Isys2120 lab10
tutor material, sample answers
Index structures

Sem2 2024

A. Review of relevant “lecture” content

Agenda

- **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)

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

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).

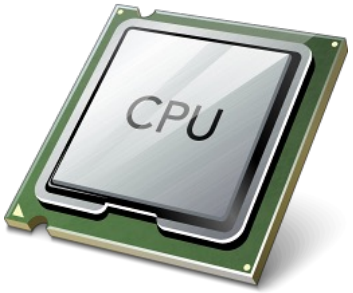
Where is Data Stored?

Main Memory (DRAM):

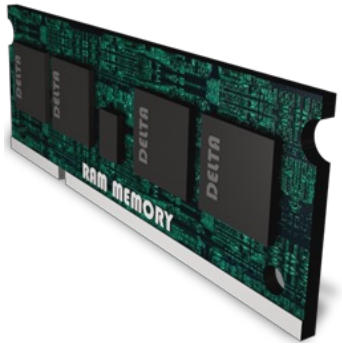
- Expensive
- Fast
- Volatile

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

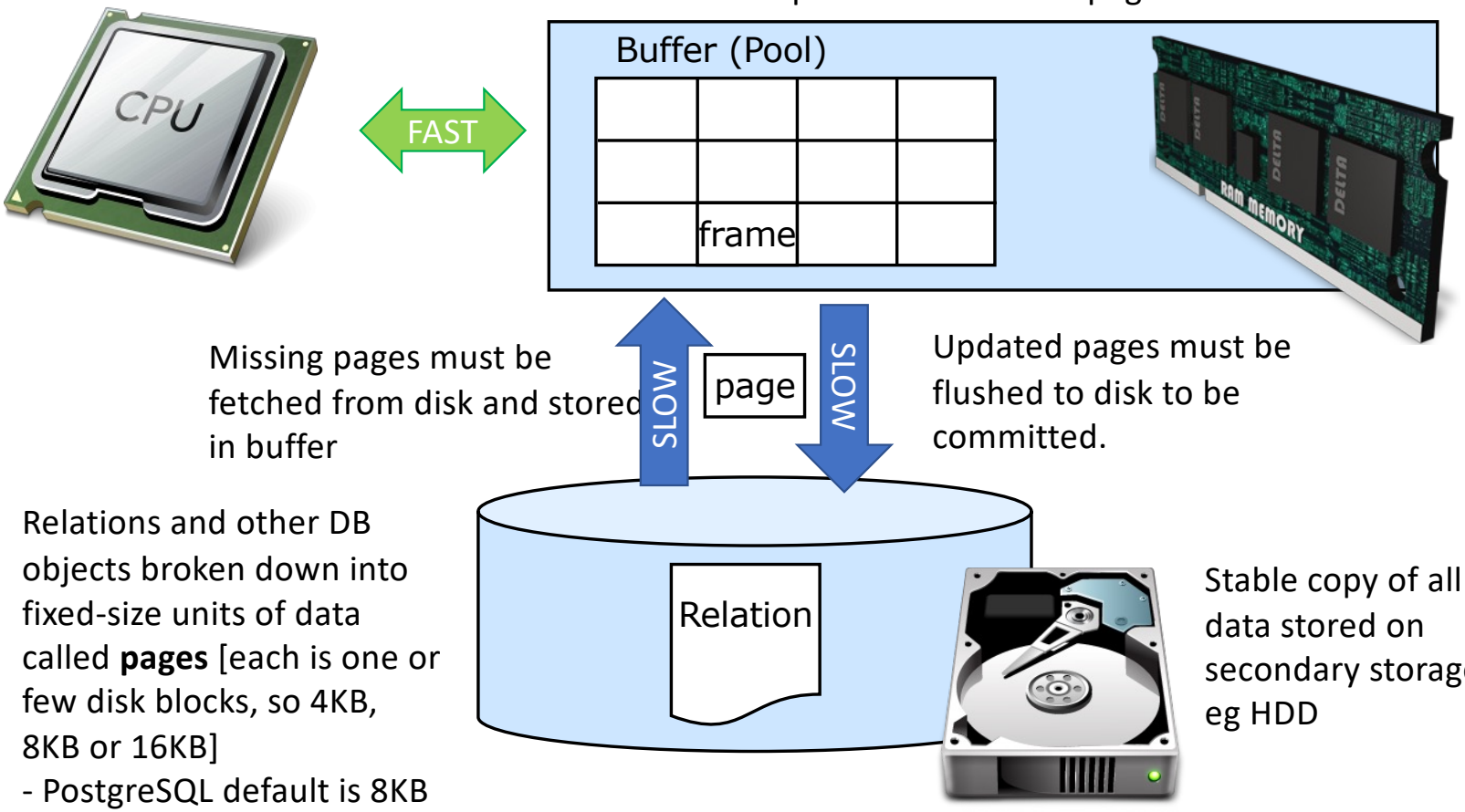
https://www.iconfinder.com/icons/104158/ram_icon

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

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

Using a Buffer to Hide Access Gap

(as much as possible)

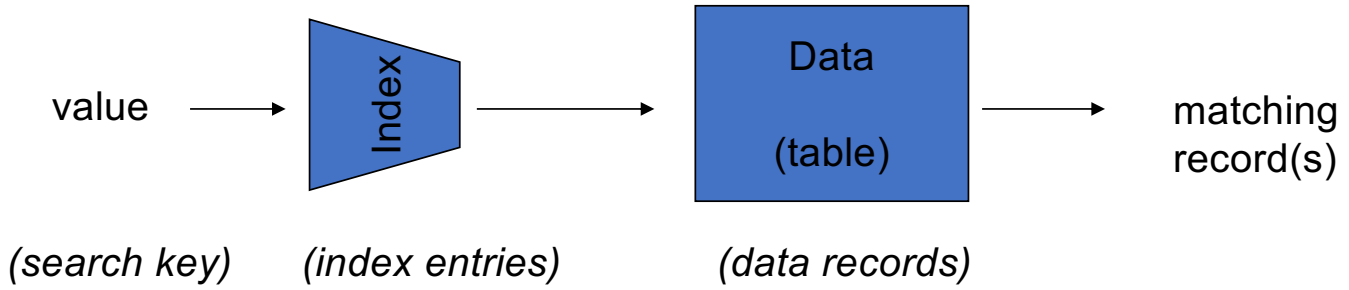


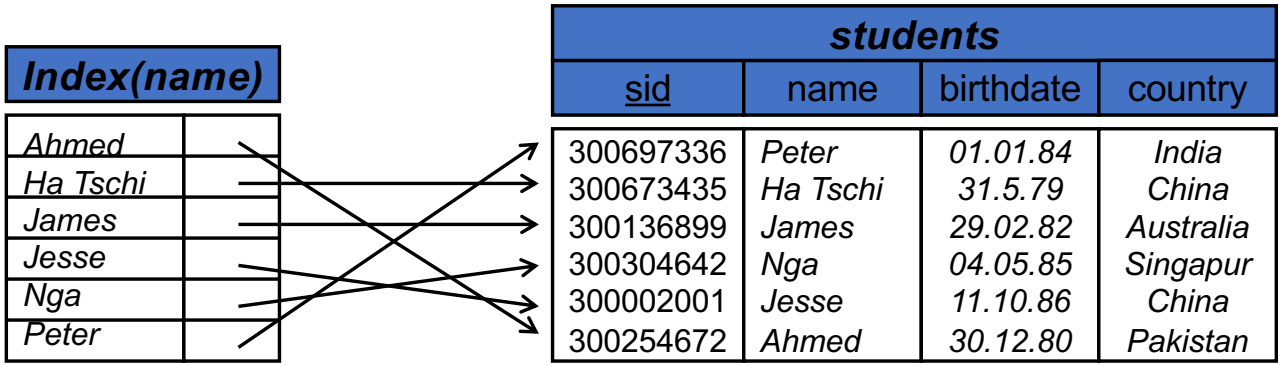
Indices

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





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*

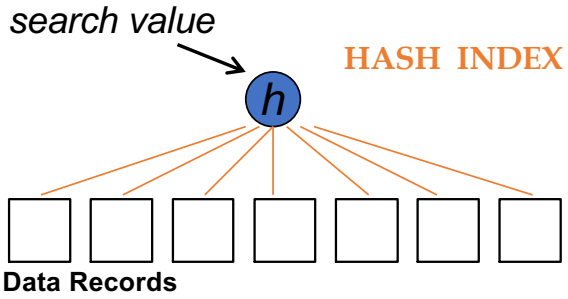
- Sidenote: SQL-92 does actually not officially define commands for creation or deletion of indices.
  - vendors kind-of 'agreed' to use this syntax consistently

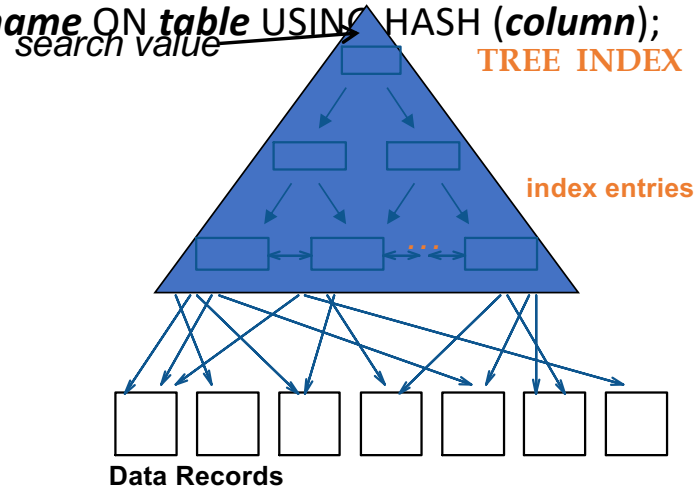
Indices - The Downside

- 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***);



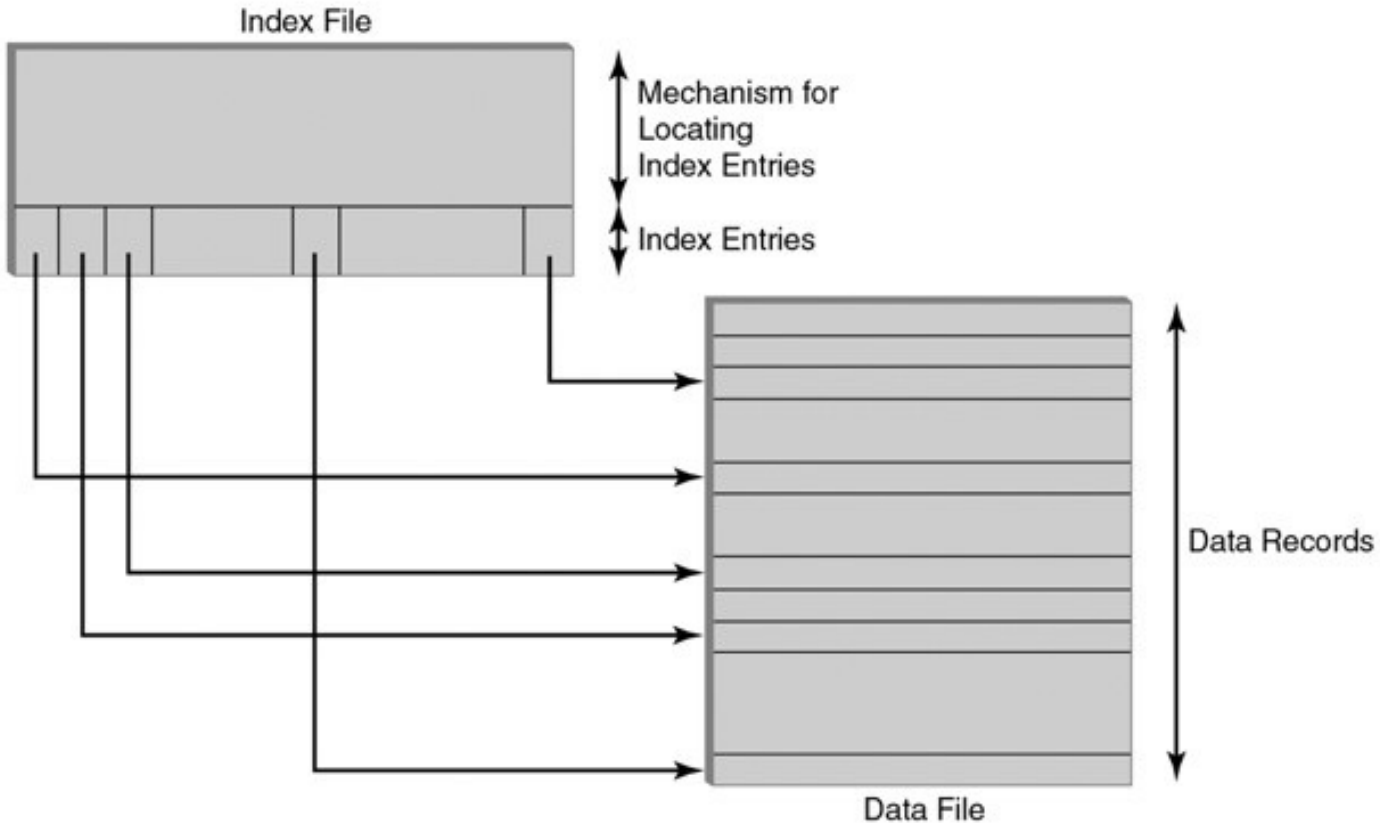


Some platforms also have other index types
eg bitmap for OLAP, R-tree for spatial data

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

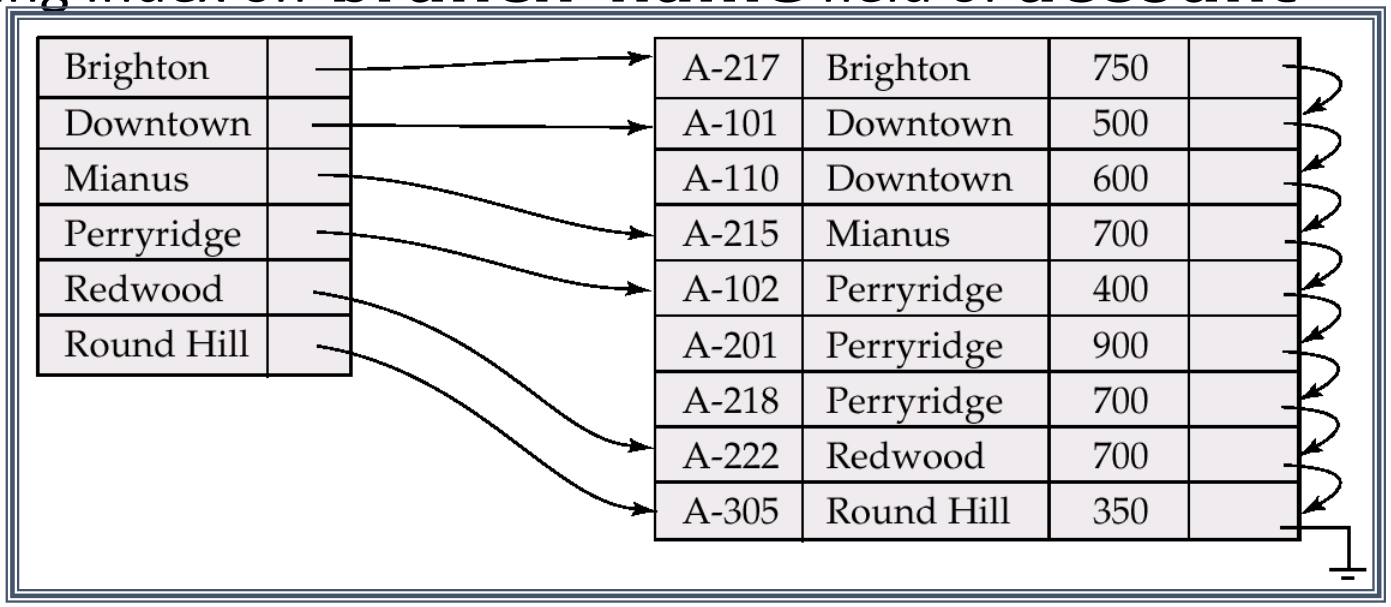
- Also called: clustered index
- We say index is “clustering” when index entries and data rows are ordered in the same way
  - If multiple records are found from index, they are likely to be on only a few data blocks (and so fetching them needs only a few block-reads, often less than number of matching 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 clustered 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 **account**



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 covers a whole query (see later)

Index Classifications

- Tree-based vs Hash
- 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**.

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*

Choices of Indexes

- 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 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)*

Activity B

```
CREATE TABLE UnitOfStudy (
  uoSCode       CHAR(8),
  deptId        CHAR(3)  NOT NULL,
  uoSName       VARCHAR(40) NOT NULL,
  credits       INTEGER NOT NULL,
  fee           INTEGER NOT NULL,
  PRIMARY KEY (uoSCode),
  UNIQUE (deptId, uoSName)
);
```

Clustering tree-based index on uoSCode

```
CREATE INDEX UoS_Name_IX ON UnitOfStudy (uoSName);
```

Unclustered tree-based index on uoSName

B(i)

```
SELECT uoSName
FROM UnitOfStudy
WHERE uoSCode = 'ISYS2120';
```

- Query is for a known value of uoSCode. Therefore the query can use the clustering index on uoSCode
  - This will very quickly find the (one) relevant tuple, reading from disk only one block of data [if that block isn't already in the buffer]
    - Plus perhaps some read of index, but that is likely to be in buffer already

B(ii)

- `SELECT uoSCode`
- `FROM UnitOfStudy`
- `WHERE uoSName = 'Data and Information Management' AND deptId = 'SCS';`

- Query is for a known value of uoSName. Therefore the query can use the unclustered index on uoSName
  - This will quickly find the pointers to all of the tuples with that uoSName, and then the calculation can get each of those (likely from different blocks of the data, since this index is unclustered), and filter each in turn to see whether the deptId is SCS.
  - Overall speed is fairly fast
    - Number of blocks transferred = (approx.) number of records with given uoSName

B(iii)

- Would it be improvement to use hash index (if that were created instead of tree-based index, on same column)?
- Answer: We could use hash index if that existed instead, because each query looks for a single value of the indexed search key [recall: hash index can find single value, but not effectively find values in a range]
- Is this any better? In theory, hash index can be even faster than tree-index (less searching in the index itself), but in practice, index is likely to be in buffer already, so unlikely to reduce the number of blocks transferred from disk to memory

B(iv)

```
SELECT uoSCode
FROM UnitOfStudy
WHERE (fee BETWEEN 5000 and 10000) AND deptId = 'SCS';
```

- Neither index is useful here, as query does not specify the value of uoSCode nor of uoSName
- So query needs to do linear-scan-and-filter
  - It reads the whole data file from disk into memory, block by block
  - Number of blocks transferred = number of blocks in the data file (unless some are already in buffer)

B(v)

```
SELECT uoSCode
FROM UnitOfStudy
WHERE (fee BETWEEN 5000 and 10000) AND deptId = 'SCS';
```

- Possible indices to consider
  - Unclustered index on fee [must be tree-based, to be useful for this query]
    - ok but still needs to do lots of filtering, as it needs to check all data records with appropriate fee
    - Usefulness will depend on how common this dept is, among all rows for fees in this range
  - Unclustered index on deptId [could be either tree-based or hash index]
    - ok but still needs to do lots of filtering, as it needs to check all data records with appropriate deptId
    - Usefulness will depend on how common this range of fees is, among all rows for this dept
  - Composite unclustered index on (deptId, fee)
    - Good: only needs to transfer a block for each data record that matches the whole condition
  - Composite unclustered index on (deptId, fee, uoSCode)
    - This covers the query, so won't need to access data records at all
    - Just a few parts of index to transfer [If index is already in buffer (as may be likely) no data transfers at all!]
  - Note: not particularly useful would be an index on (fee, deptId) or (fee, deptId, uoSCode) because matching index entries will be scattered within the index, due to lexicographic ordering

B(vi)

```
SELECT uoSName
FROM UnitOfStudy
WHERE (credits BETWEEN 6 AND 12);
```

- Consider that most rows have credits = 6; many of the rest have credits = 12
- Possible indices
  - Unclustered index on credits [must be tree-based, to be used at all for this query]
    - Can be used, will transfer most of the data blocks anyway
    - Likely better than this is just to do table scan and filter!
  - Unclustered composite index on (credits, uoSName)
    - This covers the given query, so only need to access index, not data records
    - Will access most of the index, but that will likely be smaller than the data file (fewer columns in each index entry, compared to each data record)