Database internals (indexing, etc)

ISYS2120 Data and Information Management

Prof Alan Fekete

University of Sydney

Acknowledge: slides from Uwe Roehm and Alan Fekete, and from the materials associated with reference books (c) McGraw-Hill, Pearson

Agenda

- **Motivation**
- **Database Internals: Overview**
  - Software
  - Hardware
  - Storage Layer: Physical Data Organisation
- **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)

Motivation

- The performance of queries can be changed greatly, depending on how the data is physically arranged
  - Cf comp2123 (linked list, hashmap, search tree, etc)
- SQL provides commands that allow DBA or data owner to alter the physical arrangement of the data
  - Especially CREATE INDEX command
  - Doing this wisely can often improve time to run a query from hours to seconds!
- This lecture aims to help you understand some of the simpler cases like this, and make a good choice for which index command to propose

Agenda

- **Motivation**
- **Database Internals: Overview**
  - Software
  - Hardware
  - Storage Layer: Physical Data Organisation
- **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)

The DBMS Platform: Overview

- The dbms is a large, complex piece of software, conceptually structured with many components for different purposes
  - Often though the dbms runs as a single process, in which various parts of the code are executed at different times, while requests are processed
  - Some platforms are structured as server, taking requests from application code and replying with results; others are embedded, that is code of the dbms is combined with application code in a single executable

Simple view of Processing a SQL query

- Parse the SQL
- Produce a “plan” with a sequence of operations that will calculate the result
- Execute the plan, by accessing various tables in specific ways
- Return the result

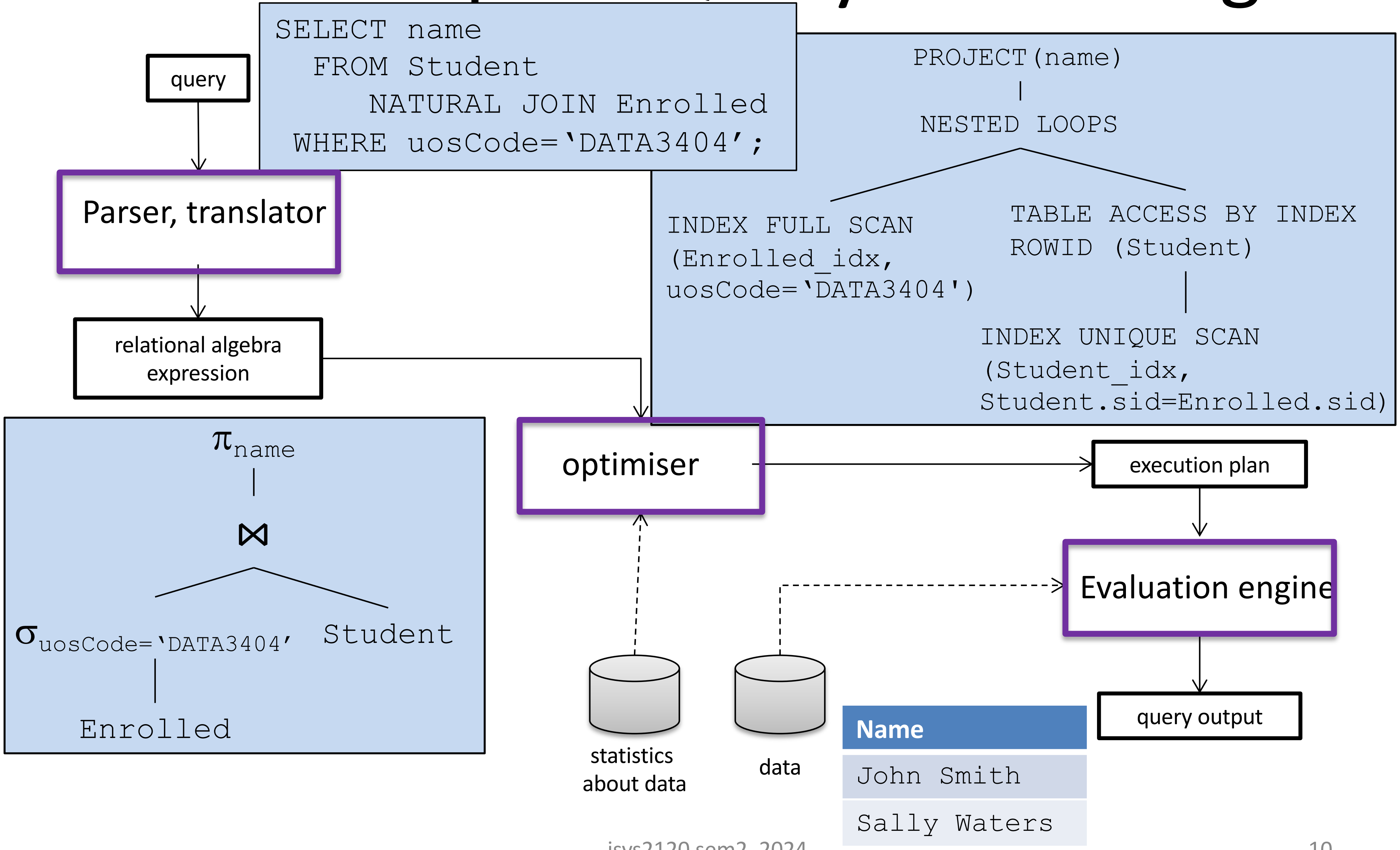
More detailed view

- Parse the SQL
- Produce *a number of potential "plans",* each with a sequence of operations that will calculate the result
- *Choose one* of these plans
  - Hopefully, one which will run quickly
- Execute the chosen plan, by accessing various tables in specific ways
- Return the result

Compiling a query

- SQL is declarative
  - It indicates what information is to be returned
- The DBMS must execute a concrete, imperative program, that describes step-by-step what to do
- The DBMS must therefore begin by finding an imperative program which will calculate the result described in the declarative query
  - This is somewhat similar to the way a programming language can be compiled into machine instructions
- The process is done in stages
  - Convert the text of SQL into a structured logical form (described using Relational Algebra)
  - Convert the Relational Algebra logical plan into a plan with physical operations (each of which has a program already available)
  - Actually, there are many possible physical plans for a single SQL query
    - The way the DBMS chooses a good physical plan is called “query optimisation”

Basic Steps in Query Processing



```
SELECT name
  FROM Student
     NATURAL JOIN Enrolled
 WHERE uosCode='DATA3404';
```

$$\pi_{name}(\sigma_{uosCode='DATA3404'}(Enrolled) \bowtie Student)$$

```
                 PROJECT(name)
                      |
                 NESTED LOOPS

INDEX FULL SCAN               TABLE ACCESS BY INDEX
(Enrolled_idx,                ROWID (Student)
uosCode='DATA3404')                 |
                              INDEX UNIQUE SCAN
                              (Student_idx,
                              Student.sid=Enrolled.sid)
```

Name
John Smith
Sally Waters

Access Paths and Selections

An **access path** is a method of retrieving tuples.
Examples:

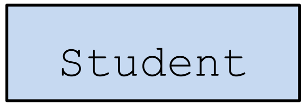
Query

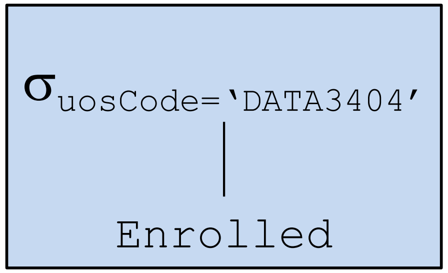
```
SELECT *
FROM Student;
```

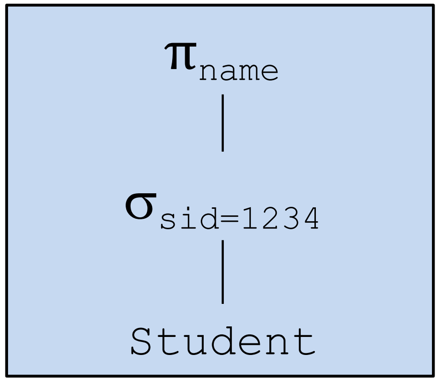
```
SELECT *
FROM Enrolled
WHERE uosCode='DATA3404';
```

```
SELECT name
FROM Student
WHERE sid=1234;
```

RA







Access Paths

- **Table Scan:** Retrieve all pages of relation
- Table Scan + Filter
  (table scans typically try to include filtering)
- **Index Scan:** Use index with <u>matching</u> search key to find matching records (if available)
  - tree index?
  - hash Index?
- **Index-only scan**: Use a covering index without accessing records (if it exists)

DBMS computation

- Code that performs DBMS computations runs on the CPU
- Code runs in response to requests which usually arrive over the network, or as procedure call from application-processing code
- Code will need to operate on the data which physically represented the database contents

Where is the data?

- 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

HDD characteristics

- HDD is persistent; it is (relative to DRAM) slower, cheaper (and hence usually larger), and accessed in larger chunks.
  - *DRAM is volatile, but HDD is persistent*
    - We insist that data to be saved between runs, and despite crashes. (Obviously!)
  - *HDD is accessed much slower than main memory*
    - Order of 2ms latency to fetch data, vs 0.1μs [that is, 20,000 times slower!]
  - *HDD is much cheaper than (DRAM) (per Mb) than HDDs, so a system can afford much more capacity in HDD*
    - A very large dataset may not fit into the available DRAM
  - *HDD is accessed in blocks (sometimes called pages) usually 4KB in one operation, while DRAM is accessed in words (4-8 Bytes at once)*

Impact of HDD characteristics

- Because HDD is persistent; it is (relative to DRAM) slower, cheaper (and hence usually larger), and accessed in larger chunks, this has major implications for DBMS design!
- Authoritative copy of 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!
  - Try to ensure that related data that is needed together, is in the same block, so one move brings it all to main memory for computation
- Indeed, **overall performance is determined largely by the number of disk I/Os done**

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
- Traditional 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).

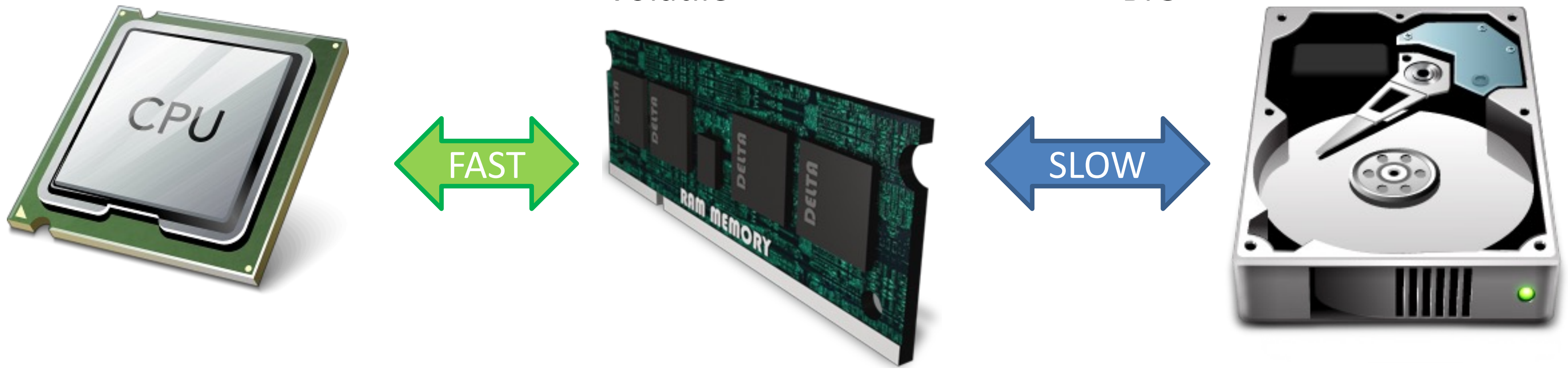
Where is Data Stored?

Main Memory (DRAM):

- Expensive
- Fast
- Volatile

Secondary Storage (HDD):

- Cheap
- Stable
- BIG



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 most effectively

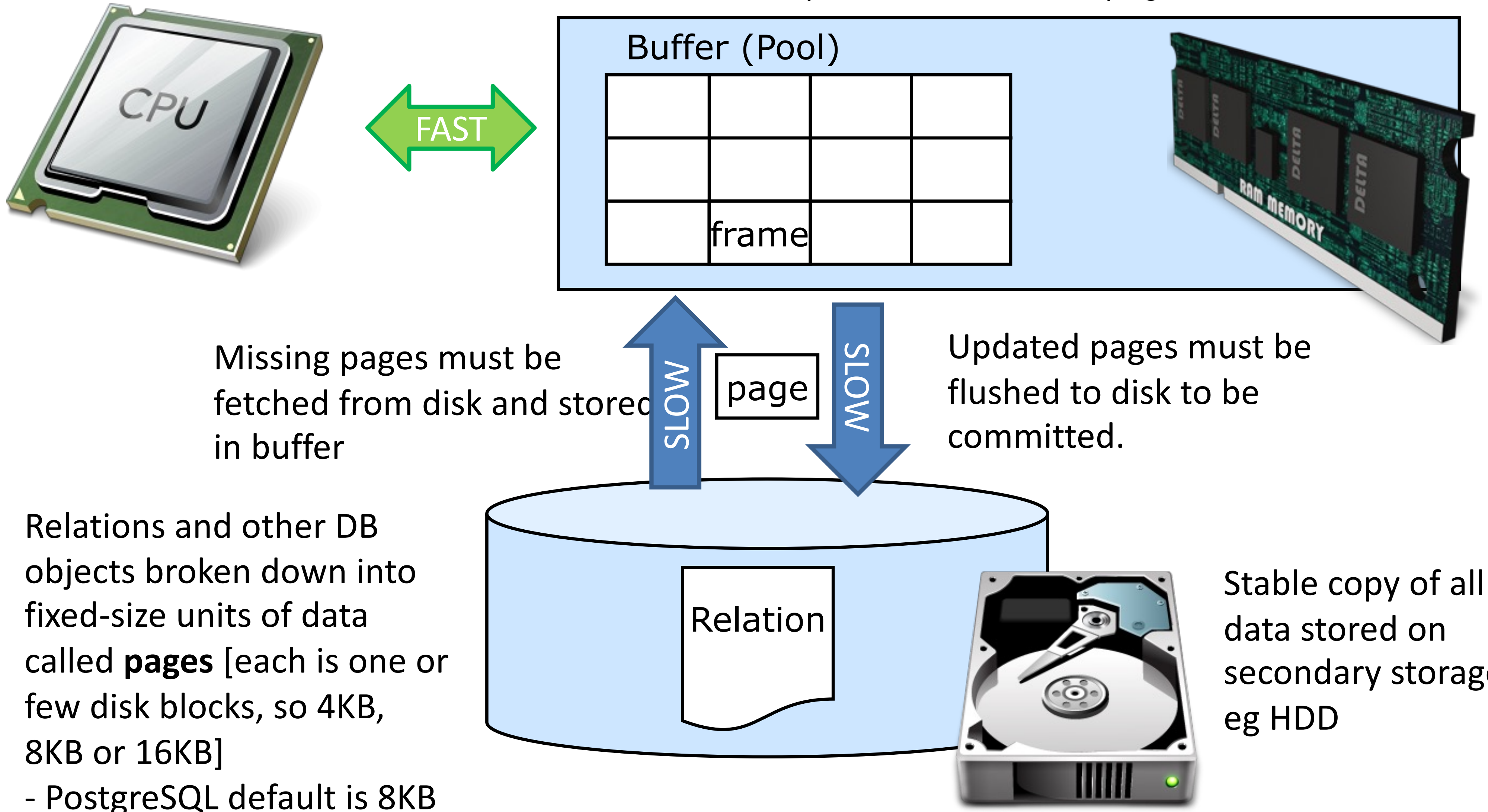
Accessing data from secondary storage

- **Because time to bring a page from storage is much greater than time for even many memory accesses, the way to lower time spent accessing data, is to reduce the number of times that pages that are accessed from secondary storage**
  – Aim where possible to reduce the number of pages that contain data items one must examine
  – Aim where possible that data one needs to access is on a page that was already brought into memory

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 units of data called **pages** [each is one or few disk blocks, so 4KB, 8KB or 16KB]
- PostgreSQL default is 8KB

Stable copy of all data stored on secondary storage eg 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)
  - Only way to access is a *file scan* checking all records
  - Simple, good for learning, not usually used in dbms
- **Sorted Files** – store records in sequential order, based on the value of the search key of each record
  - May be useful if records must be retrieved in some order, or only a `*range*' of records is needed.
  - Simple, good for learning, not usually used in dbms
- **Indexes** – complex data structures to organize records via trees or hashing
  - Several variants, used in real dbms
  - Choice is controlled by SQL commands eg CREATE INDEX

Warning: there is a different complex data structure, also called “heap” which is taught in some other classes

(Unordered) Heap Files

- Simplest file structure contains records in no particular order.
  - No way to find a record of interest, except to look through all the records one after another, until you find the one you want
- Rows appended to end of file as they are inserted
  - Hence the file stays unordered
  - Insertion is quick
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

Example: Transcript Stored as Heap File

Note: this example is for illustration; actually a page is likely to contain many more records than just 2-4!

page 0

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

page 1

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

page 2

234567	CS305	S1999	4.0
878787	MGT123	S1996	3.0

Access to Heap Files

- How to find records in a heap file, to answer a query?
- Eg SELECT <some columns> FROM table WHERE <condition>
- Access method is a ***linear scan with filter*** (also called: table scan)
  - Examine each block in turn (from buffer, or fill buffer from disk if necessary)
  - Within the block, look at each record in turn, and check whether <condition> is true
    - If so, output the contents of the appropriate columns
- This is expensive!
  - Usually, the whole file must be processed
  - If you can stop once you have found what you are looking for (eg you know there is only one matching record), then on average half of the pages in a file must be read,
  - This is as efficient as possible if all rows are returned (SELECT * FROM *table*)
  - Very inefficient (relative to what is really needed) if a *few* rows are wanted

Sorted File

- Rows are sorted based on some attribute
  - Successive rows are stored in same (or successive) pages
  - This makes it fairly fast to get to records with a particular value for the sorting attribute
    - Or for records where sorting attribute lies in some range
  - The values of the other attributes are not arranged in any particular way

- One problem: Maintaining sorted order
  - After the correct position for an insert has been determined, shifting of subsequent tuples necessary to make space (very expensive)

Example: Transcript as Sorted File

Here, we suppose the records are sorted based on StudentId
Notice: therefore, they are not sorted based on other attributes!

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

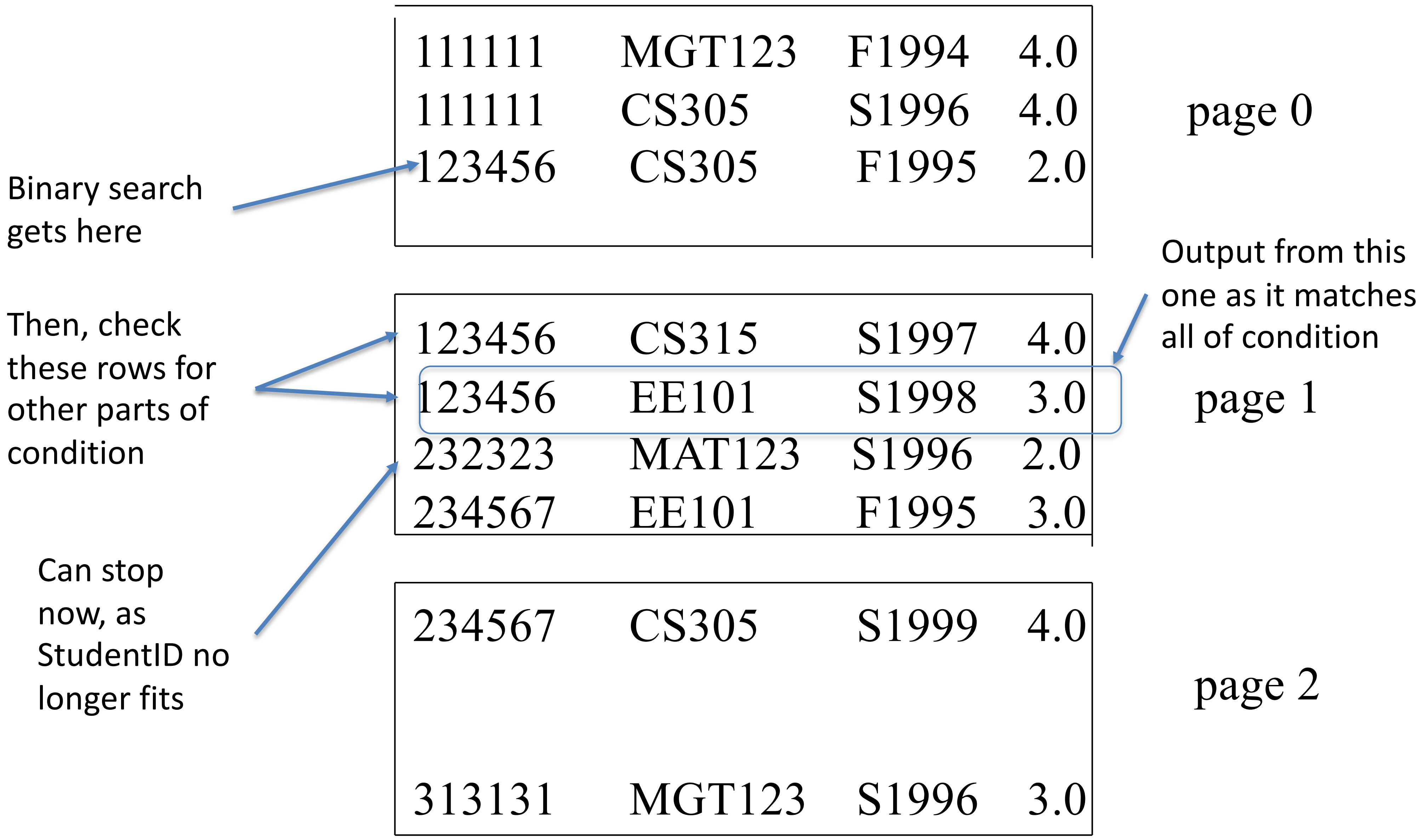
page 2

Access to Sorted File

- How to find records in a heap file, to answer a query?
- Eg SELECT <some columns> FROM table WHERE <condition>
- First, suppose that <condition> determines the sorting attribute (either exactly, or within a range)
  - Eg suppose sorting attribute is StudentID, condition is WHERE semester = ‘S1998’ AND StudentID = 123456
- Access method could be a ***binary search*** **(details in comp2123)**
  - In logarithmic time (much faster than linear), get to first row that has appropriate value for the sorting attribute
  - Then check each successive record until sorting attribute is no longer suitable, to see if the rest of <condition> also is true
    - If so, output the contents of the appropriate columns

Access in Sorted Transcript File

WHERE semester = ‘S1998’ AND StudentID = 123456



Binary search gets here

Then, check these rows for other parts of condition

Can stop now, as StudentID no longer fits

Output from this one as it matches all of condition

page 0

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

page 1

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

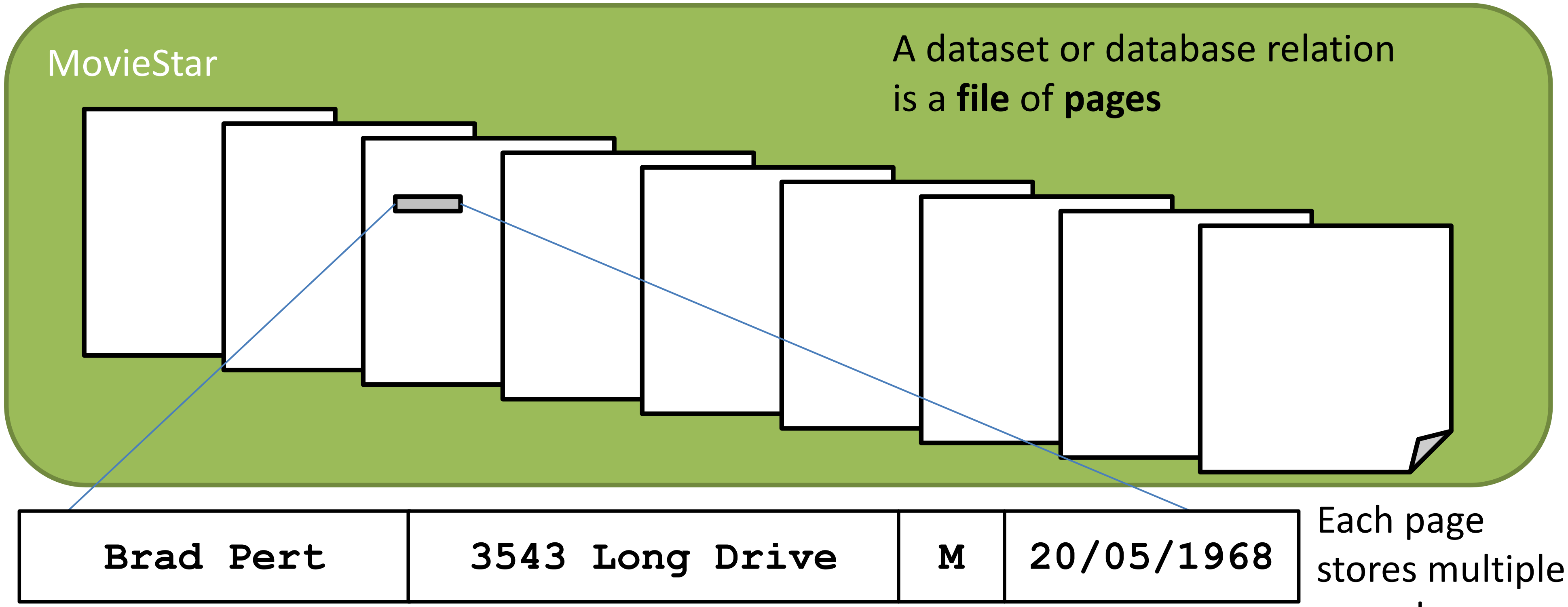
page 2

234567	CS305	S1999	4.0
313131	MGT123	S1996	3.0

Access to Sorted File

- How to find records in a sorted file, to answer a query?
- Eg SELECT <some columns> FROM table WHERE <condition>
- Now, consider case when <condition> does NOT determine the sorting attribute (either exactly, or within a range)
  - Eg suppose sorting attribute is StudentID, condition is WHERE semester = ‘S1998’ AND gpa >= 3.0
- Access method needs to be linear scan-and-filter
  - Look at all the rows, one after another
  - (just as if data is stored as Heap, as far as condition is concerned!)
  - Very inefficient (relative to what is really needed) if a *few* rows are wanted

Heap Files vs Sorted Files



A dataset or database relation is a **file** of **pages**

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 sorting key); Linear scan (if sorting key is not determined by condition)

Agenda

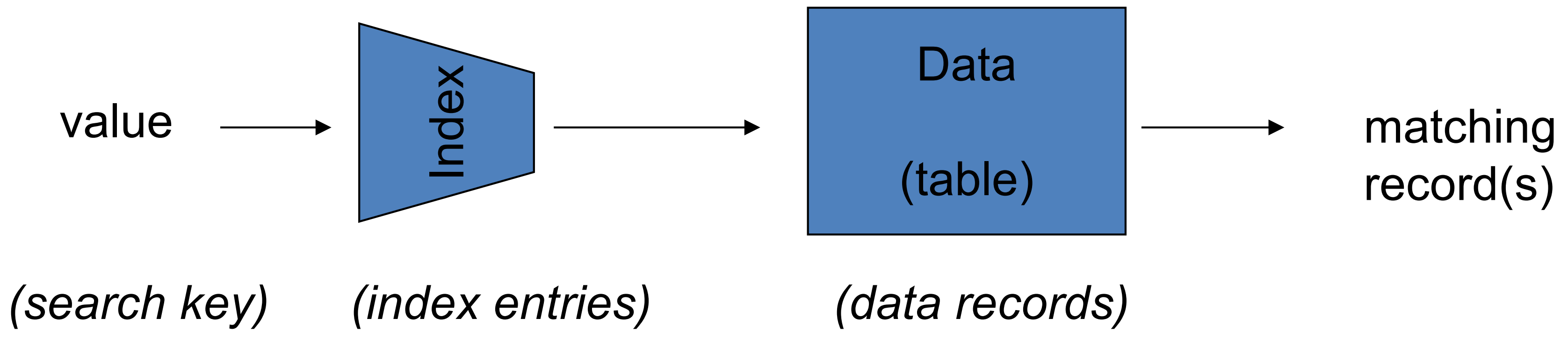
- **Motivation**
- **Database Internals: Overview**
  - Software
  - Hardware
  - Storage Layer: Physical Data Organisation
- **Indexing of Databases**
  - Efficient data access based on search keys
  - Several design decisions…
- **Database Tuning**
  - How to suggest appropriate indexes for a given SQL workload
  - Awareness of the trade-off between query performance and indexing costs (updates)

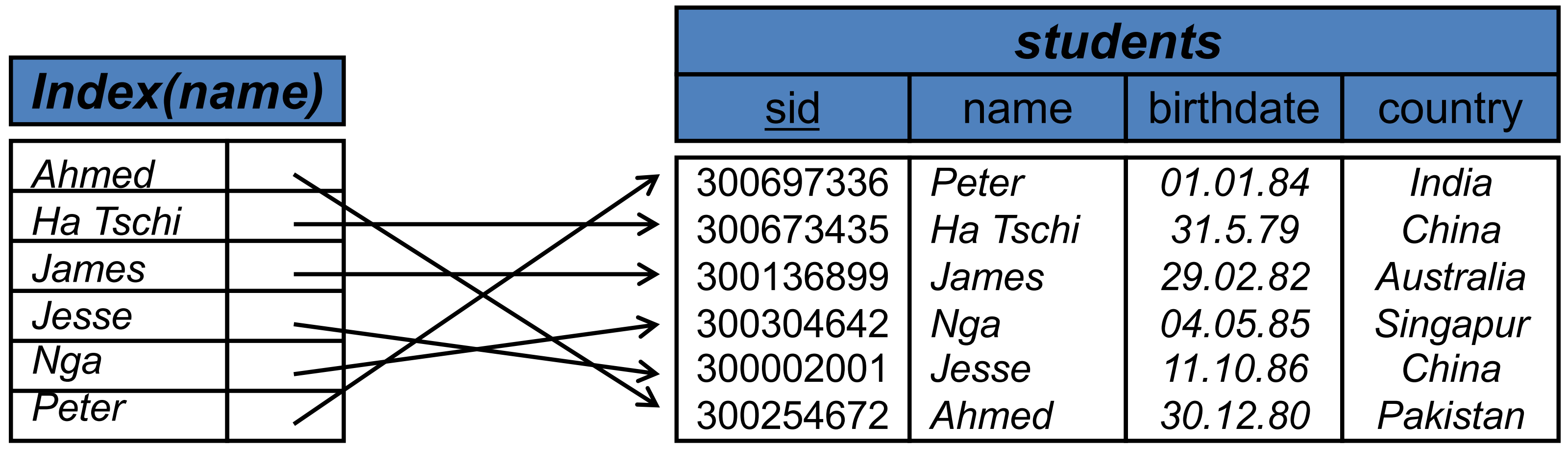
Indices

- Can we come up with a file organisation that is
  - as efficient for searches (especially on ranges) as an ordered file?
  - Able to speed access for a variety of different conditions?
  - 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 find the contents page which contains the value
  - Index typically much smaller than the actual data
  - Instead of scanning through whole book each time,
    using the index is much faster to navigate (less material to search, and arranged in order for fast search)

Index Example





students

sid	name	birthdate	country
300697336	*Peter*	*01.01.84*	*India*
300673435	*Ha Tschi*	*31.5.79*	*China*
300136899	*James*	*29.02.82*	*Australia*
300304642	*Nga*	*04.05.85*	*Singapur*
300002001	*Jesse*	*11.10.86*	*China*
300254672	*Ahmed*	*30.12.80*	*Pakistan*

Index Definition in SQL

- Create an index

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

  – Example:

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

- Index on primary key is generally created automatically by CREATE TABLE command (see later)
- To drop an index

  DROP INDEX *index-name*

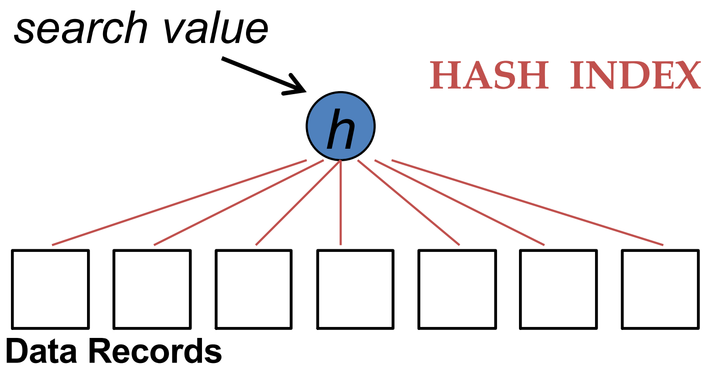
- Sidenote: SQL-92 does actually not officially define commands for creation or deletion of indices.
  – vendors kind-of ‘agreed’ to use this syntax consistently

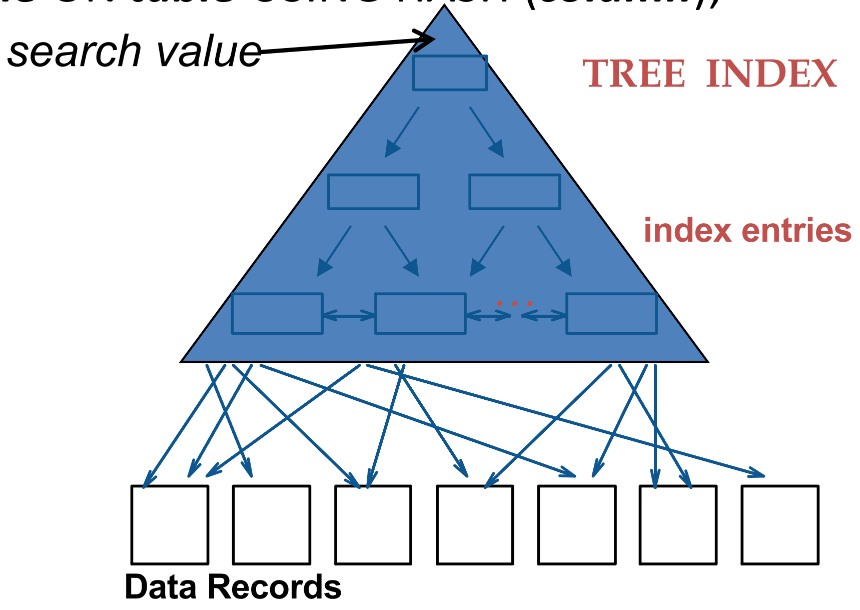
Indices - The Downside

- 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)

Which Types of Indexes are available?

- Tree-based Indexes eg B+-Tree
  - *Very flexible, supports point queries (equality to a specific value), range queries and prefix searches*
  - *Index entries are stored in sorted order by search key (with a complex arrangement for access that looks at very few blocks)*
  - *Found in every DBMS platform*
  - This is default index in PostgreSQL (what is done by plain CREATE INDEX statement)
- Hash-based Indexes
  - *Fast for point (equality) searches – but not fast for other calculations*
  - PostgreSQL syntax: CREATE INDEX ***indexname*** ON ***table*** USING HASH (***column***);



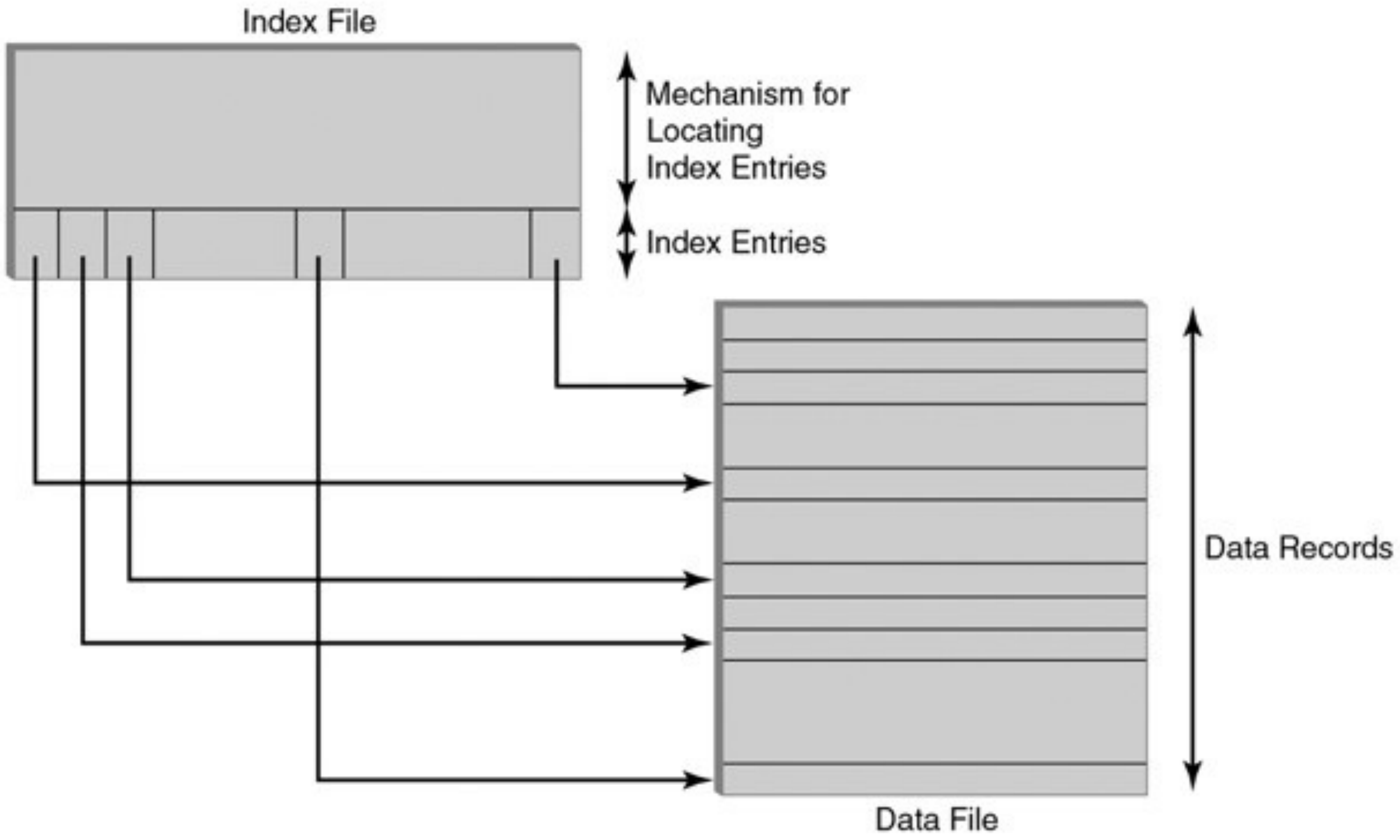


Some platforms also have other index types
eg bitmap for OLAP, R-tree for spatial data

Unclustered Index

- We say index is “unclustered” when index entries and data rows are not ordered in the same way
  - If multiple data records are found from index, they are likely to be on different data blocks (and so fetching them needs perhaps as many block-reads as there are matching records)
- There can be many unclustered indices on a table, each arranged for access on a different column (or even a sequence of columns, see later)
- Index created by CREATE INDEX is generally an unclustered index (also called: secondary index)

Unclustered Index



Clustering Index

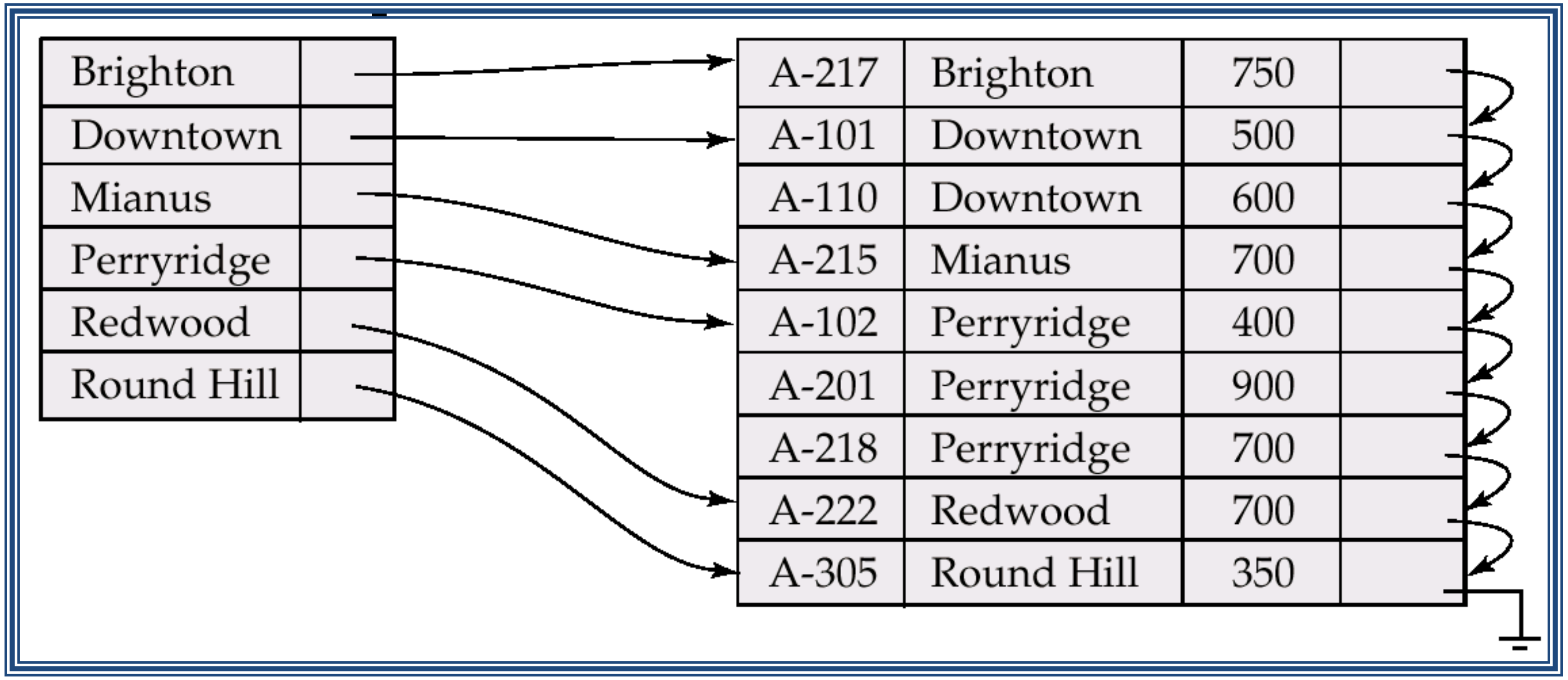
- Also called: clustered index
- We say index is “clustering” when index entries and data rows are ordered in the same way
  - If multiple records are found from index, they are likely to be on only a few data blocks (and so fetching them needs only a few block-reads, often less than number of matching records)
- The particular index structure (eg, hash, tree) dictates how the data rows are organized in the storage structure

Clustering Index

- 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 a clustering index on primary key
- Instead, one can change the order of data records, to match a (previously created) index
  - Note that the table will no longer be clustered on primary key; primary key access will likely become significantly slower!
  - PostgreSQL syntax: CLUSTER *tablename* USING *indexname*;

Example: Clustering Index

- Clustering Index on **branch-name** field of



Source: Silberschatz/Korth/Sudarshan textbook

Comparison

- ***Clustering index***: index entries and rows are ordered in the same way
- There can be at most one clustering index on a table
  - CREATE TABLE generally creates an integrated, clustering (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 buffer hits

- ***Unclustered (secondary) index***: index entries and data rows are not ordered in the same way
- There can be many unclustered indices on a table
  - As well as perhaps one clustering index
  - Index created by CREATE INDEX is generally an unclustered 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)
- Note that index entries with given Att1 are all together, and *within that collection*, the index entries are arranged based on Att2

Access with Multicolumn Index

- CREATE INDEX Inx ON Tbl (A*tt1*, A*tt2*)
- This supports efficient finer granularity equality search:
  - "Find row with value (A1, A2)"
- Also, (for tree index) it supports efficient range search on A1
  - "Find rows with Att1 between A1 and B1"
- (for tree index) it supports some compound searches where Att1 has equality and Att2 is a range
  - "Find rows with Att1 = A1 and Att2 between A2 and B2"
- Especially useful when it <u>covers</u> a whole query (see later)

Index Classifications

- 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
- Clustering 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)**

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*

Agenda

- **Motivation**
- **Database Internals: Overview**
  - Software
  - Hardware
  - Storage Layer: Physical Data Organisation
- **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)

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?
  - Clustering? Hash or Tree?

- Not considered in isys2120, but important for the DBA: Where 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 isys2120, we discuss only 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 rows with that given value.
- 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 clustering 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 clustering index on that attribute; however that index is of no use for this query.
  - Choose to create an extra (unclustered) 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 clustering index on these PK attributes
  - but it is of no use for this query…
- Hence: Choose to create an extra unclustered index with search key *Grade*
  - Could be either tree or hash index (as condition is equality)
  - Maybe consider: a *covering* index with composite search key *(grade, studId) which* would allow to answer complete query from 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 clustering index on these sequence of attributes
  - But the search condition is on *year* and *semester* => must be prefix of index structure, or index is not useful!
  - Hence PK index not of use
  - Create a Covering INDEX: either (*year, semester, uosCode)* or *(semester, year, uosCode)*

Choosing an Index (cont’d)

- Example 4:

```
SELECT T.uosCode
  FROM Transcript T
 WHERE T.year > 2009 AND T.year < 215 AND
         T.semester = ‘Sem1’
```

- This query has an equality search on *semester*, and range search on *year*.
- If the primary key is *(studId, year, semester, uosCode)*, it is likely that there is a clustering index on these sequence of attributes; no use for this query
- Unclustered tree composite index on *(semester, year)* in that order, will put index entries for matching rows together in the index
- Or consider a covering index on *(semester, year, uosCode)*

References

- Kifer/Bernstein/Lewis(complete version, 2ed)
  - Chapter 9.1 -9.4, 9.8, 12 (further reading in rest of Chapters 9-12)

Also

- Silberschatz/Korth/Sudarshan(7ed)
  - Chapters 12.1, 14.1, 14.2, 14.6, 14.8 (further reading in Ch 12-16)
- Ramakrishnam/Gehrke(3ed)
  - Chapter 8 (further reading in Chapters 9-15)
- Garcia-Molina/Ullman/Widom(complete book, 2ed)
  - Chapter 8.3-8.4, 13.1-13.3.1, 14.1 (further reading in Ch 13-16)
- *All these books have technical details on this topic, providing a lot of insight into how disk-based indexes are implemented, and how the query optimiser makes its decisions. We only need an overview here; more technical details are covered in data3404.*

Summary

- Key concepts
  - overall structure of dbms internal processing of a query (hardware and software)
  - Index structures
    - Clustering vs unclustered
    - Tree-based vs hash
- Key skills
  - understand impact of table structure on processing efficiency of a simple query (select-project on single table)
  - Understand tradeoffs from creating an extra index
  - Choose indices for a table, to speed up some simple queries