Data storage in DBMS

- **page**: unit of information read from/written to disk
  - size: DBMS parameter, e.g. 4-8KB
- cost of page IO dominates cost of typical DB operations, so DB systems need to minimise this cost
- cost of reading several pages in order physically stored « cost of reading pages in random order
- **buffer manager**: handles page fetching
- **record ID/rid**: unique ID for each record/tuple
  - rid value identifies disk address of page containing the record
- **file of records**: can be created, destroyed, have records inserted/deleted, scan sequentially
  - relation will be represented by a file of records

File Organisations

- **heap file**: records stored unordered
  - supports retrieval of all records, or particular page specified by `rid`
- **sorted file**: pages, and records within them, are stored in order
  - fast for range queries, expensive to maintain
- **index file**: store records directly in index
  - fastest retrieval for a given order

Index

- **index**: data structure organising data records on disk to optimise certain retrieval operations
  - allows efficient retrieval of records satisfying search conditions on **search key** fields of the index
  - **data entry**: record stored in an index file; in this course typically leaf is a $\langle , k, rid \rangle$ pair, however could also store the record itself
- **clustered index**: ordering of data records same as ordering of data entries in index
  - only clustered if data records are sorted on search key field

- **unclustered index**: ordering of data records very different to data entries in index
  - order of data records is defined by physical order
- files are rarely kept sorted because it is too expensive to maintain, so most indexes will be unclustered (unless they are storing records as data entries directly)
- clustered indices are very efficient for range search
- cost of using an index for a range search query varies significantly depending on whether or not the index is clustered:
  - for a clustered index, `rid`s in qualifying data entry points to a contiguous collection of records, so only a few page IOs are needed
  - for an unclustered index, `rid`s in data entries point to records in many different pages, so a large number of page IOs may be needed
- **primary index**: index on a set of fields that includes the *primary key*
- **secondary index**: any other index
- **duplicate** data entry: if they have the same value for search key field associated with the index
  - primary index guaranteed not to have duplicates
  - secondary index may contain duplicates
- **unique index**: if no duplicates exist, the search key contains a candidate key

Hash-based Index

- index is a collection of **buckets**
  - bucket is a primary page + overflow pages as required
- hashing allows you to quickly find records based on given search key value
- **hash function** maps search key to bucket
  - `h(r.search_key)`: bucket in which record `r` should be placed
- good for equality selections
- no good for range selections, as buckets aren't sorted

Tree-based index

- organise records in a B+ tree: nodes contain pointers to lower levels
  - left subtree: lower values

  - right subtree: higher values
  - leaves contain data entries sorted by search key values
  - structure ensures all paths from root to leaf are the same length, i.e. balanced
  - finding correct leaf page is faster than binary search of pages in sorted file, as each non-leaf node can accommodate many pointers
  - nodes typically a physical page, so retrieving a node involves 1 IO
  - in practice height typically ~3-4, so ~ 3-4 IOs to retrieve desired leaf page

- **fanout**: average number of children for a non-leaf node
- if every non-leaf node has n children, tree of height h has n^h leaf pages
- typically each non has on average 100 children, so with height 4: 100 million leaf pages
- search a file with 100 million leaf pages and retrieve page you want in 4 IOs, vs. binary search taking $\log_2 10^8 \approx 25$ IOs
- good for range selections

Query evaluation

- SQL queries are translated into extended form of relational algebra
- Query evaluation plans are represented as trees of relational operators
- relational operators are building blocks for query evaluation
- implementation of relational operator is optimised for performance

Operator evaluation techniques

- there are several algorithms for implementing each relational operator
- choice will depend on e.g. size of tables, existing indexes, sort order
- common techniques:
  - **indexing**: selection/join, use index to examine only the tuples satisfying the condition
  - **iteration**: iteratively examine all tuples in an input table
    * if only a few fields required, and a corresponding index with these fields exists, you can scan all index data entries instead
  - **partitioning**: e.g. sorting, hasing. Partition tuples on a sort key to decompose operation to a less expensive collection of operations on partitions

Access paths

- **access path**: way of retrieving tuples from a table. Consists of either:

  - file scan
  - index + matching selection condition

- every relational operator acceps 1+ tables as input, and access methods to retrieve tuples contribute significantly to cost of operator
- consider selection of form `attr op value`. An index **matches** the selection condition if the index can be used to retrieve only the tuples that satisfy the condition
- **hash index** matches selection if there is a term of the form `attribute=value` in the selection for each attribute in the index search key
- **tree ndex** matches selection if there is a term of form `attr op value` for each attribute in a **prefix** of the index's search key
  - e.g. $\langle a \rangle, \langle a, b \rangle$ are prefixes of $\langle a, b, c \rangle$; $\langle a, c \rangle, \langle b, c \rangle$ are not
- an index can match a subset of conditions in a selection condition, even if it does not match the entire condition
- **primary conjunct**: conditions an index matches
- **selectivity**: number of pages retrieved (index pages + data pages) if we use this access path to retrieve all desired tuples
  - table containing an index matching a given selection: at least two access paths, namely the index and a scan of the data file
  - sometimes you can scan the index itself
- **most selective access path**: retrieves fewest pages, minimising cost of data retrieval
- **reduction factor**: fraction of tuples in a table satisfying a given condition
  - with several primary conjuncts, the fraction of tuples satisfying all of them is approximately the product of their reduction factors

External Sorting

- sorting records on a search key (1+ attributes) is very useful, e.g.:
  - user wants answer in some order
  - useful for eliminating duplicates (e.g. projection)
  - joins require sorting
- **external sort**: used when data to be sorted won't fit in main memory
  - aims to minimise cost of disk accesses
  - typically generates runs, and then merges the runs
- **run**: sorted subfiles as intermediate result of external sort

External Merge Sort

- B buffer pages available in memory
- need to sort file with N pages

```
ExternalSort(file)
    # given a file on disk, sort it using three buffer pages
    # produce runs B pages long: Pass 0
    Read B pages into memory
    Sort them
    Write out a run
    # Merge (B-1) runs at a time to produce longer runs, until only
    # one run (containing all input records) remains
    while number of runs at end of previous pass > 1:
      # pass i = 1, 2, ...
      while there are runs to be merged from previous pass:
          choose next (B-1) runs from previous pass
          read each run into an input buffer (page by page)
          merge runs and write to output buffer
          write output buffer to disk page by page
end ExternalSort
```

- pass 0: read in B pages at a time, sort internally to produce $\lceil N/B \rceil$ runs of B pages each
- in passes 1, 2, ...: use $B-1$ buffer pages for input, use remaining page for output: i.e. a $(B-1)$-way merge in each pass
- number of passes: $\lceil \log_{B-1} N \rceil + 1$

Overview of relational operator algorithms

Selection

- selection: $\sigma_{R.attropvalue}(R)$
- if there is no index on $R.attr$, we have to scan R
- if 1+ indexes on R match selection: use index to retrieve matching tuples, then apply any remaining conditions to restrict the result set
- rule of thumb: cheaper to scan entire table instead of using unclustered index if > 5% of tuples are to be retrieved

Projection

- drop certain fields
- expensive part is dropping duplicates (i.e. `DISTINCT` keyword used)

- to retrieve subset of fields
  - iterate over table
  - iterate over index whose key contains required fields. Irrelevant if clustered, as the values needed are in the data entries of index itself
- to eliminate duplicate, sort subset of fields, then remove adjacent duplicates

Evaluating Relational Operators