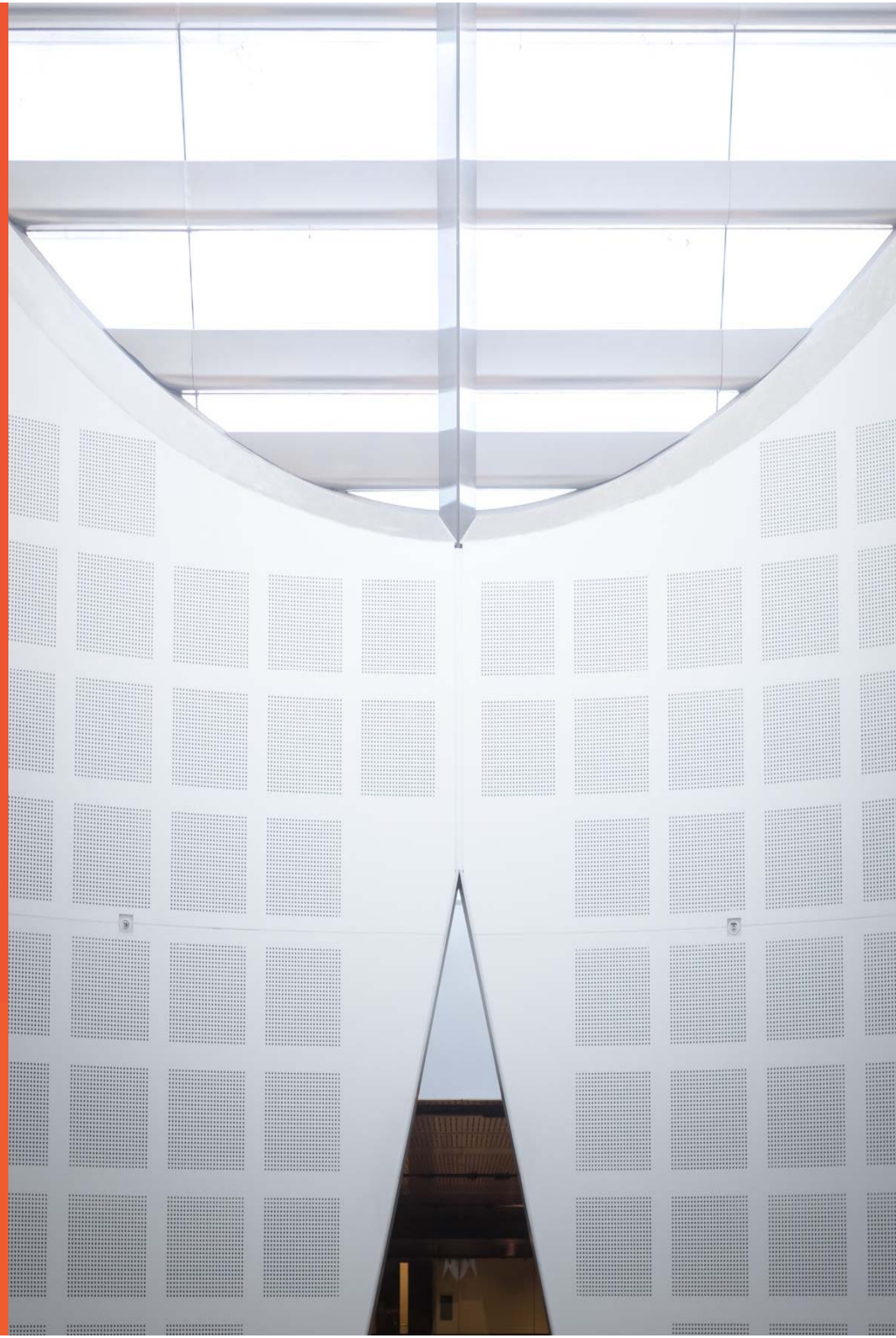
DATA2001: Data Science: Big Data and Data Diversity

W5: Scalable Analytics: The Role of Indexes and Data Partitioning

Presented by
Dr. Matloob Khushi
School of IT

THE UNIVERSITY OF SYDNEY



Quiz 1 – Tuesday 1st May

- Quiz will be held during the tutorial time
- No lecture on Wednesday 25 April (ANZAC day)
- Introduction to SQL vial GROK Learning is live now
- https://groklearning.com/learn/usyd-data2001-2018-s1/
- Additional Python practice via online course of our University
  - OLEO1306 - Beginner Programming for Data Analysis introduces Python programming for data analytics. https://canvas.sydney.edu.au/courses/4545/pages/list-of-zero-credit-point-ole-units

Overview of the last two weeks

- Week 3
  - Accessing data in relational databases
  - Basic RDBMS design (Primary key, Foreign key)
  - Querying and Summarising data with SQL

- Week 4
  - Cleaning data with Python
  - Upload data in a relational database using Python
  - More SQL, joins and sub-queries
  - Querying and Summarising data with SQL and Python

This Week's Learning Objectives

- **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…
  - Awareness of the trade-off between query performance and indexing costs (updates)

- **Distributed Databases**
  - Architectures
  - Data Partitioning / Sharding

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

Motivation: Disk Storage

- DBMS stores information on disks.
  - *Main memory is much more expensive than HDDs.*
    A$100 will buy you either 8 GB of RAM or 1 TB of disk today.
  - *Main memory is volatile.*
    We want data to be saved between runs. (Obviously!)

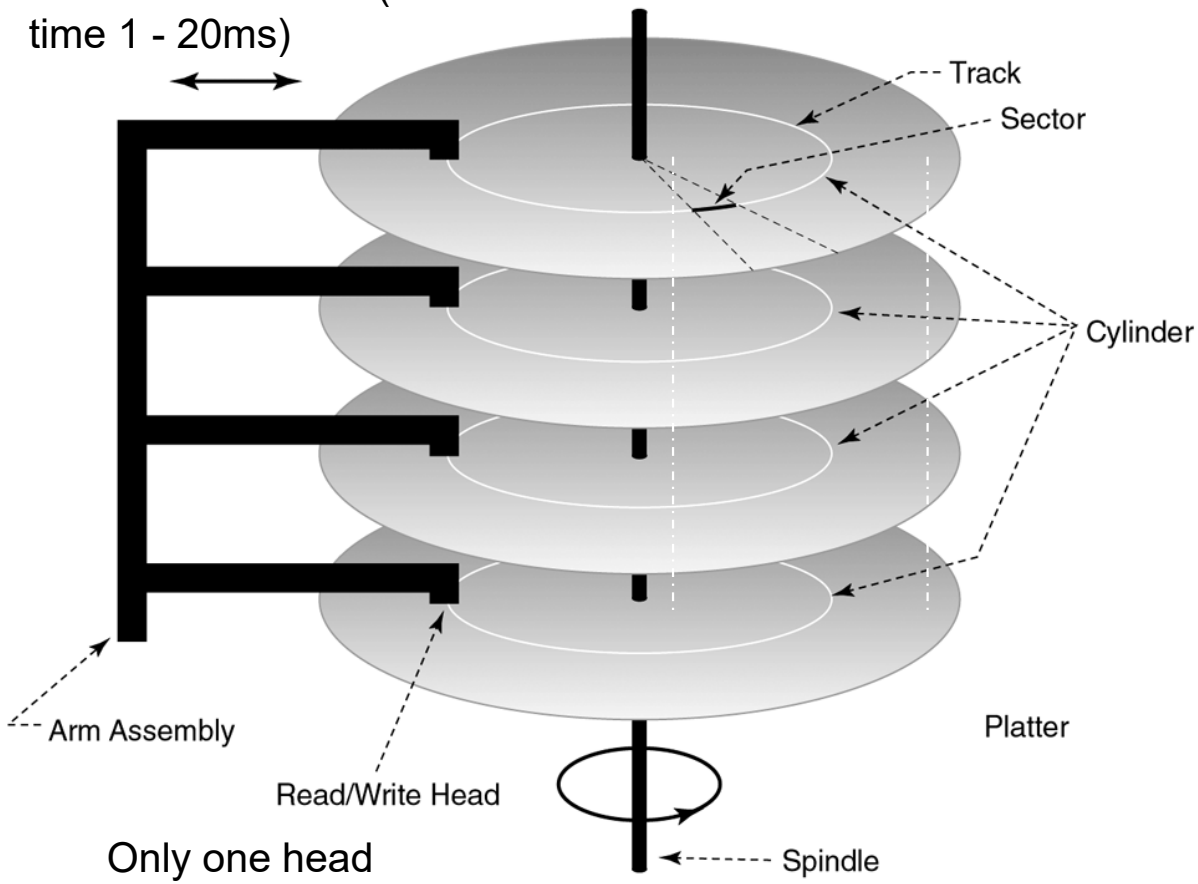
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
  - Indeed, overall performance is determined largely by 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



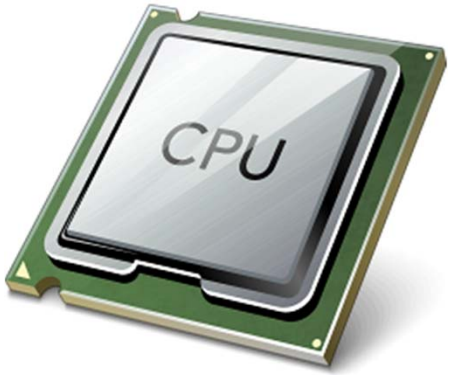
Only one head reads/writes at any one time.

Platters spin ~ 7200 rpm

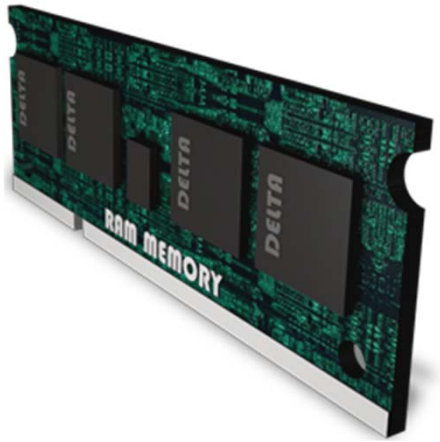
Where is Data Stored?

Main Memory (RAM):
- Expensive
- Volatile

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



FAST



SLOW

Tertiary Storage (e.g. Tape
- Very Cheap
- 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

Storage Hierarchy

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

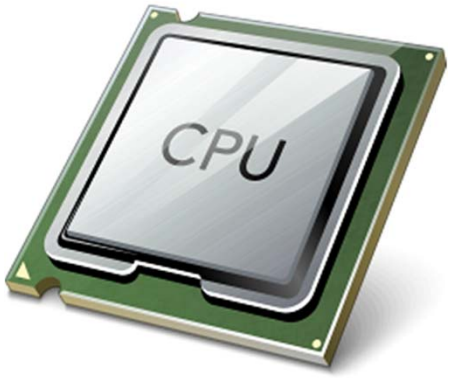
Accessing a Database Disk Page

- **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)

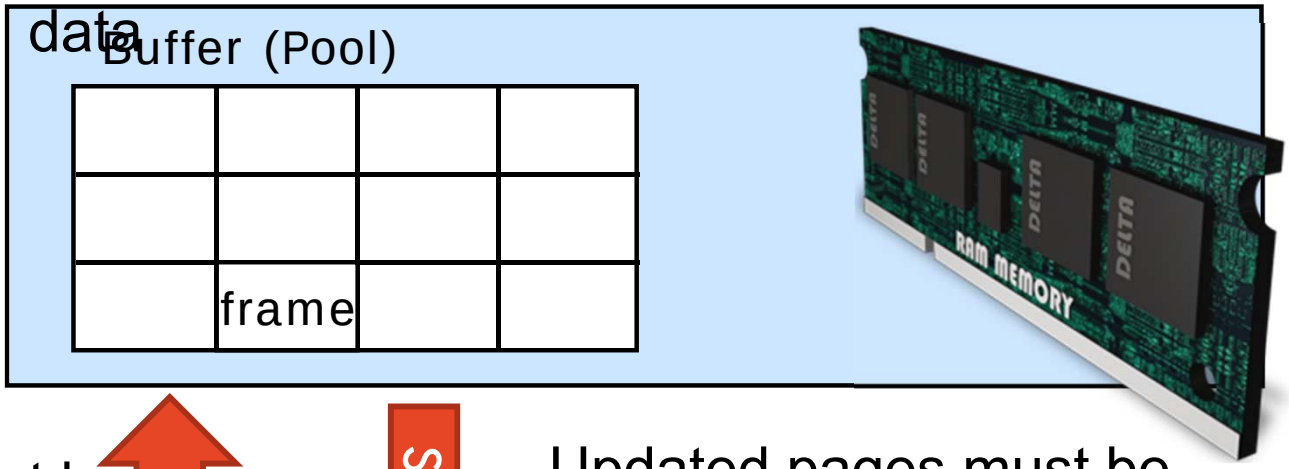
- **Seek time and rotational delay dominate.**
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page

- **Key to lower I/O cost: reduce seek/rotation delays!**

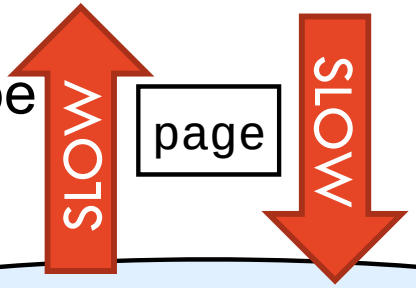
Using a Buffer to Hide Access Gap
(as much as possible)



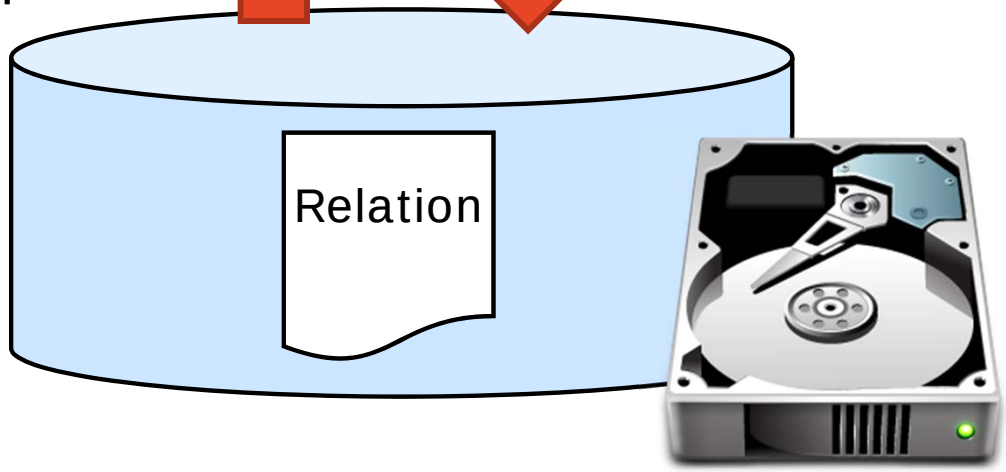
Buffer holds copies of most useful pages of data



Missing pages must be fetched from disk and stored in buffer



Updated pages must be flushed to disk to be committed.



Relations and other DB objects broken down into fixed-size (RDBMS: 4kB or 8kB) units of data called **blocks** or **pages**

Stable copy of all data stored on HDD

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

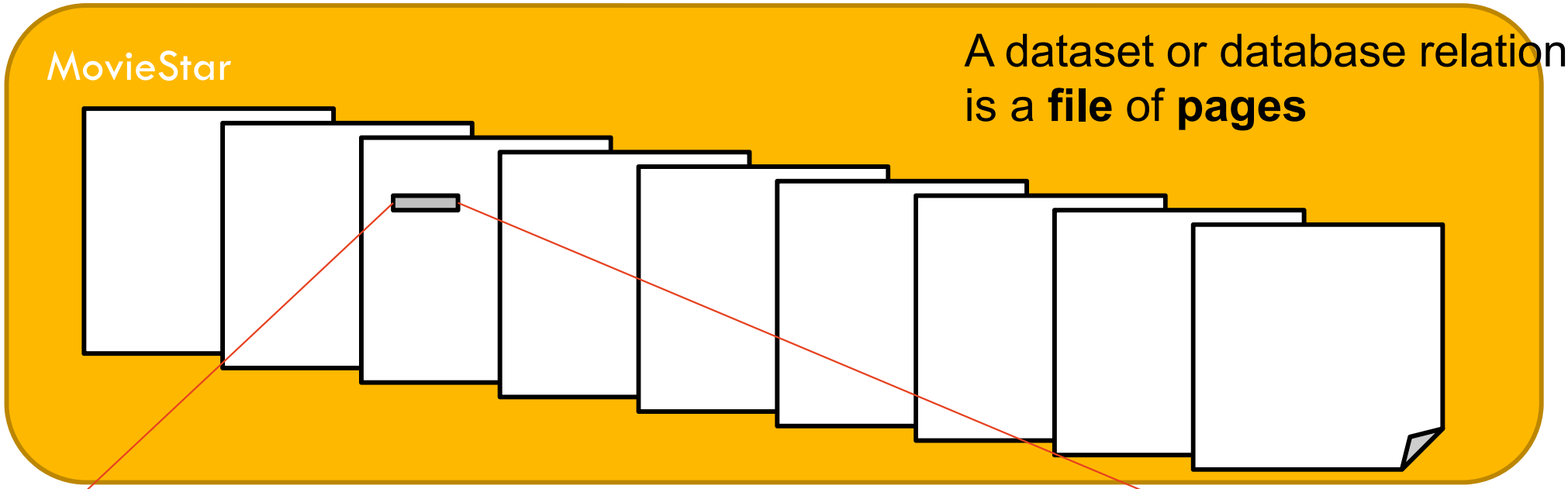
123456	CS315	S1997	4.0
123456	EE101	S1998	3.0
232323	MAT123	S1996	2.0
234567	EE101	F1995	3.0

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

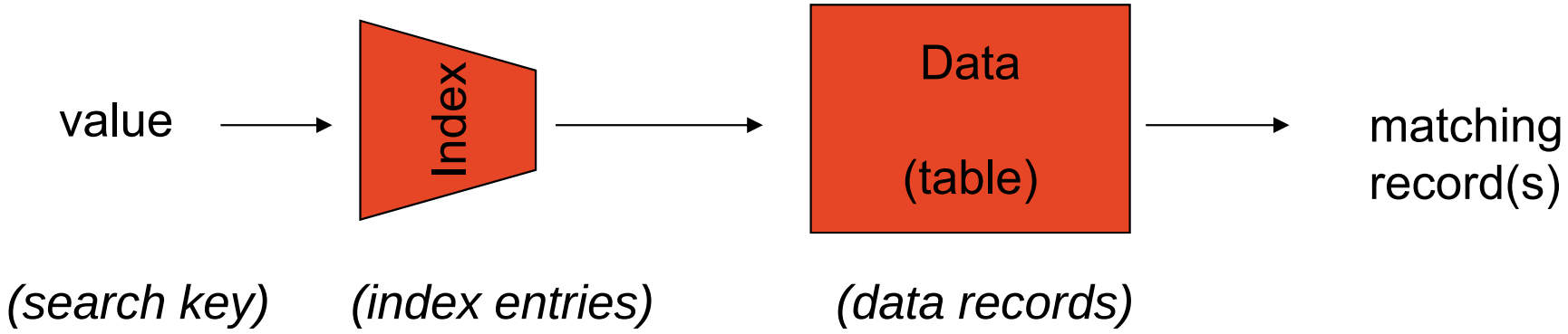
- Can we come up with a file organisation that is
  - as efficient for searches (especially on ranges) as an ordered file?
  - as flexible as a heap file for inserts and updates?

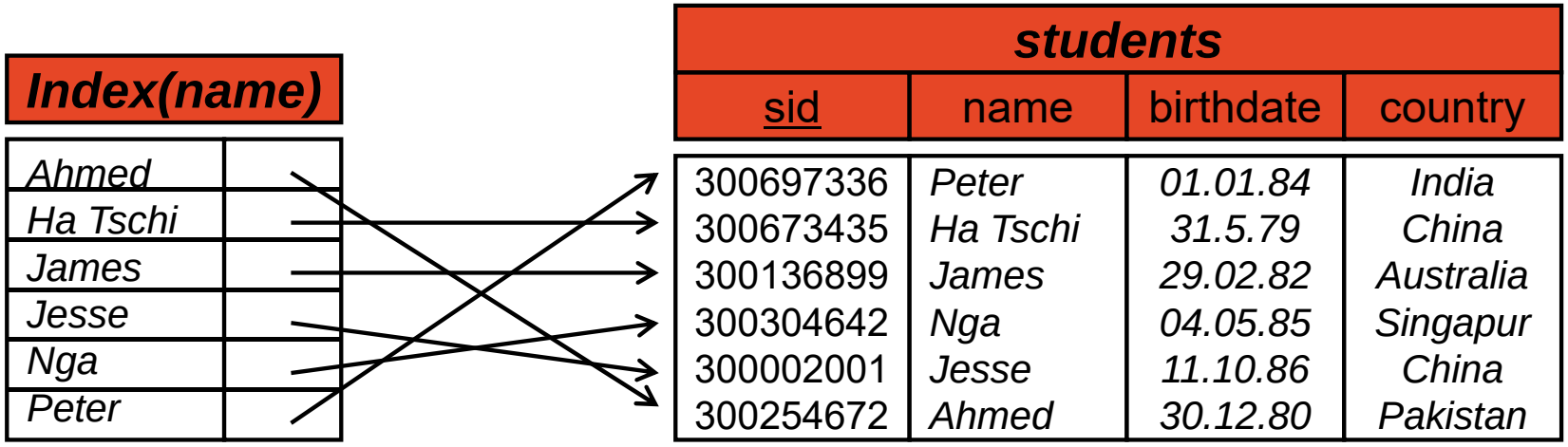
- Idea: Separate location mechanism from data storage
  - Just remember a book index:
    Index is a set of pages (a separate file) with pointers (page numbers) to the data page which contains the value

  - 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…

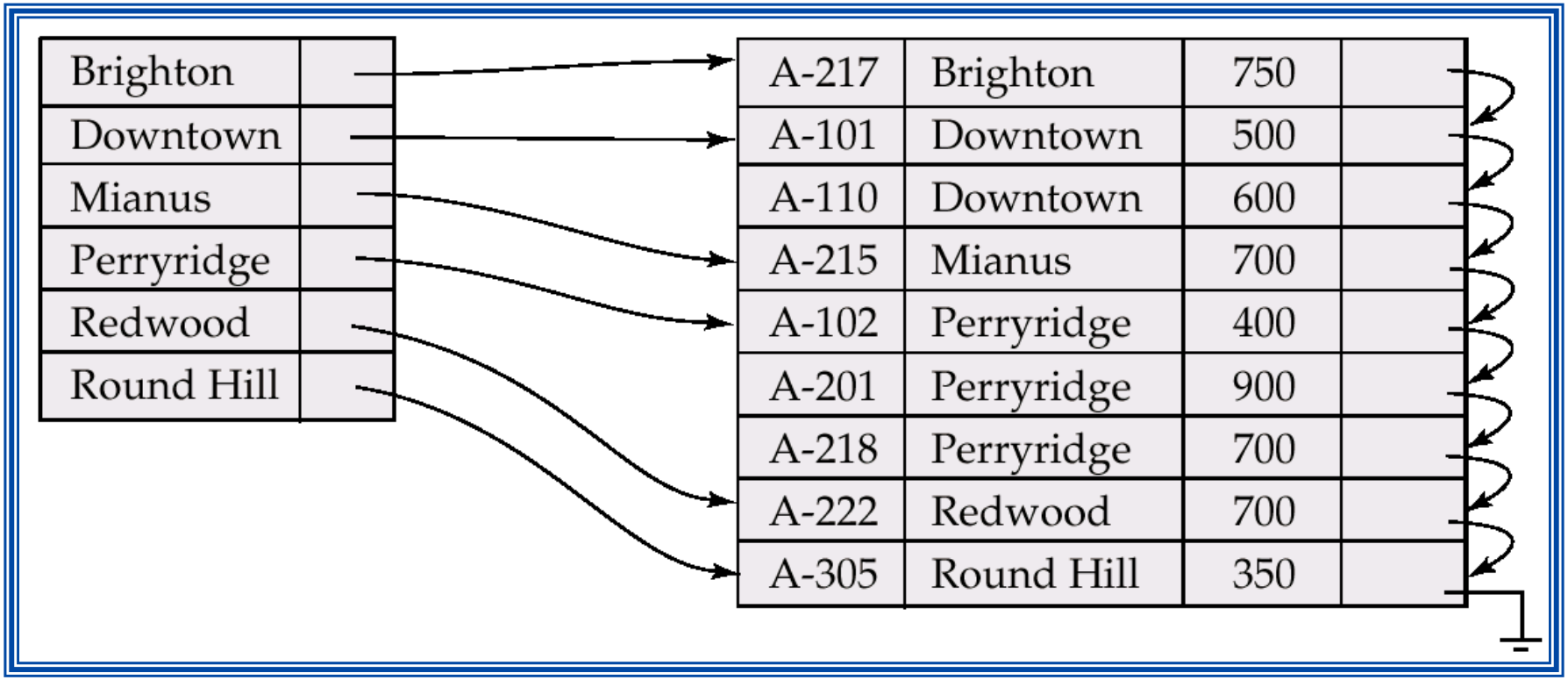
- Decisions, decisions…
  - Index on primary key is generally created automatically
  - Other indices must be defined by DBA or user, through vendor specific statements
  - Choose which indices are worthwhile, based on workload of queries
    (cf. later this lecture)

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**

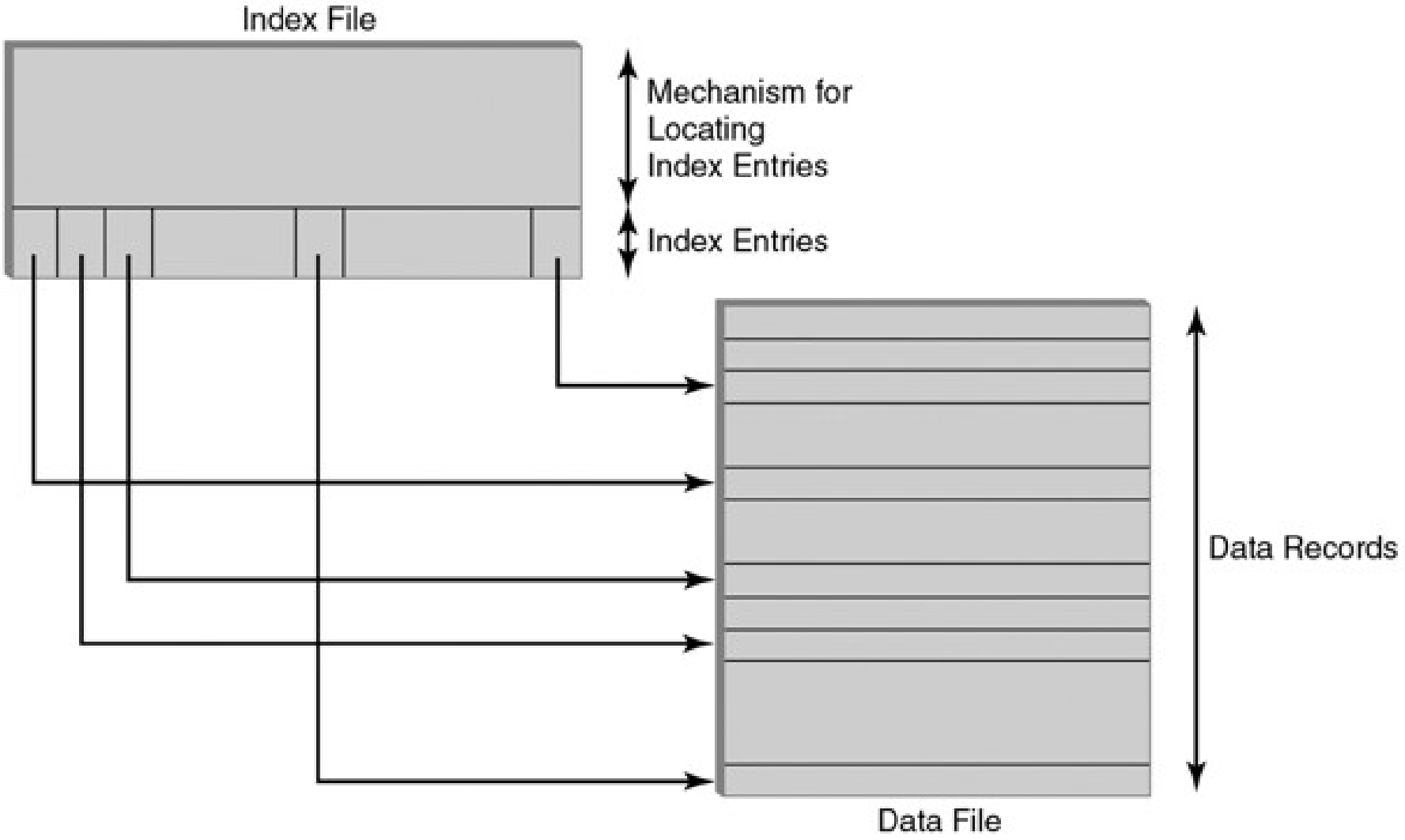


Source: Silberschatz/Korth/Sudarshan: *Database System Concepts*, 2002.

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

Indexing in the "Physical World"

- Library:

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

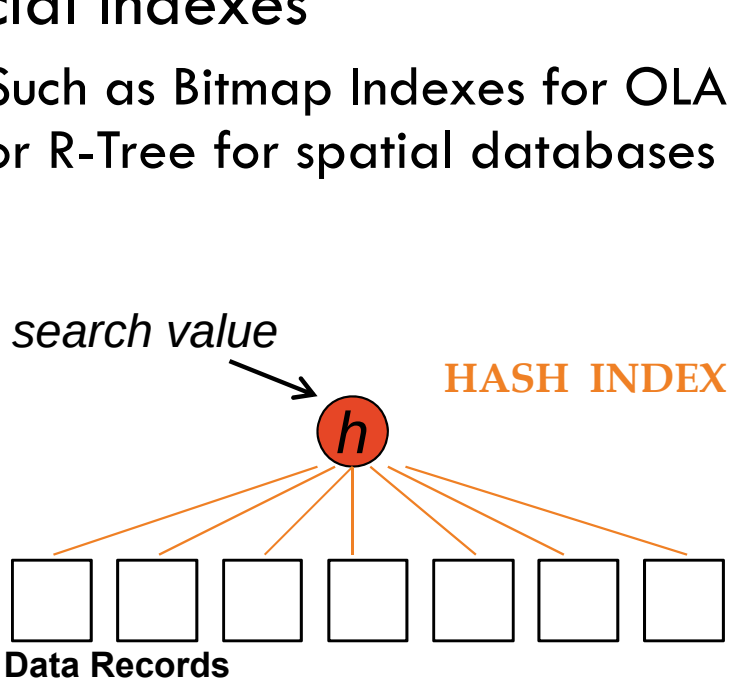
- Library stacks are "clustered" by call number.
  - However, we typically search by title, author, subject/keyword
- The catalog is a **secondary** index…say by Title

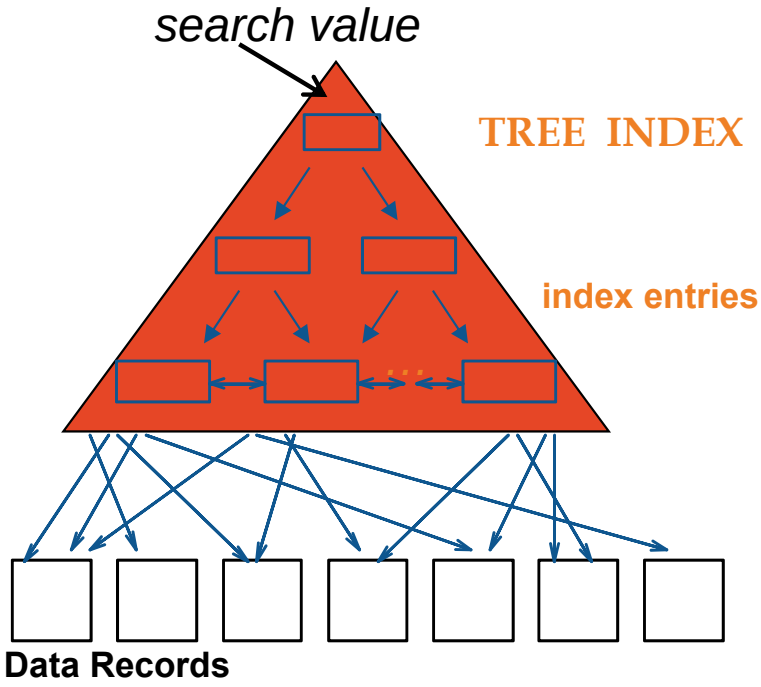
- **CREATE Index TitleCatalog on Library(title)**

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 - and that's it*
- 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 INFO3404

Multi-Level Tree Index

Most DBMS use B+-tree data structure

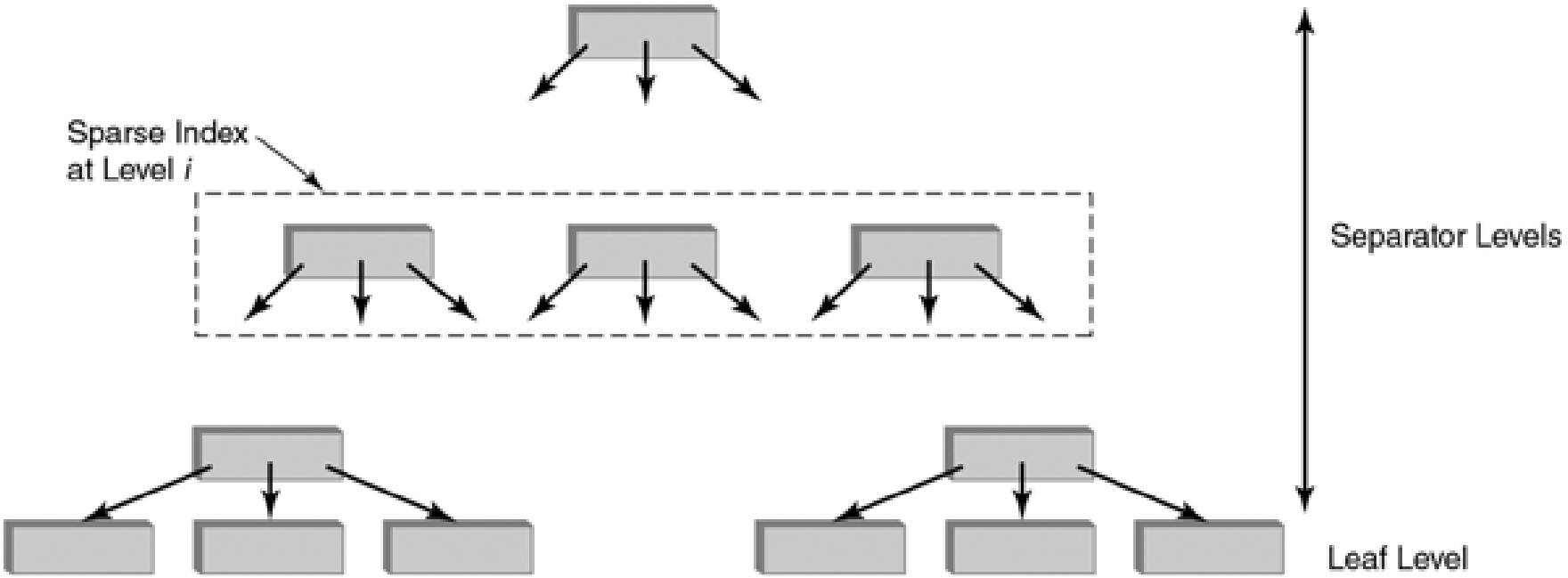


FIGURE 9.14 Schematic view of a multilevel index.

- 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

Covering Index

- Goal: Is it possible to answer whole query just from an index?
- **Covering Index** - 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: Prefix of the search key must be the attributes from the WHERE

DBMS Comparison: Index Types

	DB2 UDB 8.2	Oracle 10g	SQLServer 2008	Sybase ASE 12.5	Postgres 9	MySQL 5	Tera Data
B+-Tree	yes	yes	yes	yes	yes	yes	---
Hash Index	---	yes	---	---	yes	MEMORY tables	yes
Bitmap Index	(yes) (called EVI)	yes (since v8.1)	*Bitmap filter* ---	yes (in Adaptive IQ)	*bitmap scan* (since v8.1)	---	yes (join indexes)
Specialities	*R-Tree(*)*	*R-Tree (*)*	Quad Tree; fulltext index	---	Inverted idx; GiST	Fulltext *R-Tree*	---
Integrated (Main) Index	---	yes	yes	yes	---	InnoDB (always PK)	yes (hash table)
Clustered Index	yes	yes (only as so-called index-organised table)	yes (every clustered index is an integrated index)	yes (a clustered index is an integrated index)	yes	InnoDB (always PK)	Primary Index
Unique Index	yes	yes	yes	yes	yes	yes	yes
Multi-Column Index	yes	yes	yes	yes	yes	yes	yes

(*) spatial index via extension module

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 that 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:

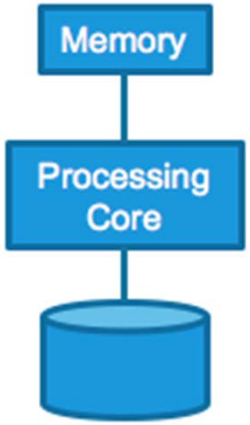
```
SELECT T.uosCode, COUNT(*)
  FROM Transcript T
 WHERE T.year = 2009 AND T.semester = 'Sem1'
 GROUP BY T.uosCode
```

- 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
  - Covering INDEX: *(year, semester, uosCode)*

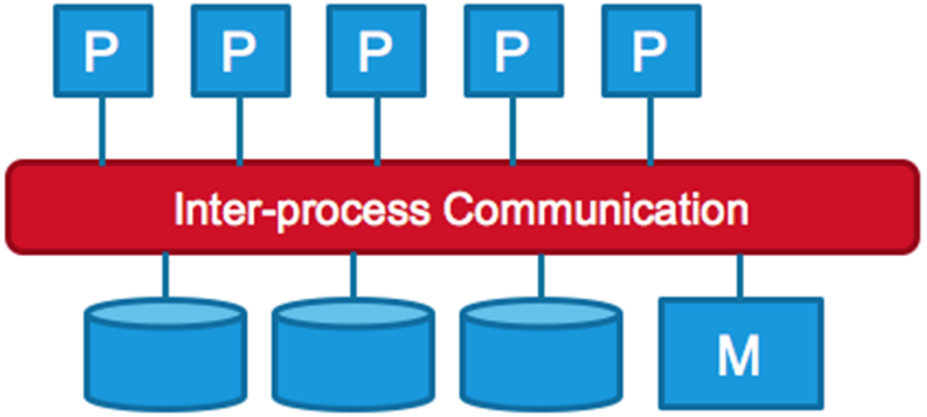
Distributed Databases

Parallel DBMS Architectures

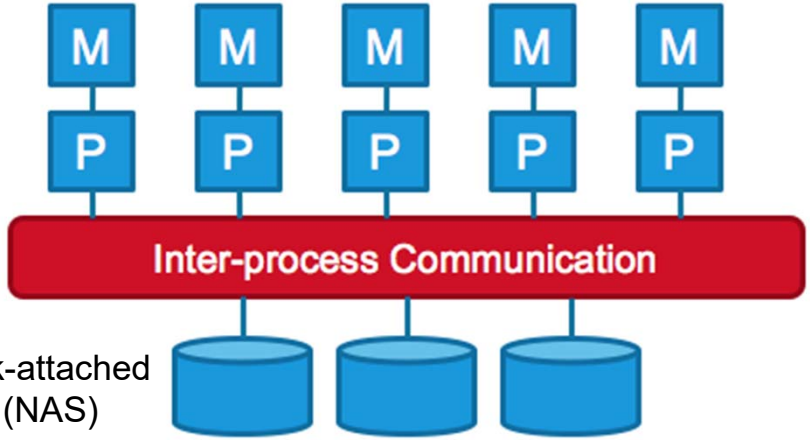
stand alone



shared-memory

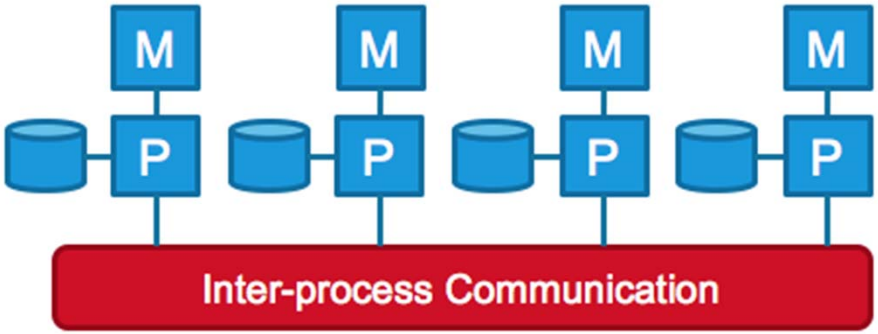


shared-disk



Network-attached storage (NAS)

shared-nothing



Preferred modal these day

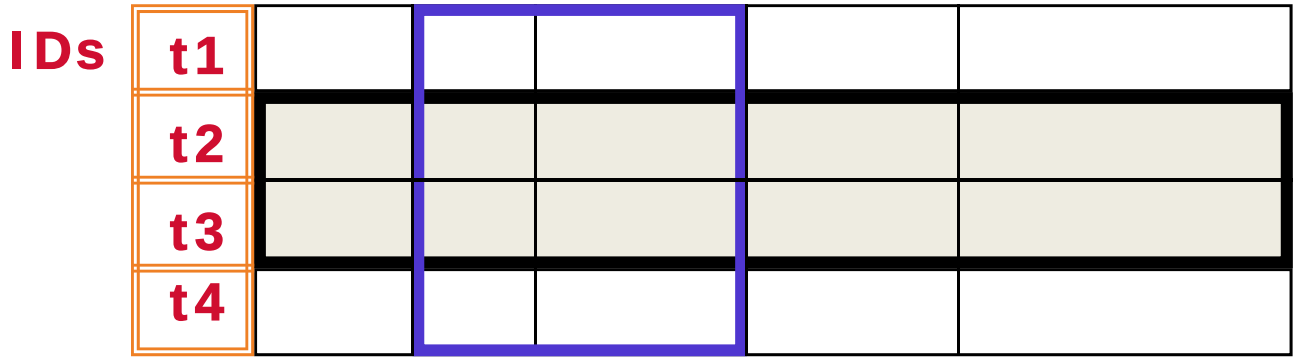
Distributed Data Management

- Two main physical design techniques:
- **Data Partitioning**
  - Storing sub-sets of the original data set at different places
    - can be in different tables in schema on same server, or at remote sites
  - Goal is to query smaller data sets & to gain scalability by parallelism
  - Sub-sets can be defined by
    - columns: ***Vertical Partitioning***
    - rows: ***Horizontal Partitioning***
      (if each partition is stored on a different site also called ***Sharding***)

- **Data Replication** (Not covered in this unit of study)
  - Storing copies ('replicas') of the same data at more than one place
  - Goal is fail safety / availability

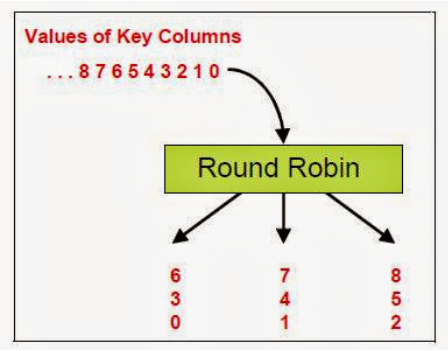
Data Partitioning

- **What** is partitioned?
  - **Horizontal Partitioning:** set of rows (known as a *shard*)
  - **Vertical Partitioning:** set of columns

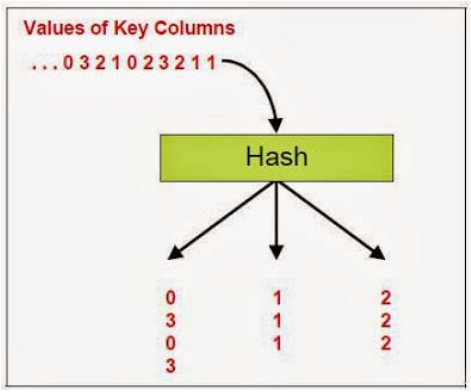


How to place data into partitions?

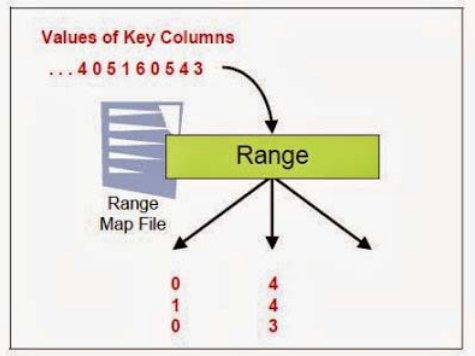
- **round robin**
  - Placement of partitions is going through all nodes in rounds



- **range partitioning**
  - Each node stores a partitioned defined by a range predicate



- **hash partitioning**
  - Target node is determined by a hash function on the tuple id or key



Example (Worksheet time)

Imagine we have this *account* table.
How would we horizontal/vertical partition into two tables?

ID	account_number	branch_name	balance
1	A-305	Hillside	500
2	A-402	Valleyview	10000
3	A-639	Valleyview	750
4	A-408	Valleyview	1123
5	A-226	Hillside	336
6	A-177	Valleyview	205
7	A-155	Hillside	62

Horizontal Range Partitioning of *account* Relation

ID	*account_number*	*branch_name*	*balance*
1	A-305	Hillside	500
5	A-226	Hillside	336
7	A-155	Hillside	62

$$account_1 = \sigma_{branch\_name="Hillside"}(account)$$

ID	*account_number*	*branch_name*	*balance*
2	A-177	Valleyview	205
3	A-402	Valleyview	10000
4	A-408	Valleyview	1123
6	A-639	Valleyview	750

$$account_2 = \sigma_{branch\_name="Valleyview"}(account)$$

Vertical Partitioning of *account* Relation

ID	account_number	balance
1	A-305	500
2	A-402	10000
3	A-639	750
4	A-408	1123
5	A-226	336
6	A-177	205
7	A-155	62

ID	branch_name
1	Hillside
2	Valleyview
3	Valleyview
4	Valleyview
5	Hillside
6	Valleyview
7	Hillside

$Account_1' = \Pi_{id,\ account\_number,\ balance}(account)$

$Account_2' = \Pi_{id,\ branch\_name}(account)$

Partitioning and Data Sharding

- Advantages of Partitioning for parallel databases:
  - Easier to manage than a large table
  - Better **availability**:
    if one partition is down, others are unaffected **if** stored on different tablespace / disk
  - Helps with bulk loading, e.g for data warehouse applications
  - Queries faster on smaller partitions; can be evaluated in parallel

- **Data Sharing (Fragmenting)**: Distributing data partitions over several sites in a distributed database
  - Assumes queries only access one shard only
  - Otherwise, counter-productive (eg. if we need to join two partitioned tables which are no *co-located* per site…
    - Co-located: join tuples are on the same node, e.g. by partitioning both tables on the join attributes

Updating Distributed Data

- Data Partitioning: Updates handled same as queries - just go to the sites which hold affected data;
  - If more than one site, an atomic commit protocol is needed
- Data Replication: Several Design Choices
  - not covered in this UoS

Lessons Learned this Week

- **Physical Data Organisation**
  - Core Problem: Persistency requires disks, but those are SLOW
  - Several physical design alternatives possible for same logical schema

- **Understanding of the concept of an Index**
  - Efficient access to single tuples or even ranges based on *search keys*
  - One *primary*, but several *secondary indices* possible per relation
  - single- vs. *multi-attribute indices*, *clustered* vs. *unclustered indices*

- **Practical experience with indexing a relational database**
  - How to suggest appropriate indexes for a given SQL workload
  - Awareness of the trade-off between SQL query performance and indexing costs (updates)

- **Distributed Databases**
  - Data Partitioning

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 INFO2120 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 info3404.*
- 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*

- [Oracle 10g Database Concepts, Chap. 5.4]
  Oracle Corporation: Oracle 10g Documentation, *Database Concepts*.