Kim-Reino Hjelde
Kimreinh
IN3020 Assignment 1

Exercise 1

a)

```
SELECT s.Course_number, s.Semester, s.Yearr, count(gr.Section_Identifier) as stdNr
FROM Section s, Grade_report gr
WHERE s.Instructor = "King"
  AND s.Section_Identifier = gr.Section_Identifier
GROUP BY s.Course_number;
```

b)

```
SELECT s.Name, s.Major
FROM Student s JOIN Grade_Report gr
  ON s.Student_number = gr.Student_number
GROUP BY s.Name, s.Major
HAVING SUM(case when gr.Grade <> 'A' then 1 else 0 end) = 0;
```

c)

```
INSERT INTO Student (Name, Student_number, Class, Major)
VALUES ("Smith", 17, 1, "CS");
```

d)

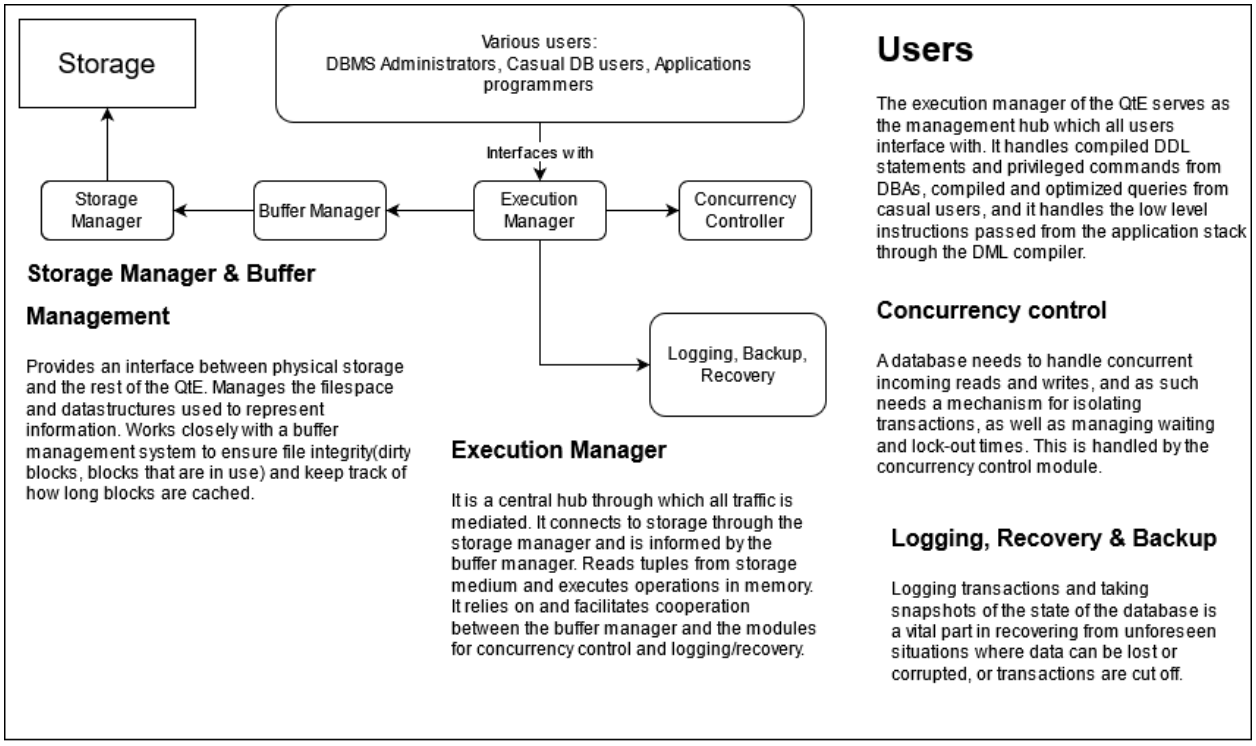
```
UPDATE Student s
SET s.Class = 2
WHERE s.Name = 'Smith';
```

e)

```
DELETE FROM Student
WHERE Name = 'Smith'
  AND Student_number = 17;
```

Exercise 2

2.1



2.2

The central components of a DBMS are modules that directly interact with the operating system interface to manage their resources. The modules for system buffer management, lock management, and logging, are central components. Higher-level components on the other hand, are components that rely on central components as mediators between themselves and the resources they interact with. An example of this is transaction management and access path management systems.
The three central components interact in three different ways. Log←→ Buffer, Log←→ Lock, Lock←→ Buffer.

Exercise 3

3.1

Balanced binary search tree:
Multi-level index arranged in a tree-structure, can maintain a low number of levels, typically 3, and still keep track of a very large number of records. However, all searches

must start from the root node and number of block accesses is determined by the height of the tree. Leaf nodes are all of equal distance to the root node, and are linked. Good support for random access and sequential access, fast interval search. Disk I/O is very expensive, and can be reduced by keeping some index blocks in memory. Further, it is a very well supported data structure, with plenty of standardized solutions and backing.

Hash table:
Under ideal conditions where all elements of a hash value fits into one bucket block, hash tables require significantly fewer disk operations than B+ trees, and specific key searches are fast. However, scaling becomes problematic if the number of records increases while the hash table remains static, causing overflow. These cases can be mitigated with dynamic hash tables. Hash tables are inappropriate for interval searches.

Bitmaps: Often used in read-only databases that are specialized for fast queries because they are less efficient than B+ trees when data is frequently updated. Operates with bit arrays on which logical bitwise operations are performed to answer queries. As such they are excellent in terms of space usage and performance. However, bitmaps are only appropriate for certain kinds of data, namely data that has a low number of distinct values, the obvious example being booleans.

In situations where one needs to store and query data that represents multi-dimensional objects instead of simple numerics and characters, such as an object in geometric space or geographical coordinates, data-structures that use multi-dimensional indexing are used:

K-dimensional trees perform well for range and nearest-neighbour searches(multi-dimensional search keys).
Quad-trees partition two-dimensional space into 4 quadrants, which can be used to good effect in image compression.
R-trees differ from kd-trees in that they represent bounding boxes instead of planes, and these regions may overlap. A kd-tree on the other hand are disjoint, meaning points belong to only one region. This means R-trees are excellent at representing geographical spaces, where searches for intersections, contained-in and nearest neighbour is highly relevant.

For multi-dimensional tree-structures, balancing is often complicated, which means insertion poses a challenge.

3.2

Inverted indexing is used if you want to map from content to location in a table, for example implemented in text processing. With inverted indexes you can look up every instance of a search word dispersed over multiple documents. The prime example of this is a search engine that locates documents containing a given keyword.

3.3
A linked-list of overflow records for each block

Exercise 4

4.1

a)

Think of WHERE IN as a join instead of subquery. Select with relevant conditions from the cartesian product. Project the required attribute from selection

$$\pi_{S.A}(\sigma_{P.A = S.A \wedge S.D > 0}(P \times S))$$

b & c)

$$\pi_{S.A}(P \bowtie_{S.D > 0} S)$$

Here I have combined selection and cartesian product into the appropriate join on condition. This essentially pushes selection and reduces the amount of tuples that are processed.

4.2

a)

$$\pi_M(\sigma_{<c> \wedge <d>}(P \bowtie Q \bowtie R)) = \pi_M((P \bowtie_{<c> \wedge <d>} Q) \bowtie R)$$

$$\pi_M((P \bowtie_{<d> \wedge <c>} Q) \bowtie R) = \pi_M(\{(\sigma_{<c>}(P)) \bowtie_{<d>} Q)\} \bowtie R)$$

$$\pi_M(\{(\sigma_{<c>}(P)) \bowtie_{<d>} Q)\} \bowtie R) = \pi_M(\{(\sigma_{<c>}(P)) \bowtie (\sigma_{<d>}(Q))\} \bowtie R)$$

$$\pi_M(\{(\sigma_{<c>}(P)) \bowtie (\sigma_{<d>}(Q))\} \bowtie R) = \pi_M[(\{(\pi_A(\sigma_{<c>}(P)) \bowtie \pi_{A,H}(\sigma_{<d>}(Q))\} \bowtie R)]$$

b)

I believe that the second expression will perform much better than the initial expression. In general one wants to split select with multiple conditions, such as in this case, and push the selections further down the tree. The transformed expression does this. This results in smaller temporary relations in cache, and pushed selections and projections give smaller pruned relations to compare in the joins. Less space and fewer operations.

Exercise 5

5.1

SQL Query → Parse tree → Convert to LQP → Improve LQPs → Estimate size → Construct PQPs → Estimate costs → pick the best and execute

An **SQL query** is passed, and it is decomposed into a **parse tree** using context-free grammar. This is also where it is ensured that the query is syntactically and semantically correct. The parse tree is then handed along and used to generate a **Logical Query Plan** with relational algebra expressions. LQP is a conceptual execution plan for the given SQL statement. After the LQP is generated, **algebraic laws** are applied to the relational algebra expression in order to **improve** it, for example by reducing the number of tuples by pushing selections, splitting conditions into multiple selections which are then pushed down the tree, managing duplicate eliminations and such. The **improved LQP** is then judged in terms of the size of the relations. Approximate **estimates of size** are given by a set of simple and consistent rules and calculations which are applied to the relevant LQPs.

Physical Query Plans are constructed from the proposed LQPs, and they differ from LQPs in that they are the proposed actual execution plans rather than just conceptual. After having determined the order of operators, choices must be made regarding what algorithms to use for implementation, how to pass the intermediate results between operators and so on. The estimation of costs of PQPs is also important of course, and there is overlap in the estimations of LQPs and PQPs. Some of the variables that constitute ‘cost’ is amount of disk IO necessary, number of tuples to process, estimates of size in bytes. Some approaches to **choosing a PQP plan** are: exhaustive, heuristic, hill-climbing. A PQP plan cannot be evaluated in isolation, and knowledge of the system is taken into account, for example considering memory requirements. Finally, the PQP that has been deemed best is passed and executed as the final step.

5.2
a)

```
SELECT c.Course_name, s.Semester, s.Year, s.Instructor
FROM Course c JOIN Section s
  ON c.Course_number = s.Course_Number
WHERE c.Course_name = "Database Systems"
  AND s.Year > 2010
```

b & c)

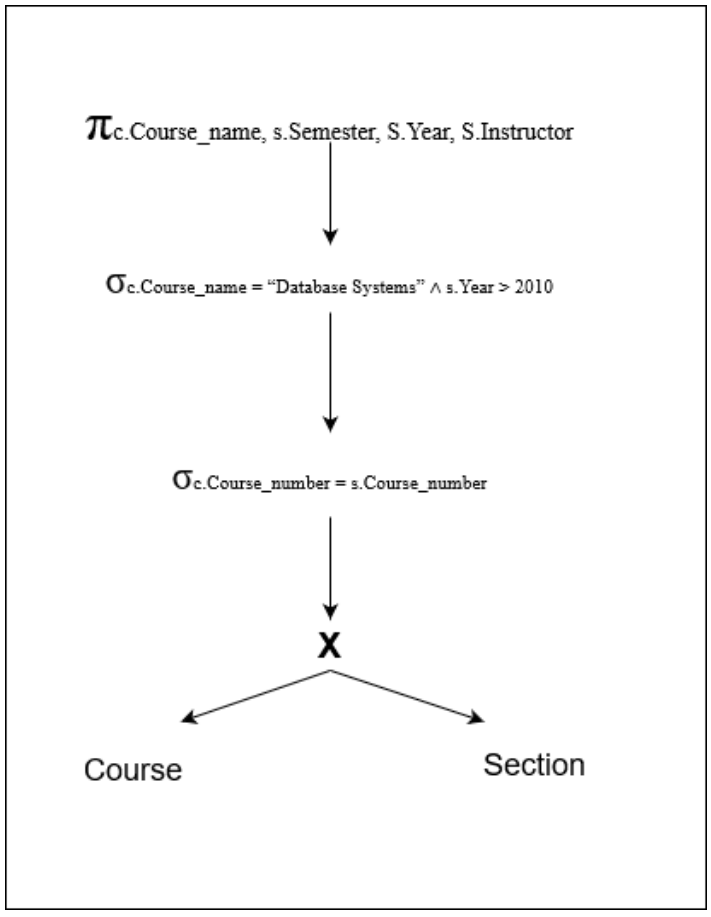
$\Pi_{c.Course\_name,\ s.Semester,\ s.Year,\ s.Instructor}$

$(\sigma_{c.Course\_name = \text{“Database Systems”} \wedge}$

$s.Year > 2010$

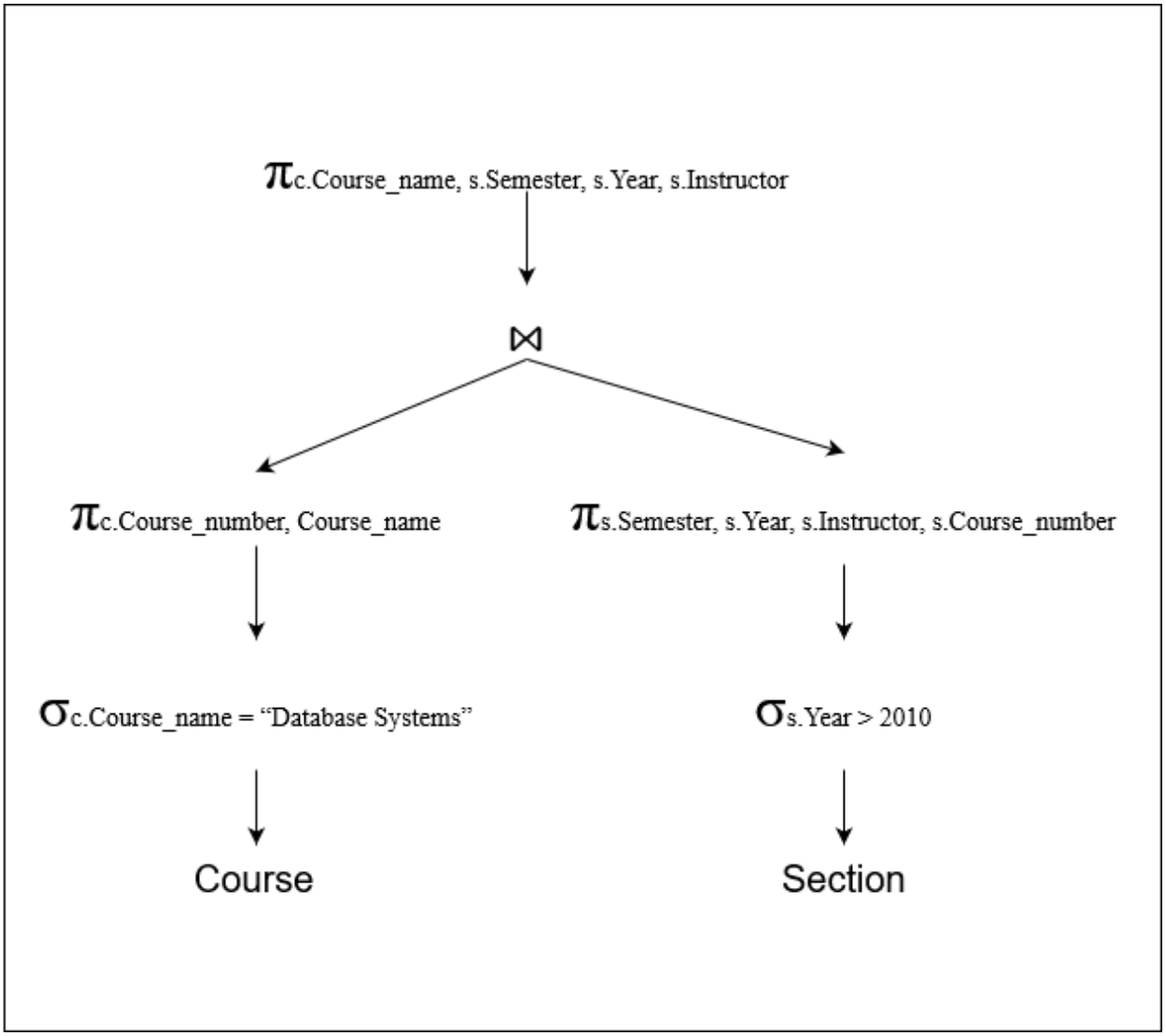
$(\sigma_{c.Course\_number = s.Course\_number}$

$(CourseXSection)))$



d)

$$\pi_{c.Course\_name,\ s.semester,\ s.Year,\ s.Instructor}\Big(\pi_{c.Course\_number,\ c.Course\_name}\Big(\sigma_{c.Course\_name\ =\ \text{“Database Systems”}}(Course)\Big) \bowtie \big(\pi_{s.Semester,\ s.Year,\ s.Instructor,\ s.Course\_number}\Big(\sigma_{s.Year\ >\ 2010}(Section)\Big)\big)\Big)$$



e)

Rough calculations on the initial tree, based on relations from exercise 1:
Tup(Course X Section) = 4 * 6 = 24
TSize(Course X Section) = $TSize_{Course}$ + $TSize_{Section}$ = 192 bytes (assuming 6x 30byte varchar and 3x 4byte int)
Size(Course X Section) = 24 * 192bytes = 4608 bytes.
24 operations with R(Course X Section) = 4608 bytes cached.
Then a selection with 1 condition checks each tuple, another 24 operations.
From here the number of operations will be negligible in comparison, assuming only a few tuples satisfy condition. Another selection occurs, and finally a projection. These will be cheap.

Rough calculations on the improved tree:
Check each tuple in Course, 4 operations.
Cache the R resulting from selection, 1 tuple = 94bytes (assuming 3x 30byte varchar +1x 4byte int). Assuming very few tuples will be selected.
Project 2 attributes, checking very few tuples and keeping 2x30 bytes per tuple.

Check each tuple in Section, 6 operations.
Cache the R resulting from selection, 1 tuple = 98 bytes. Assuming few tuples.
Project 4 attributes, keeping 68 bytes per tuple.

At this point with the ⋈ I am assuming the pushed selections/projections have trimmed away the vast majority of tuples, resulting in a projection from a very small joined relation. Even with such rough estimates I am confident in asserting that the improved tree caches significantly fewer bytes with its relations and temporary relations compared to the potentially massive cartesian product. There are fewer operations involved also.