several attributes (*fields*)?
  - How to represent collection of tuples and whole tables?
  - How do we find specific tuples?

Alternative File Organizations

Many alternatives exist, each ideal for some situations, and not so good in others:

- **Heap Files** – a record can be placed anywhere in the file where there is space (random order)
  - suitable when typical access is a *file scan* retrieving all records.
- **Sorted Files** – store records in sequential order, based on the value of the search key of each record
  - best if records must be retrieved in some order, or only a `*range*' of records is needed.
- **Indexes** – data structures to organize records via trees or hashing
  - like sorted files, they speed up *searches for a subset of records*, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.

(Unordered) Heap Files

- Simplest file structure contains records in no particular order.
- Access method is a ***linear scan***
  - In average half of the pages in a file must be read, in the worst case even the whole file
  - Efficient if all rows are returned (SELECT * FROM *table*)
  - Very inefficient if a *few* rows are requested

- Rows appended to end of file as they are inserted
  - Hence the file is unordered
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

Example: Transcript Stored as Heap File

666666	MGT123	F1994	4.0
123456	CS305	S1996	4.0
987654	CS305	F1995	2.0

page 0

717171	CS315	S1997	4.0
666666	EE101	S1998	3.0
765432	MAT123	S1996	2.0
515151	EE101	F1995	3.0

page 1

234567	CS305	S1999	4.0
878787	MGT123	S1996	3.0

page 2

Sorted File

- Rows are sorted based on some attribute(s)
  - Successive rows are stored in same (or successive) pages
- Access method could be a ***binary search***
  - Equality or range query based on that attribute has cost $log_2 B$ to retrieve page containing first row

- Problem: Maintaining sorted order
  - After the correct position for an insert has been determined, shifting of subsequent tuples necessary to make space (very expensive)
  - Hence sorted files typically are not used per-se by DBMS, but rather in form of index-organised (clustered) files

Example: Transcript as Sorted File

111111	MGT123	F1994	4.0
111111	CS305	S1996	4.0
123456	CS305	F1995	2.0

page 0

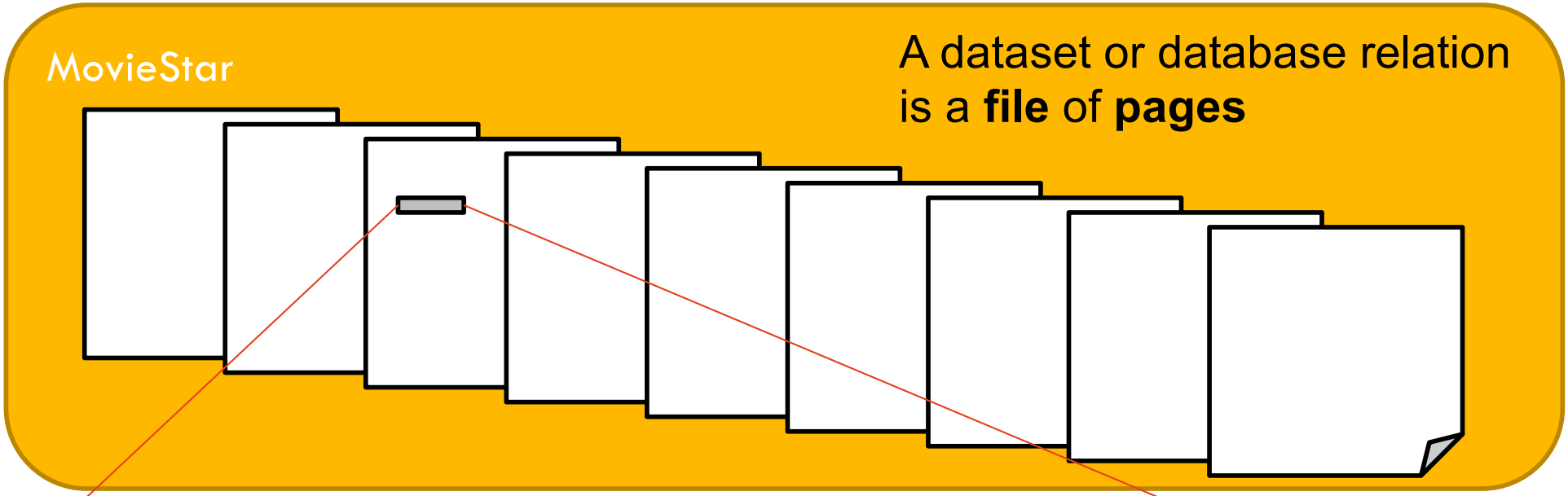
123456	CS315	S1997	4.0
123456	EE101	S1998	3.0
232323	MAT123	S1996	2.0
234567	EE101	F1995	3.0

page 1

234567	CS305	S1999	4.0
313131	MGT123	S1996	3.0

page 2

Heap Files vs Sorted Files



Brad Pert	3543 Long Drive	M	20/05/1968

Each page stores multiple records

	Heap File	Sorted File
Insert	Wherever there's space (or just at end of file)	After previous record according to key
Delete	Leave empty space	Leave empty space
Access	Linear Scan	Binary Search (if on key)

Indices

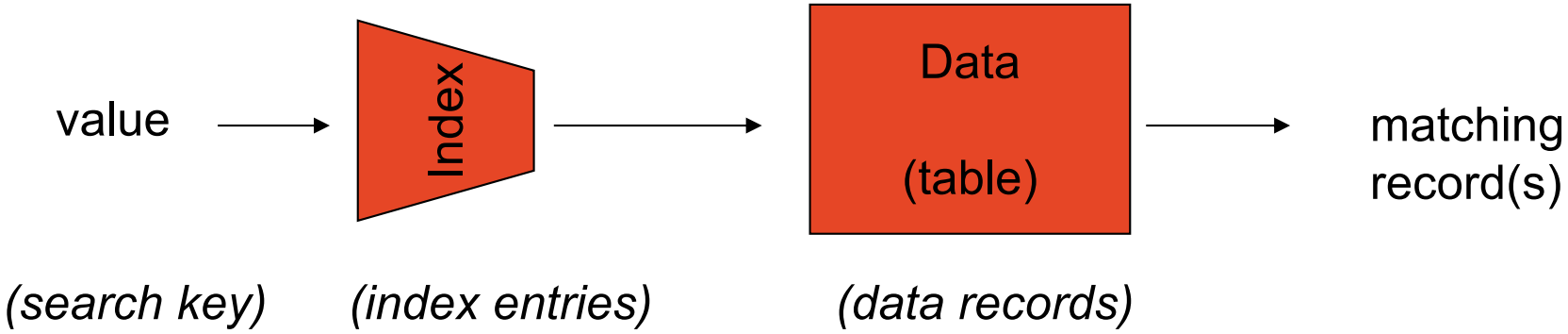
- Can we come up with a file organisation that is
  - as efficient for searches (especially on ranges) as an ordered file?
  - as flexible as a heap file for inserts and updates?

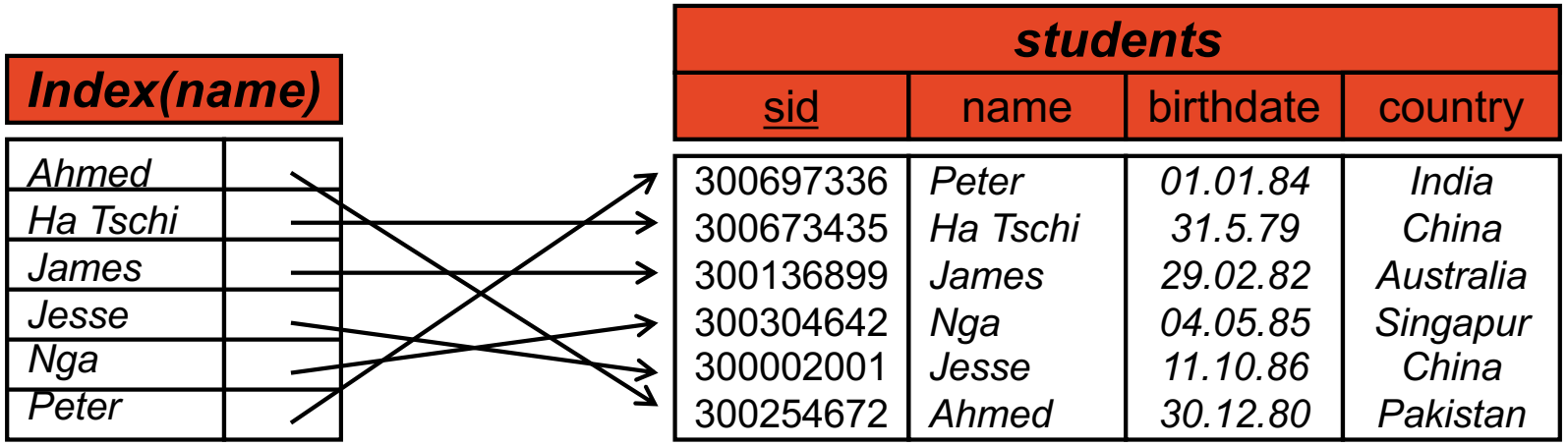
- Idea: Separate location mechanism from data storage
  - Just remember a book index:
    Index is a set of pages (a separate file) with pointers (page numbers) to the data page which contains the value

  - Instead of scanning through whole book (relation) each time,
    using the index is much faster to navigate (less data to search)

  - Index typically much smaller than the actual data

Index Example





- **Ordered index:** search keys are stored in sorted order
- **Hash index:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Definition in SQL

- Create an index

  CREATE INDEX *name* ON *relation-name* (<*attributelist*>)

  - Example:

    CREATE INDEX *StudentName* ON Student(*name*)

- Index on primary key generally created automatically
  - Use **CREATE UNIQUE INDEX** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  DROP INDEX *index-name*

- Sidenote: SQL-92 does actually not officially define commands for creation or deletion of indices.
  - vendors kind-of 'agreed' to use this syntax consistently

Indices - The Downside

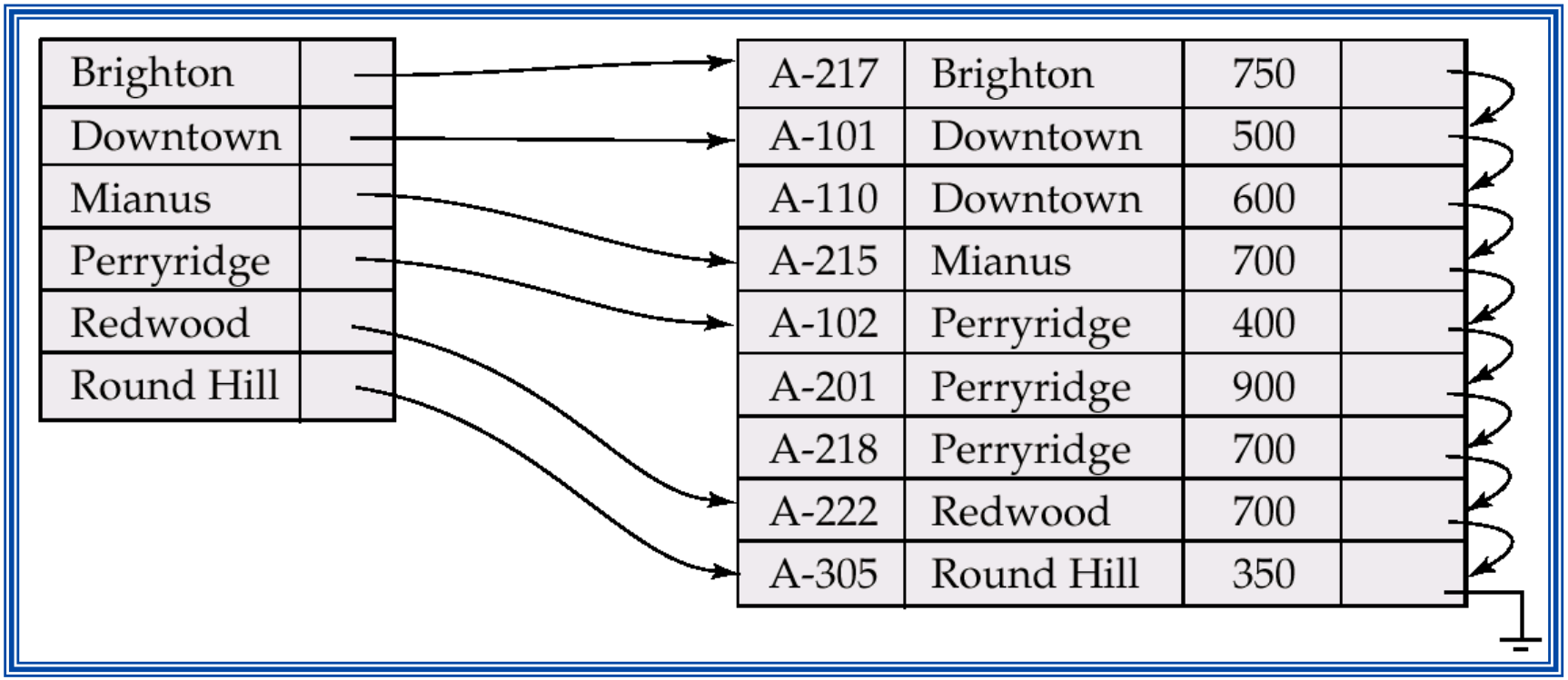
- Additional I/O to access index pages
  (except if index is small enough to fit in main memory)
  - The hope is that this is less than the saving through more efficient finding of data records
- Index must be updated when table is modified.
  - depends on index structure, but in general can become quite costly
  - so every additional index makes update slower…
- Decisions, decisions…
  - Index on primary key is generally created automatically
  - Other indices must be defined by DBA or user, through vendor specific statements
  - Choose which indices are worthwhile, based on workload of queries
    (cf. later this lecture)

Clustering Index

- index entries and rows are ordered in the same way
- The particular index structure (eg, hash, tree) dictates how the rows are organized in the storage structure
- There can be at most one clustering index on a table
  - e.g The white pages of the phone book in alphabetical order.
- CREATE TABLE statement generally creates an clustered index on primary key

Example: Clustering Index

- Clustered Index on **branch-name** field of **account**

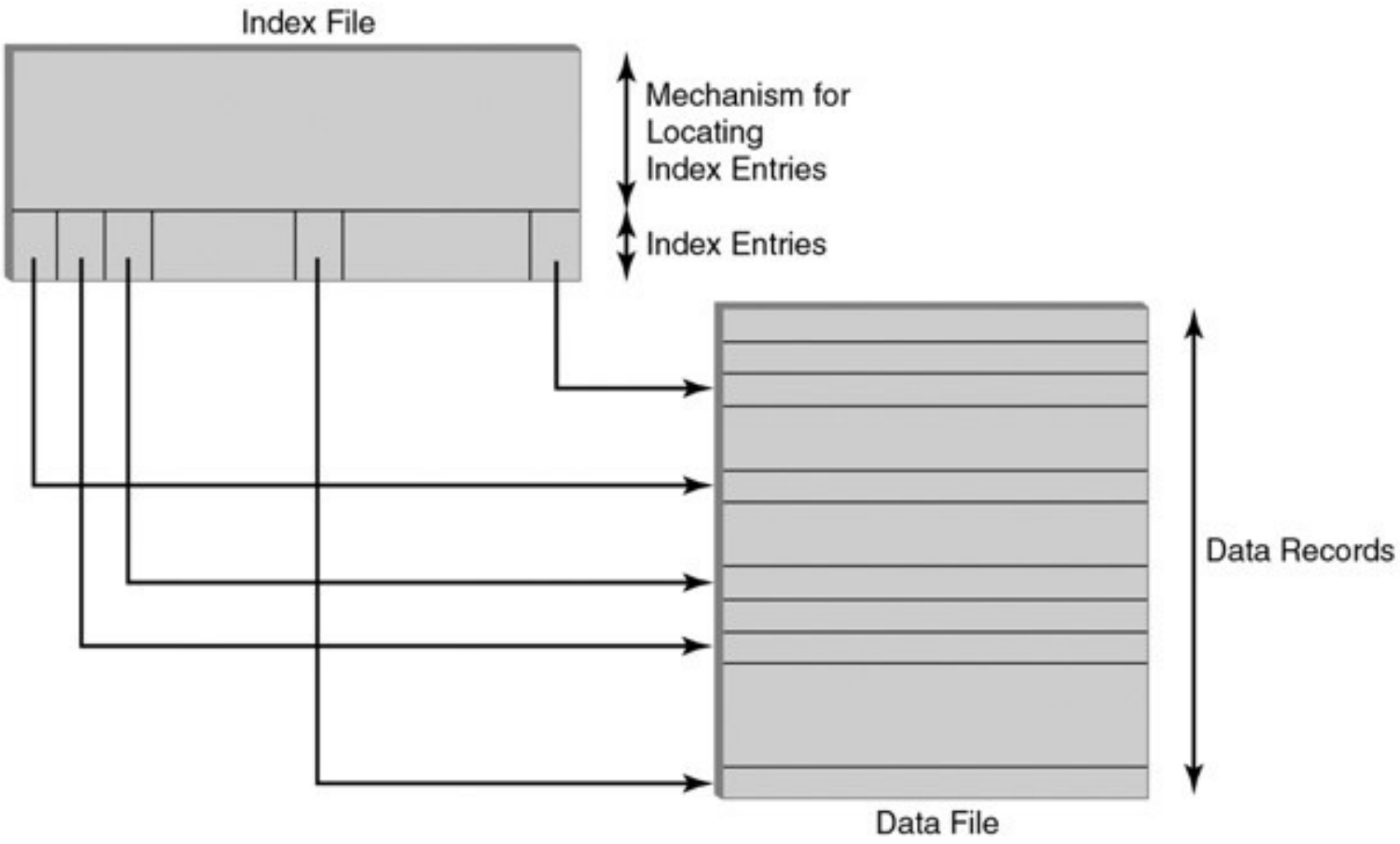


Source: Silberschatz/Korth/Sudarshan: *Database System Concepts*, 2002.

Unclustered Index

- Index entries and rows are not ordered in the same way
- There can be many secondary indices on a table
- Index created by CREATE INDEX is generally an unclustered, secondary index

Unclustered Index



Clustered vs. Unclustered Indices

- Clustered Index: Good for range searches over a range of search key values
  - Use index to locate first index entry at start of range
    - This locates first row.
  - Subsequent rows are stored in successive locations if index is clustered (not so if unclustered)
  - Minimizes page transfers and maximizes likelihood of cache hits

- Example: Access Costs of a Range Scan
  - Data file has 10,000 pages, 100 rows in search range
  - Page transfers for table rows (assume 20 rows/page):
    - Heap: 10,000 (entire file must be scanned)
    - File sorted on search key: log_2 10000 + (5 or 6) ≈ 19
    - Unclustered index: ≤ *index-height* + 100
    - Clustered index: *index-height* + (5 or 6)

Comparison

- *Clustered index*: index entries and rows are ordered in the same way
- There can be at most one clustered index on a table
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key
- Especially good for "range searches" (where search key is between two limits)
  - Use index to get to the first data row within the search range.
  - Subsequent matching data rows are stored in adjacent locations (many on each block)
  - This minimizes page transfers and maximizes likelihood of cache hits
- *Unclustered (secondary) index*: index entries and rows are not ordered in the same way
- There can be many unclustered indices on a table
  - As well as perhaps one clustered index
  - Index created by CREATE INDEX is generally an unclustered, secondary index
- Unclustered isn't ever as good as clustered, but may be necessary for attributes other than the primary key

Multicolumn Search Keys

- CREATE INDEX Inx ON Tbl (A*tt1*, A*tt2*)
- Search key is a *sequence* of attributes; index entries are lexically ordered
  - That is, the index entry for Att1 = X1, Att2 = X2 comes before the index entry for Att1 = Y1, Att2 = Y2 when either X1 < Y1 or (X1=Y1 and X2 < Y2)
- Supports finer granularity equality search:
  - "Find row with value (A1, A2) "
- Supports range search (tree index only):
  - "Find rows with values between (A1, A2) and (A1′, A2′) "
- Supports partial key searches (tree index only):
  - Find rows with values of A*tt1* between A1 and A1′
  - But not "Find rows with values of A*tt2* between A2 and A2′ "
- Especially useful when it covers a whole query (se elater)

Index Classification

- Primary Index vs. Secondary
  - If index entries contain actual data rows, then called **primary index**.
    - Oracle calls this *integrated storage structure* an 'index-organised table'
    - Note: Some literature refers to this as main index
  - Otherwise **secondary index**
- Unique vs. Non-Unique
  - an index over a candidate key is called a **unique index** (no duplicates)
- Single-Attribute vs. Multi-Attribute
  - whether the search key has one or multiple fields
- Clustered vs. Unclustered
  - If data records and index entries are ordered the same way, then called **clustering index**.

Indexing in the "Physical World"

- Library:

```
CREATE TABLE Library (
    callno  CHAR(20)  PRIMARY KEY,
    title      VARCHAR(255),
    author  VARCHAR(255),
    subject VARCHAR (128)
 )
```

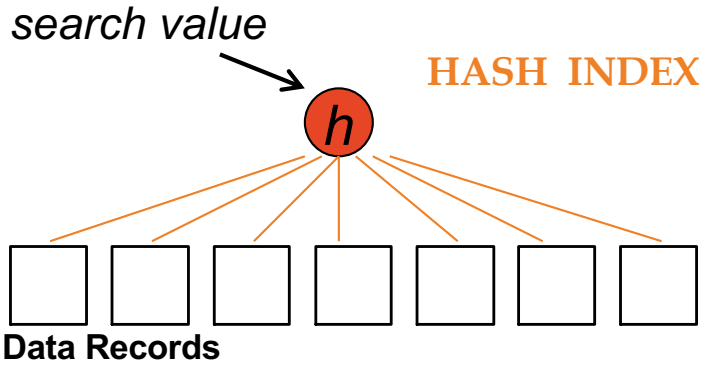
- Library stacks are "clustered" by call number.
  - However, we typically search by title, author, subject/keyword
- The catalog is a **secondary** index…say by Title

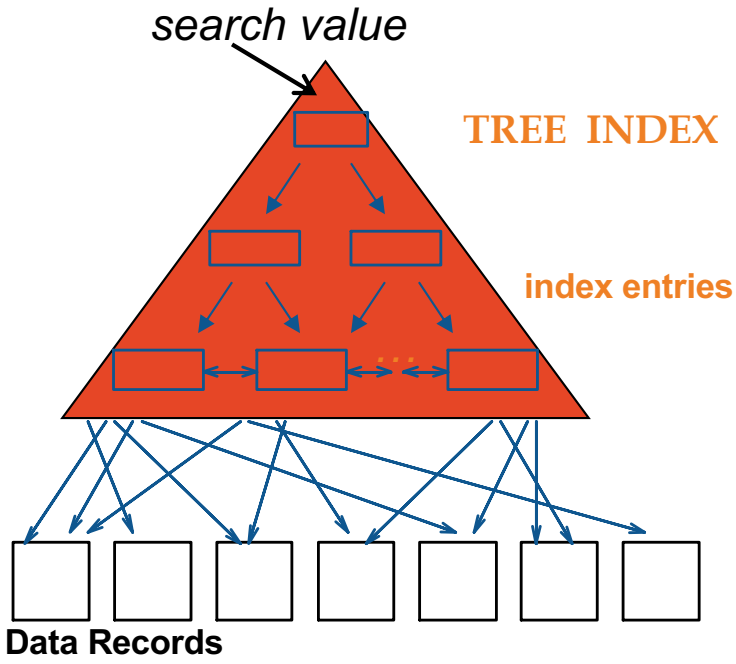
- **CREATE Index TitleCatalog on Library(title)**

Which Types of Indexes are available?

- Tree-based Indexes: B+-Tree
  - *Very flexible, only indexes to support point queries, range queries and prefix searches*
- Hash-based Indexes
  - *Fast for equality searches – but not other calculations*
- Special Indexes
  - Such as Bitmap Indexes for OLAP or R-Tree for spatial databases

Found in every database engine





=> More details on disk-based index structures in DATA3404

Multi-Level Tree Index

Most DBMS use B+-tree data structure

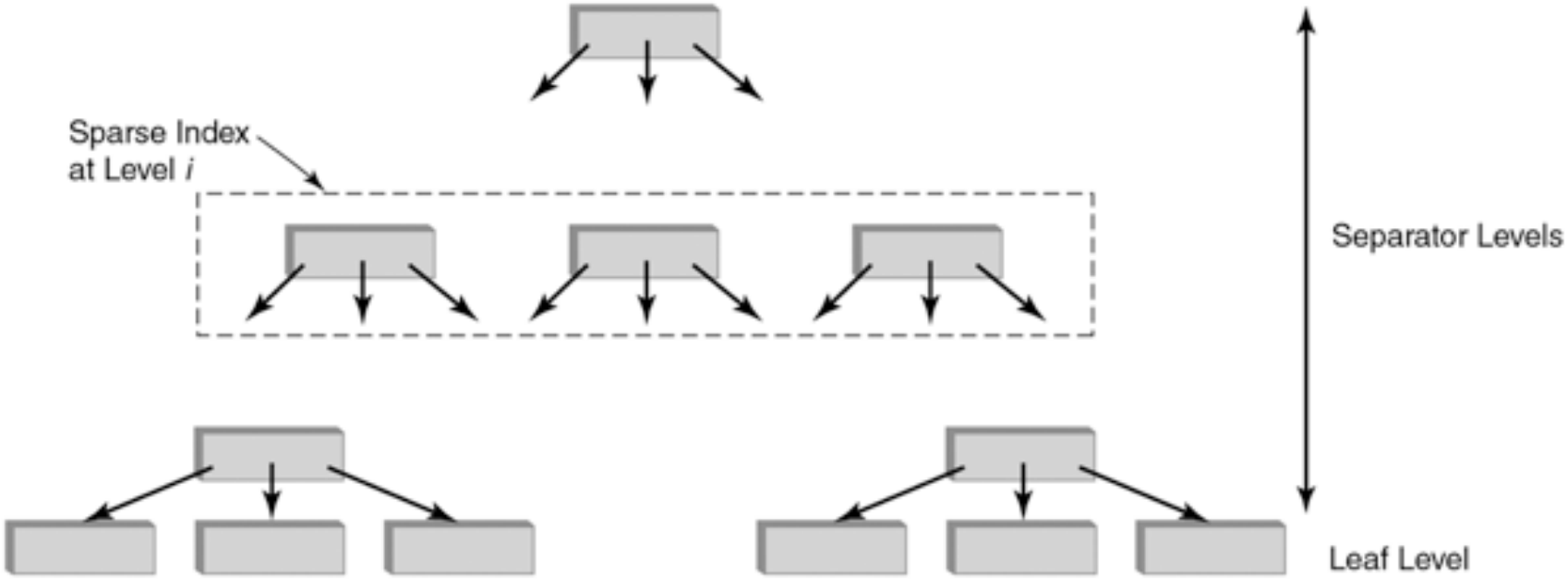


FIGURE 9.14 Schematic view of a multilevel index.

- Locate data records by descending the tree from root to leaf
  - Search cost to find pointer to data row(s) = number of levels in index tree
  - This is logarithmic in theory, but in practice can be considered as a small constant!
  - Typical index with 2 or 3 levels can support millions of data rows

Index that covers a query

- Goal: Is it possible to answer whole query just from an index?
- **Covering Index for a particular query** - an index that contains all attributes required to answer a given SQL query:
  - all attributes from the WHERE filter condition
  - if it is a grouping query, also all attributes from GROUP BY & HAVING
  - all attributes mentioned in the SELECT clause
- Typically a multi-attribute index

- Order of attributes is important: the attributes from the WHERE clause must form a prefix of the index search key (ie these attributes come first in the list of attributes which are used to build the index)
- Index on (Y,X) can answer "`SELECT X FROM table WHERE Y=const`" *without needing to access the data records themselves*
  - *But index on (X,Y) does not cover this query*

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Choices of Indexes

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash or Tree?
- How should the indexes be created?
  - Separate tablespace? Own disk?
  - Fillfactor for index nodes?

Choices of Indexes (cont'd)

- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - For now, we discuss simple 1-table queries.

- 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.

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query only supported by tree index types.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable **index-only** strategies for important queries (so-called *covering index*).
    - For index-only strategies, clustering is not important!
- 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.
- Create indexes in own tablespace on separate disks

Choosing an Index

- An index should support a query of the application that has a significant impact on performance
  - Choice based on frequency of invocation, execution time, acquired locks, table size

- Example 1:

```
SELECT E.Id
     FROM Employee E
    WHERE E.Salary < :upper  AND  E.Salary > :lower
```

  - This is a range search on *Salary*.
  - Since the primary key is *Id*, it is likely that there is a clustered, main index on that attribute; however that index is of no use for this query.
  - Choose a B+ tree index with search key *Salary*

Choosing an Index (cont'd)

- Example 2:

```
SELECT T.studId
  FROM Transcript T
 WHERE T.grade = :grade
```

- This is an equality search on *grade*.
- We know the primary key is *(studId, semester, uosCode)*
  - It is likely that there is a main, clustered index on these PK attributes
  - but it is of no use for this query…
- Hence: Choose a B+ tree index (or hash index) with search key *Grade*
  - Again: a *covering* index with composite search key *(grade, studId)* would allow to answer complete query out of index
    - but then only as B-Tree index…

Choosing an Index (cont'd)

- Example 3:

```
SELECT T.uosCode, COUNT(*)
  FROM Transcript T
 WHERE T.year = 2009 AND T.semester = 'Sem1'
 GROUP BY T.uosCode
```

- This is a group-by query with an equality search on *year* and *semester*.
- If the primary key is (*studId, year, semester, uosCode*), it is likely that there is a clustered index on these sequence of attributes
  - But the search condition is on *year* and *semester* => must be prefix!
  - Hence PK index not of use
  - Create a Covering INDEX: (*year, semester, uosCode*)

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 9 (9.1-9.4)
  - Chapter 12 (database tuning)
  - *Kifer/Bernstein/Lewis gives a good overview of indexing and especially on how to use them for database tuning. This is the focus for ISYS2120 too.*
- Ramakrishnan/Gehrke (3rd edition - the 'Cow' book)
  - Chapter 8
  - *The Ramakrishnan/Gehrke is very technical on this topic, providing a lot of insight into how disk-based indexes are implemented. We only need the overview here (Chap8); technical details are covered in data3404.*
- Ullman/Widom (3rd edition - '1st Course in Databases')
  - Chapter 8 (8.3 onwards)
  - *Mostly overview, but cost model of indexing goes further than we discuss here in the lecture*

- Learn much more: study DATA3404