Evaluating an Access Method

- **Access Time**
  - Time to search for a data record

- **Insertion Time**
  - Time to insert a new record

- **Deletion Time**
  - Time to delete a record

- **Space Overhead**
  - Amount of disc space used by the method

- **B+-Trees**, extendible hashing indexes, clusters and denormalisation are all methods of improving the performance of a database. They are *access methods*.

- **Korth & Silberschatz** identify four properties that can be used to evaluate an access method:
  - **Access Time** The time it takes to locate a particular data record using the access method. For example, the time it takes to search a B+-Tree to locate a particular key value.
    In addition to the average access time it is important to consider the worst-case access time (the longest retrieval time) and the best-case access time (the shortest time). For example, a very bad worst-case time may be acceptable for some applications if the average access time is particularly good.
  - **Insertion Time** The time to insert a new key value into the structure. This includes the time to locate the position of the new key and any file reorganisation that must be performed. The average, worst and best performance is important.
  - **Deletion Time** The time to remove an existing key from the structure.
  - **Space Overhead** An index structure contains a large number of pointers and so it is important to consider how much space is required to store the structure. For example, a B+-Tree guarantees that every block be at least half full thus guaranteeing a maximum space overhead. Frequently, there is trade-off between performance and space overhead.

- Ref: Silberschatz sec 11.1.
### B⁺-Tree

- **In B⁺-Tree of \( lv \) levels:**
  - **Access Time**
    - Balanced
    - A search for a single key requires \( lv \) reads
  - **Insert Time**
    - An insert will require:
      - \( lv \) nodes to be read
      - A minimum of one node to be written
      - A maximum of \( (lv+2)*1 \) nodes to be written
    - Average is much less
  - **Space Overhead**
    - Each node is at least 50% (average 69%) full
    - Maximum of 50% (average 31%) ‘wasted’ space

- **Properties:**
  - The B⁺-Tree, with \( lv \) levels, has the following properties:
    - **Access Time** The tree is balanced. A search for an individual key will *always* require \( lv \) disc reads.
      - As the leaf nodes in the B⁺-Tree are ordered sequentially, sibling pointers can be used to perform very efficient range/sequential searches.
    - **Insert Time**
      - To perform an insert the position of the new key in the tree must be found by searching the index (\( lv \) reads).
      - Then,
        - A minimum of one node must be written (when the node is not full).
        - A maximum of \( (lv*2)+1 \) nodes must be written (when all the nodes in the search path are full, a split occurs at each level requiring two writes plus a new root node).
      - However, the average number of nodes is *much less* than the maximum.
    - **Space Overhead** The tree guarantees 50% occupancy of each node. That is, each node is at least half full. This is the worst-case performance. The average is 69% full.
    - Therefore, a B⁺-Tree has a guaranteed search time and space overhead and a very efficient insertion time.

### Extendible Hashing Index

- **Access Time**
  - An extendible hashing index has two levels
    - Directory + Bucket
- **Insert Time**
  - Minimum
    - Read the directory and one bucket
    - Write the bucket
  - Maximum
    - Read the directory and one bucket
    - Split and write two buckets and the directory
- **Space Overhead**
  - Does not guarantee occupancy of buckets

- **Properties:**
  - The extendible hashing index has the following properties:
    - **Access Time** The index has a single directory pointing to individual buckets (blocks). Therefore, a search for a single key requires only one bucket read (assuming the directory is in memory). However, if the directory becomes too large it may require a special organisation to store it efficiently.
      - Unless the hash function preserves the ordering of the key values when it produces the hash code, the extendible hash function is not useful for range/sequential searches.
    - **Insert Time**
      - To perform an insert the position of the new key must be found by searching the directory.
      - Then,
        - A minimum of one bucket must be written (when the bucket is not full).
        - The maximum number of nodes can be very large if doubling the directory does not distribute the contents of a bucket between the new buckets. This will depend on the hash function and its ability to differentiate between the keys.
      - However, the average number of buckets to be written is *much less* than the maximum, for example, two bucket.
    - **Space Overhead** The extendible hashing index does not split buckets in half. It uses the bits in a binary hash code. This does not guarantee that each bucket will have a minimum occupancy.
      - Therefore, an extendible hashing index has a very quick search time (directory + 1 bucket read) but provides poor space overhead guarantees.
### Clustering

<table>
<thead>
<tr>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Very fast for certain queries</td>
</tr>
<tr>
<td>- Joins between clustered relations</td>
</tr>
<tr>
<td>- Very slow for certain queries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Insert Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>- May require reorganising the cluster values.</td>
</tr>
<tr>
<td>- May require a cluster index to be updated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Allocating each cluster key to a single block may waste space.</td>
</tr>
</tbody>
</table>

- The cluster has the following properties:
  - **Search Time** The performance of a cluster depends on the relations being clustered and queried:
    - When a search is joining two relations that are clustered, the performance should be good. The data is already joined on the disc.
    - When a search is joining a clustered relation with a non-clustered relation, the performance will be poor as the clustered relation is not stored contiguously on the disc.
  - **Insert Time** Inserting into a cluster may require creating a new block when a new cluster key value is added.
    - Changing the value of a cluster key will involve reorganising the records in the cluster.
    - A cluster may also have a cluster index that must be updated.
  - **Space Overhead** In many systems each cluster key will be allocated its own disc block. If there are relatively few records per cluster key then a lot of space in each disc block will remain unused. A cluster only stores the cluster key value once for all records with the cluster key value. This may save some space.

- Therefore, a cluster provides very fast search times for queries that are based on the cluster structure (joining two relations that are clustered) but very poor search times for queries that are not based on the clustered relations. Inserting can require reorganisation of the records and may require space to be allocated in advance.

### Denormalisation

<table>
<thead>
<tr>
<th>Search Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Denormalisation is performed to improve the performance of specific queries.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Insert Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Denormalisation frequently introduces redundancy that increases insert times.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Denormalisation introduces redundancy that increases space used.</td>
</tr>
<tr>
<td>- Certain denormalisations may reduce space used.</td>
</tr>
</tbody>
</table>

- Denormalisation is not an access method but is an organisation technique.

- Denormalisation has the following properties:
  - **Search Time** Denormalisation is performed to improve the performance of specific queries or updates. Therefore, it will improve the search time of certain queries but, like clustering, will make other queries perform poorly.
  - **Insert Time** Denormalisation introduces data redundancy that requires more than one insert per update to avoid update anomalies. It may mean that some data cannot be represented, for example, update anomalies.
  - **Space Overhead** The redundancy introduced by denormalisation will require extra space overhead but this is not always the case. Denormalisation may reduce the storage required for a set of relations.
    - For example, in the detailed data example (see notes on denormalisation) normalisation introduces a very large primary key that is duplicated in each tuple of a relation. The denormalised version reduces the overall storage requirement but is more difficult to process using SQL.

- Therefore, denormalisation, like clustering, is a method of organising data to improve performance. The main disadvantages of denormalisation are that it may introduce redundancy and make the relations difficult to process. In addition, unlike B*-Trees, hash indexes and clustering, a denormalised set of relations is visible to the end-user.
**Scenario A**

- One relation
  - `employee(eno, ename, sal)`

- Queries
  - All the employee tuples are read to produce the pay slips.

- Updates
  - New employees are added infrequently.
  - Employee details are updated infrequently.

**B+-Tree**

A B+-Tree allows individual tuples to be accessed quickly. In this scenario, the only query accesses the whole of the relation and there is no indication that the data must be ordered. A B+-Tree is not required. If the data was to be ordered then a B+-Tree would improve the speed of sorting the data on the index key. A B+-Tree could be used to enforce the integrity of the primary key.

**Extendible Hashing**

Same as B+-Tree but the extendible hash index will not help to sort the data as the keys in the extendible hash index are not ordered.

**Clustering**

The database only contains one relation therefore a cluster is not required. Oracle, for example, will store all the tuples in `employee` contiguously on the disc when possible.

**Denormalisation**

Same as clustering.

---

**Scenario A - Alternatives**

- Sequential query of all data
  - Indexing not needed
  - Potential overhead
    - If query is ordered - B+-Tree may be useful
      - Leaf nodes sorted
  - Clustering may be useful
    - Cluster data in one location
      - Reduce number of data blocks read
      - Only one table - may waste lots of space
        - “The Adam Ward Effect”
    - In Oracle
      - Create one extent (contiguous data blocks)
        - Works because there are few updates
Scenario B

- One relation
  - employee(eno, ename, sal)
- Queries
  - All the employee tuples are read to produce the pay slips.
  - Individual employee tuples are read frequently using the eno.
- Updates
  - New employees are added infrequently.
  - Employee details are updated infrequently.

B+-Tree
Same as Scenario A except that a B+-Tree on eno will allow fast access to individual employee tuples with little insert/update overhead.

Extendible Hashing
Same as B+-Tree. The extendible hashing index will perform as well as the B+-Tree if a good hash function can be found for eno.

Clustering
Same as Scenario A.

Denormalisation
Same as Scenario A.

Scenario B - Alternatives

- Sequential & exact query
  - Indexing useful for exact query
    - Extendible hashing
      - Very fast
      - Not good for sequential query
    - B+-Tree
      - Fast exact search
      - Fast sequential search
        - No sort required
        - Leaf nodes sorted
  - Clustering may be useful
    - Create one extent (contiguous data blocks)
      - Works because there are few updates
Scenario C

- Two relations
  - employee(eno, ename, sal, pno)
  - project(pno, pname, cost)
- Queries
  - A monthly report on the cost of all projects
  - Ad hoc report of employees on specific project.
- Updates
  - New employees are added infrequently.
  - Employee details are updated frequently.
  - Project details are added infrequently.

B+-Tree
A B+-Tree on the primary and foreign keys will improve the performance of queries that join the two relations and will allow fast access to individual projects or employees. The sequential order of the keys in the index makes certain joins very efficient, for example, the merge-sort join.

Extendible Hashing
Same as B+-Tree. The data may have to be sorted to perform the join as the extendible hashing index does not support a sequential searching.

Clustering
Clustering on pno in employee and project will allow the join between the two relations to be performed very quickly. This is because the data will be stored as a joined set of data on the disc. The cluster index on pno will not improve the performance of queries on employee alone. A B+-Tree could be added to the eno attribute in employee but as the employee data is clustered around projects access to each employee may involve reading a different cluster block.

Denormalisation
The two relations could be denormalised into a single employee_project relation but it would contain redundant project details. This would mean that joining the two relations was unnecessary and a B+-Tree on eno would be an efficient method of indexing the relation to answer queries about individual employees. However, the redundancy in the project details will make updates of the project information slower although these updates are infrequent. More importantly, an update anomaly would occur in that it would not be possible to create a project without first allocating employees to it.

Scenario C - Alternatives

- Clustering may be useful
  - Stores data from multiple tables
  - Better for “report of employees on project”
    - Locating employees on one project
    - Cluster on project ID
    - Accessing one data block
  - Poor update performance
    - “Employee details are updated frequently”

- Indexing
  - Any index type on primary/foreign keys
    - Improves most join types

- Denormalisation
  - Single table emp_proj
    - Good performance
    - Lots of redundancy
    - Frequent updates may cause problems
### Search Time

<table>
<thead>
<tr>
<th>Method</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>B⁺-Tree</td>
<td>Guarantees the search time as it is balanced but may involve reading a number of nodes. It orders the data and will perform sequential searches very efficiently.</td>
</tr>
<tr>
<td>Extendible Hashing</td>
<td>Guarantees the search time as only the directory must be read which points directly to the buckets. However, it does not order the data.</td>
</tr>
<tr>
<td>Clustering</td>
<td>Does not guarantee search time but will substantially improve the performance of certain queries (joins) while substantially reducing the performance of other queries. Certain sequential searches may be slow.</td>
</tr>
<tr>
<td>Denormalisation</td>
<td>Does not guarantee search time but will substantially improve the performance of certain queries while substantially reducing the performance of other queries.</td>
</tr>
</tbody>
</table>

### Insert Time

<table>
<thead>
<tr>
<th>Method</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>B⁺-Tree</td>
<td>Guarantees the worst-case insert time (splitting all the nodes in the search path including the root). Average insert time is very low (insert in single node).</td>
</tr>
<tr>
<td>Extendible Hashing</td>
<td>Worst-case insert time could be very poor (continually splitting buckets until the keys can be distributed). Average will require inserting in one or two nodes.</td>
</tr>
<tr>
<td>Clustering</td>
<td>Inserting could require the clustered records to be reorganised. To overcome this space is often allocated in advance to allow records to be inserted efficiently.</td>
</tr>
<tr>
<td>Denormalisation</td>
<td>Frequently, introduces redundancy that may require data to be updated in more than one location. May reduce the time taken to insert as the overall space allocation may be reduced.</td>
</tr>
</tbody>
</table>
### Space Allocation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B* Tree</td>
<td>Guarantees 50% occupancy of all nodes (except root). The average is better, 60%.</td>
</tr>
<tr>
<td>Extendible Hashing</td>
<td>Does not guarantee the occupancy of buckets. Therefore, the index could become very sparse and caching, for example, will not be as useful.</td>
</tr>
<tr>
<td>Clustering</td>
<td>To avoid frequent reorganizations, space may have to be allocated in advance. But, a cluster saves space when storing the cluster key value by storing it once.</td>
</tr>
<tr>
<td>Denormalisation</td>
<td>Denormalisation may save space when required, for example, storing detail data (see lecture notes). When query speed is important, denormalisation may use more space.</td>
</tr>
</tbody>
</table>