1 Query Processing

Given a query, the DBS **translates** a query stated in a high-level language such as SQL, into (extended) relational algebra, **optimize** it, and then **evaluate** the query.

The translation from high-level language to SQL involves the usual language interpretation phases **scanning** (finding language tokens), **parsing** (building a parse tree), and **semantic analysis** (verify existance of relations mentioned in statements, etc.). **Views** are also replaced with the relational algebra expression to compute the view.

Translating from SQL, which is declarative (goal explicit, algorithm implicit), to relational algebra can be done in several ways, e.g.

```
SELECT age FROM person WHERE age >= 18
```

could be translated to one of

$$\sigma_{\text{age}\geq 18}(\Pi_{\text{age}}(person))$$
$$\Pi_{\text{age}}(\sigma_{\text{age}\geq 18}(person)).$$

Furthermore, relational algebra expressions are analyzed by the **query optimizer** to find the optimum execution, e.g. transforming the query to an equivalent but more efficient query, and whether to use indexes or not. The result of this analysis is a **query evaluation plan**, which can be represented graphically as a tree decorated with instructions to the **query execution engine**, which executes the query and returns the results.

2 Cost

The cost of evaluating a query is measured by **CPU cost**, i.e. algorithm complexity, and **disk access**, i.e. seek time, blocks read, blocks written. We focus on disk access, as it often is the bottle-neck of DBSes, and simplify the disk operation to **number of block transfers**.

3 Sorting

If records are sorted on disk, we can gain performance in queries on the sorting attribute. The records of a relation may not fit in memory, so we use the **external mergesort** algorithm. If we can fit M blocks of a relation in memory, the external mergesort algorithm is as follows:

```
i = 0
repeat
    read M blocks (or whatever is left) of the relation into memory
    sort the M blocks using quicksort (divide-and-conquer sort alg.)
    write M blocks to file F_i
    i++
until end of relation

read 1 block from each file F_i into memory
repeat
    choose the first of all the blocks and write it to output
    replace that block with the next block from the file F_i
until all blocks from all files was read
```

If there are more files F_i than memory M, the algorithm must be performed in multiple passes - sort the first M files to 1 output file, etc.

4 Evaluating Selection

4.1 File Scan

Evaluating a point query selection (equality($\sigma_{attr=value}$(relation))) on a relation which is stored in a single file can be done by **binary search** if the file is sorted. Binary search rules out half the tuples for each iteration, resulting in a cost of $log_2\ b$, where b is the number of disk blocks in the file. If the file is not sorted, we must use **linear search**, which has average cost $b/2$ and worst-case cost b.

4.2 Index Scan

If we have an index on the search key (selection attributes), we can do an **index scan**. I assume that the indexes are B$^+$-trees.

- If we search in a primary index (an index on the attributes on which the file is sorted), we can do the operation accessing $h+r$ disk blocks, where h is the height of the tree and r is the number of records in the result ($r = 1$ if the search key is a superkey).
- If we search in a secondary index (the file is not sorted on the index attributes) and the search key is a superkey, we can do it accessing $h + 1$ blocks. If the search key is not a superkey, each index entry

points to a bucket of pointers to records. So we have to read the bucket record and the record itself. Worst-case, this may result in $h + 2r$ disk access (potentially worse than linear scan!)

Range queries like $\sigma_{\text{age}\geq 18}(\text{person})$ can be done effeciently with a primary index, as we may find the first record where $age = 18$ and retreive all the remaining records.

5 Evaluating Joins

In the general case, we can use the **Nested-Loop Join** to evaluate the theta join $r \bowtie_\theta s$:

```
for each tuple tr in r
    for each tuple ts in s
        if tr theta ts, add tr.ts (concatenation) to the result
```

It can always be used, but is expensive: in the worst case it uses blocks(r) + tuples$(r) \cdot$ blocks(s) disk accesses. In the best case, where s fits in memory, it uses blocks(r) + blocks(s).

The **Block Nested-Loop Join** is an improvement, as it tests per block, not per tuple, which yields a worst case of blocks(r) + blocks$(r) \cdot$ blocks(s):

```
for each block Br of r
    for each block Bs of s
        for each tuple tr in Br
            for each tuple ts in Bs
                if tr theta ts, add tr.ts to the result
```

If we have an index on the join attribute in the **inner relation** (r), we can use the **Indexed Nested-Loop Join**, where we use index lookups instead of file scan on the join attributes for the inner relation. The cost would then be blocks(r) + tuples$(r) \cdot$ cost of selection on(s).

The **Merge Join** algorithm uses the external mergesort algorithm (described earlier) to sort the relations by the join attributes, and then use a procedure similar to the 'merge' to perform the join (a binary search).

The **Hash Join** algorithm partitions the 2 relations in 'buckets' by hash value on the join attributes, and then perform a separate indexed nested-loop join on each 'bucket'.