B561 Assignment 6
Data Generation
Sorting
Indexing

Dirk Van Gucht

For this assignment, you will need the material covered in the lectures

- Lecture 14: External Merge Sorting
- Lecture 15: Indexing
- Lecture 16: B^+ trees and Hashing

In addition, you should read the following sections in Chapter 14 and 15 in the textbook *Database Systems The Complete Book* by Garcia-Molina, Ullman, and Widom:

- Section 14.1: Index-structure Basics
- Section 14.2: B-Trees
- Section 14.3: Hash Tables
- Section 14.7: Bitmap Indexes

In the file `performingExperiments.sql` supplied with this assignment, we have include several PostgreSQL functions that should be useful for running experiments. Of course, you may wish to write your own functions and/or adapt the functions in this .sql to suite the required experiments for the various problems in this assignment.

The problems that need to be included in the `assignment6.sql` are marked with a blue bullet •. The problems that need to be included in the `assignment6.pdf` are marked with a red bullet •. (You should aim to use Latex to construct your .pdf file.) In addition, submit a file `assignment6Code.sql` that contains all the sql code that you developed for this assignment.

1 Data generation

PostgreSQL functions and clauses In this assignment, you will need to conduct experiments with random data generated relations of various sizes. PostgreSQL provides useful functions and clauses to generate such relations:

`random()`	returns a random real number between 0 and 1 using the uniform distribution
`floor(random() * (u - l + 1) + l) :: int`	returns a random integer in the range $[l, u]$ using the uniform distribution
`generate_series(m, n)`	generates the set of integers in the range $[m, n]$
`order by random()`	randomly orders the tuples that are the result of a query
`limit(n)`	returns the first n tuples from the result of a query
`offset(n)`	returns all but the first n tuples from the result of a query
`row_number()`	is a *window function* that assigns a sequential integer to each row in a result set
`vacuum`	is a *garbage collection* function to clean and reclaim secondary memory space

For more detail, consult the manual pages

https://www.postgresql.org/docs/13/functions-math.html
https://www.postgresql.org/docs/current/functions-srf.html
https://www.postgresql.org/docs/current/queries-limit.html
https://www.postgresql.org/docs/8.4/functions-window.html
https://www.postgresql.org/docs/9.5/routine-vacuuming.html

Generating sets To generate a set, i.e., a unary relation, of n randomly selected integers in the range $[l, u]$, you can use the following function:[1]

```
create or replace function SetOfIntegers(n int, l int, u int)
    returns table (x int) as
$$
    select floor(random() * (u-l+1) + l)::int from generate_series(1,n);
$$ language sql;
```

Example 1 *To generate a unary relation with* 3 *randomly selected integers in the range* 5 *to* 10*, do the following:*

```
select x from SetofIntegers(3,5,10);
```

Of course, running this query multiple times, result in different sets.

[1]Typically the function `SetOfIntegers` returns a bag (multiset) but this is fine for this assignment. In case we want a set, we can always eliminate duplicates.

Generating binary relations The idea behind generating a set can be generalized to that for the generation of a binary relation.[2] To generate a binary relation of n randomly selected pairs of integers (x, y) with $x \in [l_1, u_1]$ and $y \in [l_2, u_2]$, you can use the following function:

```
create or replace function
BinaryRelationOverIntegers(n int, l_1 int, u_1 int, l_2 int, u_2 int)
   returns table (x int, y int) as
$$
   select floor(random() * (u_1-l_1+1) + l_1)::int as x,
          floor(random() * (u_2-l_2+1) + l_2)::int as y
   from   generate_series(1,n);
$$ language sql;
```

Example 2 *To generate a binary relation with* 20 *randomly selected pairs with first components in the range* $[3, 8]$ *and second components in the range* $[2, 11]$*, do the following:*

```
select x, y from BinaryRelationOverIntegers(20,3,8,2,11);
```

Generating functions A relation generated by `BinaryRelationOverIntegers` is in general not a function since it is possible that the relation has pairs (x, y_1) and (x, y_2) with $y_1 \neq y_2$. To create a (partial) *function* $f : [l_1, u_1] \to [l_2, u_2]$ of n randomly selected function pairs, we can use the following function:[3]

```
create or replace function
FunctionOverIntegers(n int, l_1 int, u_1 int, l_2 int, u_2 int)
   returns table (x int, y int) as
$$
   select x, floor(random() * (u_2-l_2+1) + l_2)::int as y
   from   generate_series(l_1,u_1) x order by random() limit(n);
$$ language sql;
```

Example 3 *To generate a partial function* $[1, 20] \to [3, 8]$ *of* 15 *randomly selection function pairs, do the following:*[4]

```
select x, y from FunctionOverIntegers(15,1,20,3,8);
```

[2] Clearly, all of this can be generalized to higher-arity relations.

[3] Observe that when $n \geq (u_1 - l_1)$, f is a total function.

[4] When using this function, it is customary to use n such that $n \in [0, u_1 - l_1 + 1]$.

Generating relations with categorical (non-numeric) data Thus far, the sets, binary relations, and functions have all just involved integer ranges. It is possible to include ranges that have different types including categorical data such as text strings. The technique to accomplish this is to first associate with a categorical range an integer range that associate with each element in the categorical range a unique value of the integer range. The following example illustrates this.

Example 4 *Consider the* `jobSkill` *relation and assume that it contents is*

skill
AI
Databases
Networks
OperatingSystems
Programming

Suppose that we want to generate a `personSkill(pid, skill)` *relation. Let us assume that the* `pid`*'s are integers in the range* $[1, m]$.

There are 5 *skills in the* `jobSkill` *and it is therefore natural to associate with each skill a separate integer (index value) in the range* $[1, 5]$. *This can be done with a query involving the* `row_number()` *window function:*

```
select skill, row_number() over (order by skill) as index
from   Skill;
```

The result is the following relation:

skill	index
AI	1
Databases	2
Networks	3
OperatingSystems	4
Programming	5

Using this technique, we can write a PostgreSQL function that generates a `personSkill` *relation with* n *randomly selected* $(pid, skill)$ *tuples, with* `pid`*'s in the range* $[l, u]$*:*

```
create or replace function GeneratepersonSkillRelation(n int, l int, u int)
returns table (pid int, skill text) as
$$
with skillNumber(skill, index) as (select skill, row_number() over (order by skill)
                                   from   Skill),
     pS as (select x, y
            from   BinaryRelationOverIntegers(n,l,u,1, (select count(1) from Skill)::int))
select x as pid, skill
from   pS join skillNumber on y = index
group by (x, skill) order by 1,2;
$$ language sql;
```

In this function, the `skillNumber` *view associates with each job skill an integer index in the range* $[1, |\texttt{Skill}|]$. *The* `pS` *view is a randomly generated binary relation with* n *tuples, with* `pid`*'s in the range* $[l, u]$, *and skill numbers in the range* $[1, |\texttt{Skill}|]$. *The* `join` *operation associates the numeric range with the* `Skill` *range. The '*`group by (x, skill) order by 1,2`*' clause eliminates duplicate tuples and orders the result.*

The query

```
select * from GeneratepersonSkillRelation(10,1,15);
```

may make the `personSkill` *relation:*

pid	skill
1	AI
2	Programming
3	Databases
4	Databases
6	Networks
6	OperatingSystems
6	Programming
9	Databases
14	Databases
14	Networks

Problems We now turn to the problems in this section.

1. • Given a discrete probability mass function P, as specified in a relation `P(outcome: int, probability: float)`, over a range of possible outcomes $[u_2, l_2]$, design a PostgreSQL function

   $$\texttt{RelationOverProbabilityFunction}(n, l_1, u_1, l_2, u_2)$$

   that generates a relation of up to n pairs (x, y) such that

   - x is uniformly selected in the range $[l_1, u_1]$; and
   - y is selected in accordance with the probability mass function P in the range $[l_2, u_2]$.

   An example of a possible P as stored in relation `P` is as follows:[5]

   P

outcome	probability
1	0.25
2	0.10
3	0.40
4	0.10
5	0.15

   Note that when P is the uniform probability mass function, then

   `RelationOverProbabilityFunction`

   and

   `BinaryRelationOverIntegers`

   are the same binary-relation-producing functions.

   Hint: For insight into this problem, consult the method of *Inverse Transform Sampling* for discrete probability mass functions.

   Test your function for the following cases:

   (a) Test case 1: uniform mass function

   P

outcome	probability
1	0.125
2	0.125
3	0.125
4	0.125
5	0.125
6	0.125
7	0.125
8	0.125

[5] Notice that the sum of the probabilities in the `probability` column in `P` sum to 1.

```
select * from RelationOverProbabilityFunction(100, 1, 150, 1, 8);
```

You actually may wish to run the above query multiple times to test that your function works as expected.

(b) Test case 2: non-uniform function

P	
outcome	probability
1	0.25
2	0.05
3	0.10
4	0.10
5	0.15
6	0.05
7	0.10
8	0.20

```
select * from RelationOverProbabilityFunction(100, 1, 150, 1, 8);
```

You actually may wish to run the above query multiple times to test that your function works as expected.

2. • Use the technique in Problem 1 and the method for generating categorical data discussed above to write a PostgreSQL function that generates a `personSkill` relation, given a probability mass function over the `Skill` relation.

   Your function should work for any `Skill` relation and any probability distribution defined over it.

   Provide test cases and run test to demonstrate that your solution works.

2 Sorting

We have learned about *external sorting*. The problems in this section are designed to look into this sorting method as it implemented in PostgreSQL.

3. • Create successively larger sets of n randomly selected integers in the range $[1, n]$. You can do this using the following function.[6]

```
create or replace function makeS (n integer)
returns void as
$$
begin
    drop table if exists S;
    create table S (x int);
    insert into S select * from SetOfIntegers(n,1,n);
end;
$$ language plpgsql;
```

This function generates a bag S of size n, with randomly select integers in the range $[1, n]$. Now consider the following SQL statements:

```
select makeS(10);
explain analyze select x from S;
explain analyze select x from S order by 1;
```

- The '`select makeS(10)`' statement makes a bag `S` with 10 elements;
- The '`explain analyze select x from S`' statement provides the query plan and execution time in milliseconds (ms) for a simple scan of `S`;
- The '`explain analyze select x from S order by 1`' statement provides the query plan and execution time in milliseconds (ms) for sorting `S`.[7]

```
    QUERY PLAN
----------------------------------------------------------------------------------------------------
 Sort  (cost=179.78..186.16 rows=2550 width=4) (actual time=0.025..0.026 rows=10 loops=1)
   Sort Key: x
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on s  (cost=0.00..35.50 rows=2550 width=4) (actual time=0.004..0.005 rows=10 loops=1)
 Planning Time: 0.069 ms
 Execution Time: 0.034 ms
```

Now construct the following timing table:[8]

[6] You should make it a habit to use the PostgreSQL `vacuum` function to perform garbage collection between experiments.

[7] Recall that 1ms is $\frac{1}{1000}$ second.

[8] It is possible that you may not be able to run the experiments for the largest `S`. If that is the case, just report the results for the smaller sizes.

size n of relation `S`	avg execution time to scan `S` (in ms)	avg execution time to sort `S` (in ms)
10^1		
10^2		
10^3		
10^4		
10^5		
10^6		
10^7		
10^8		

The following tables show the results of experiments for sorting problems, and this for various sizes of the working memory which can be thought of as the buffer size. The experiments were done on a Mac Mini with 2 main-memory modules of 8GB each and 256 SSD disk storage. (The 2 main memory modules permit parallel processing.) All experiments where done with PostgreSQL Version 13.

```
set work_mem = '4MB';
set max_parallel_workers = 0;

 size of relation S | avg execution time to scan S | avg execution time to sort S
--------------------+------------------------------+------------------------------
                 10 |                        0.009 |                        0.015
                100 |                        0.020 |                        0.044
               1000 |                        0.150 |                        0.384
              10000 |                        1.334 |                        3.668
             100000 |                       13.323 |                       47.916
            1000000 |                      135.008 |                      528.670
           10000000 |                     1387.183 |                     6376.019
          100000000 |                    13985.228 |                    95999.630
```

(a) What are your observations about the query plans for the scanning and sorting of such differently sized bags `S`? In particular, discuss the different sorting algorithms that appear in the query plans and why they can or must be used.

The following tables show the results of experiments for sorting problems, and this for various sizes of the working memory which can be thought of as the buffer size. The experiments were done on a Mac Mini with 2 main-memory modules of 8GB each and 256 SSD disk storage. (The 2 main memory modules permit parallel processing.) All experiments where done with PostgreSQL Version 13.

```
set work_mem = '4MB';
set max_parallel_workers = 0;

 size of relation S | avg execution time to scan S | avg execution time to sort S
--------------------+------------------------------+------------------------------
                 10 |                        0.009 |                        0.015
                100 |                        0.020 |                        0.044
               1000 |                        0.150 |                        0.384
```

```
    10000 |        1.334 |     3.668
   100000 |       13.323 |    47.916
  1000000 |      135.008 |   528.670
 10000000 |     1387.183 |  6376.019
100000000 |    13985.228 | 95999.630
```

(b) What do you observe about the execution time to sort `S` as a function of n and the buffer (working memory) size? In particular, explain if/why these conform with the formal time complexity of (external) sorting?

To answer this question, you should construct the above table for different *working memory* sizes.[9]

The default working memory size for PostgreSQL is 4MB and the smallest possible working memory is 64kB.

We suggest that you consider the following working memory sizes: 64kb, 4MB, 32MB, and 256MB.

We also suggest that you do not use parallel workers. This can be accomplished by issuing the PostgreSQL interpreter command `set max_parallel_workers = 0`.[10]

Solution: Recall that the time complexity of external sorting in $O(n \log_B n)$, more precisely $2n\lceil\log_B(n)\rceil$, where n denotes the size of the data and B denotes the size of the main memory buffer.

When we look at the timings for the different main memory sizes, we see that they are consistent with this theoretical complexity analysis. The shape of the curves matching the execution times are of the form $O(n \log_B n)$ and for large buffer size these curves relate to each other in the expected ways. However also note that increasing the buffer size (even dramatically) does not speedup sorting all that much. Again that is predicted by the theory results for sorting. Some more detail:

- For a fixed memory buffer size B, the execution times grow slightly faster than linear, i.e. $O(n)$, which is indeed consistent with the growth rate of an $n \log_B n$ function.
- For a fixed relation size n, we see that the execution times decrease with increasing buffer size. Again, this is consistent with the $n \log_B n$ function as B increases. But note that increasing the buffer size does not have a very dramatic effect. This is because for different buffers size B_1 and B_2, with $B_1 < B_2$, we have

$$n \log_{B_1}(n) = (\log_{B_1}(B_2)) \cdot n \log_{B_2}(n),$$

[9] For example, the PostgreSQL interpreter command `set work_mem = '16MB'` sets the working memory to 16MB. Consult `https://www.postgresql.org/docs/14/runtime-config-resource.html` for more information.

[10] Consult `https://www.postgresql.org/docs/14/runtime-config-resource.html` for more information about parallel workers.

and for large B_1, $\log_{B_1}(B_2)$ may only grow slowly as a function of B_2.
We also observe that for larger work memory, the performance of external sorting almost shows a nearly linear behavior $O(n)$. That is also consistent with the the complexity of external sorting: when the base of the logarithm is large, this logarithm is nearly constant for increasing n.

- 256kb

```
 size of relation S | Average execution time for SELECT x FROM S order by 1: 256kB
--------------------+-------------------------------------------------------------
                 10 |                                                        0.013
                100 |                                                        0.039
               1000 |                                                        0.328
              10000 |                                                        4.797
             100000 |                                                       57.464
            1000000 |                                                      694.503
           10000000 |                                                     9172.684
(7 rows)
```

- 4MB

```
 size of relation S | Average execution time for SELECT x FROM S order by 1: 4MB
--------------------+-----------------------------------------------------------
                 10 |                                                      0.014
                100 |                                                      0.039
               1000 |                                                      0.344
              10000 |                                                      3.753
             100000 |                                                     50.102
            1000000 |                                                    534.464
           10000000 |                                                   6693.707
(7 rows)
```

- 16MB

```
 size of relation S | Average execution time for SELECT x FROM S order by 1: 16MB
--------------------+------------------------------------------------------------
                 10 |                                                       0.017
                100 |                                                       0.038
               1000 |                                                       0.330
              10000 |                                                       3.751
             100000 |                                                      42.699
            1000000 |                                                     542.381
           10000000 |                                                    5767.674
(7 rows)
```

- 256MB

```
 size of relation S | Average execution time for SELECT x FROM S order by 1: 256MB
--------------------+-------------------------------------------------------------
                 10 |                                                        0.014
                100 |                                                        0.039
               1000 |                                                        0.401
              10000 |                                                        3.729
             100000 |                                                       42.907
            1000000 |                                                      495.538
           10000000 |                                                     6074.149
(7 rows)
```

(c) Now create a relation `indexedS(x integer)` and create a Btree index on **indexedS** and insert into **indexedS** the sorted relation S.[11]

```
create table indexedS (x integer);
create index on indexedS using btree (x);
insert into indexedS select x from S order by 1;
```

Then run the range query

```
select x from indexedS where x between 1 and n;
```

where n denotes the size of S.

Then construct the following table which contains (a) the average execution time to build the btree index and (2) the average time to run the range query.

size n of relation S	avg execution time to create index **indexedS**	avg execution time for range query (in ms)
10^1		
10^2		
10^3		
10^4		
10^5		
10^6		
10^7		
10^8		

What are your observations about the query plans and execution times to create **indexedS** and the execution times for sorting the differently sized bags **indexedS**? Compare your answer with those for the above sorting problems.

We observe two things:

- The time to create an index on S is expensive. It takes considerably more time to create an index than just sort the relation.
- Since after an B$^+$ tree is created, the leaf level correspond to a sorting in S, retrieving the leaf level to obtain a sort for S is very fast.

Overall, using the strategy to obtain a sorting of S by first indexing and then reading the leaf level of an index is not a good strategy to sort S. However, once the index is created, accessing the sorting of S is clearly beneficial.

4. • Typically, the **makeS** function returns a bag instead of a set. In the problems in this section, you are to conduct experiments to measure the execution times to eliminate duplicates.

   (a) Write a SQL query that uses the DISTINCT clause to eliminate duplicates in S and report you results in a table such as that in Problem 3a.

[11] For information about *indexes* in PostgreSQL consult the manual page https://www.postgresql.org/docs/14/indexes.html.

Solution:

```
select distinct from S;

     QUERY PLAN
---------------------
 HashAggregate
   Group Key: x
   ->  Seq Scan on s
(3 rows)

     size  |     execution time
-----------+------------
       100 |      0.05
      1000 |      0.37
     10000 |      3.86
    100000 |     40.11
   1000000 |    662.72
  10000000 |   9720.39
 100000000 | 102204.93
```

(b) Write a SQL query that uses the `GROUP BY` clause to eliminate duplicates in `S` and report you results in a table such as that in Problem 3a.

Solution:

```
select x from S group by (x);

     QUERY PLAN
---------------------
 HashAggregate
   Group Key: x
   ->  Seq Scan on s

    size   |     execution time
-----------+------------
       100 |      0.07
      1000 |      0.50
     10000 |      5.21
    100000 |     57.67
   1000000 |    637.33
  10000000 |   7894.79
 100000000 |  97348.35
```

(c) Compare and contrast the results you have obtained in problems 4a and 4b. Again, consider using `explain analyze` to look at query plans.

- We notice that the query plans for the `distinct` and `group by` queries to eliminate duplicates are the same, and therefore the execution times are nearly identical. The query plans reveal that S is put in a hash table on key `x` and this guarantees that duplicates are hashed into the same bucket. The complexity is linear $O(|S|)$.
- The query plan for the `UNION` query reveals that before the hash table is created that S is appended with itself. Consequently a larger relation of size $2|S|$ needs to be hashed. The complexity is linear $O(|S|)$.

3 Indexes and Indexing

Indexes on data (1) permit faster lookup on data items and (2) may speed up query processing on such data. Speedups can be substantial. The purpose of the problems in this section are to explore this.

Several other problems in this section are designed to understand the workings of the B^+*-tree* and the *extensible hashing* data structures.

Discussion PostgreSQL permit the creation of a variety of indexes on tables. We will review such `index creation` and examine their impact on data lookup and query processing. For more details, see the PostgreSQL manual:

https://www.postgresql.org/docs/13/indexes.html

Example 5 *The following SQL statements create indexes on columns or combinations of columns of the* `personSkill` *relation.*[12] *Notice that there are*

$$2^{arity(\texttt{personSkill})} - 1 = 2^2 - 1 = 3$$

such possible indexes.

```
create index pid_index on personSkill (pid);                 -- index on pid attribute
create index skill_index on personSkill (skill);             -- index on skill attribute
create index pid_skill_index on personSkill (pid,skill);     -- index (pid, skill)
```

Example 6 *It is possible to declare the type of index:* `btree` *or* `hash`*. When no index type is specified, the default is* `btree`*. If instead of a Btree, a* hash index *is desired, then it is necessary to specify a* `using hash` *qualifier:*

```
create index pid_hash on personSkill using hash (pid);  -- hash index on pid attribute
```

Example 7 *It is possible to create an index on a relation based on a* scalar expression *or a* function *defined over the attributes of that relation. Consider the following (immutable) function which computes the number of skills of a person:*

```
create or replace function numberOfSkills(p integer) returns integer as
$$
    select count(1)::int
    from   personSkill
    where  pid = p;
$$ language SQL immutable;
```

[12]Incidentally, when a primary key is declared when a relation is created, PostgreSQL will create a btree index on this key for the relation.

Then the following is an index defined on the `numberOfSkills` *values of persons:*

```
create index numberOfSkills_index on personSkill (numberOfSkills(pid));
```

Such an index is useful for queries that use this function such as

```
select pid, skill from personSkill where numberOfSkills(pid) > 2;
```

We now turn to the problems in this section.

5. • Consider a relation `Student(sid text, sname text, major, byear)`. A tuple (s, n, m, y) is in `Student` when s is the sid of a student and n, m, and y are that student's name, major, and birth year. Further, consider a relation `Enroll(sid text, cno text, grade text)`. A triple (s, c, g) is in `Enroll` when the student with sid s was enrolled in the course with cname c and obtained a letter grade g in that course.

   We are interested in answering queries of the form

   ```
   select sid, sname, major, byear
   from   Student
   where  sid in (select sid
                  from   Enroll sid
                  where  cno = c [and|or|and not] grade = g);
   ```

   Here `c` denotes a course name and `g` denotes a letter grade.

   Read Section 14.1.7 'Indirection in Secondary Indexes' in your textbook *Database Systems The Complete Book* by Garcia-Molina, Ullman, and Widom. Of particular interest are (a) the concept of *buckets* (Figure 14.7) which are sets of tids and (b) the technique of performing set operations (like intersections) on relevant buckets (Figure 14.8) to answer queries of the form as shown above.

   The goal of this problem is to use object-relational SQL to simulate these concepts. To make things more concrete, consider the following `Student` and `Enroll` relations:

   ```
   Student:
    sid  | sname  |  major  | byear
   ------+--------+---------+-------
    s100 | Eric   | CS      |  1988
    s101 | Nick   | Math    |  1991
    s102 | Chris  | Biology |  1977
    s103 | Dinska | CS      |  1978
    s104 | Zanna  | Math    |  2001
    s105 | Vince  | CS      |  2001
   ```

```
Enroll:
 sid  | cno  | grade
------+------+-------
 s100 | c200 | A
 s100 | c201 | B
 s100 | c202 | A
 s101 | c200 | B
 s101 | c201 | A
 s101 | c202 | A
 s101 | c301 | C
 s101 | c302 | A
 s102 | c200 | B
 s102 | c202 | A
 s102 | c301 | B
 s102 | c302 | A
 s103 | c201 | A
 s104 | c201 | D
```

Now consider associating a tuple id (tid) with each tuple in `Enroll`:

```
 tid | sid  | cno  | grade
-----+------+------+-------
   1 | s100 | c200 | A
   2 | s100 | c201 | B
   3 | s100 | c202 | A
   4 | s101 | c200 | B
   5 | s101 | c201 | A
   6 | s101 | c202 | A
   7 | s101 | c301 | C
   8 | s101 | c302 | A
   9 | s102 | c200 | B
  10 | s102 | c202 | A
  11 | s102 | c301 | B
  12 | s102 | c302 | A
  13 | s103 | c201 | A
  14 | s104 | c201 | D
```

Use object-relational SQL to construct two secondary indexes `indexOnCno` and `indexOnGrade` on the `Enroll` relation. These indexes should be stored in two separate relations which you can conveniently call `indexOnCno` and `indexOnGrade`, respectively. These two object-relational views should **simulate** the situation illustrated in Figure 14.8. In particular, do **not** use the '`create index`' mechanism of SQL to construct these indexes.

Then, using the `indexOnCno` and `indexOnGrade` views and the technique of *intersecting buckets*, write a function `FindStudents(booleanOperation text, cno text, grade text)` that can be used to answer queries of the form as shown above. (Here the booleanOperation is a string which can be 'and', 'or', or 'and not'.)
For example, the query

```
select * from FindStudents('and', 'c202', 'A');
```

should return the same result as that of the query

```
select sid, sname, major, byear
from   Student
where  sid in (select sid
               from   Enroll sid
               where  cno = 'c202' and grade = 'A');
```

Test your solution for the following cases on the `Student` and `Enroll` relation given for this problem:

(a) `select * from FindStudents('and', 'c202', 'A');`

(b) `select * from FindStudents('or', 'c202', 'A');`

(c) `select * from FindStudents('and not', 'c202', 'A');`

6. • Read Section 14.7 'Bitmap Indexes' in your textbook *Database Systems The Complete Book* by Garcia-Molina, Ullman, and Widom. In particular, look at Example 14.39 for an example of a bitmap index for a secondary index.

   Next, revisit Problem 5. There, we considered two secondary indexes `indexOnCno` and `indexOnGrade`. We will now consider the corresponding bitmap indexes `bitmapIndexOnCno` and `bitmapIndexOnGrade`:

```
bitmapIndexOnCno
 cno  |     bit-vector
------+----------------
 c200 | 10010000100000
 c201 | 01001000000011
 c202 | 00100100010000
 c301 | 00000010001000
 c302 | 00000001000100
```

   and

```
bitmapIndexOnGrade
 grade |     bit-vector
-------+----------------
 A     | 10101101010110
 B     | 01010000101000
 C     | 00000010000000
 D     | 00000000000001
```

   Use object-relational SQL to construct two secondary indexes `bitmapIndexOnCno` and `bitmapIndexOnGrade` as two object-relational relations in a manner that simulates the situation just illustrated above.

   Then, using the `bitmapIndexOnCno` and `bitmapIndexOnGrade` relations and the technique of forming the bitmap-`AND`, bitmap-`OR`, and bitmap-`AND NOT` of two bit-vectors, write a function `FindStudents(booleanOperation text, cno text, grade text)` that can be used to answer queries of the form as shown in Problem 5.

   For example, the query

```
select * from FindStudents('and', 'c202', 'A');
```

should return the same result as that of the query

```
select sid, sname, major, byear
from   Student
where  sid in (select sid
               from   Enroll sid
               where  cno = 'c202' and grade = 'A');
```

Test your solution for the following cases on the `Student` and `Enroll` relation given for this problem:

(a) `select * from FindStudents('and', 'c202', 'A');`

(b) `select * from FindStudents('or', 'c202', 'A');`

(c) `select * from FindStudents('and not', 'c202', 'A');`

7. • Consider the following parameters:

block size	=	8192 bytes
block-address size	=	10 bytes
block access time (I/O operation)	=	15 ms (micro seconds)
record size	=	200 bytes
record primary key size	=	8 bytes

Assume that there is a B$^+$-tree, adhering to these parameters, that indexes 10 billion (10^{10}) records on their primary key values.

Provide answers to the following problems and show the intermediate computations leading to your answers.

(a) Specify (in ms) the minimum time to determine if there is a record with key k in the B$^+$-tree. (In this problem, you can not assume that a key value that appears in a non-leaf node has a corresponding record with that key value.)

Solution: We first need to determine the order n of the B$^+$-tree. This can be done by finding the largest n such that

$$10(n+1) + 8n \leq 8192$$

I.e., $n \leq \lfloor \frac{8192-10}{18} \rfloor$. Thus $n = 454$.

Note that there will be 10^{10} keys at the leaf level of the B$^+$ tree. The keys of the 10^{10} records can be stored in $\lceil \frac{10^{10}}{454} \rceil = 22026432$ leaf nodes of the B^+ tree.

Note that the minimum time is determined by the largest possible fanout of the nodes in the B$^+$-tree. This fanout is maximally $n+1 = 454+1 = 455$.

The minimum time will therefore be $(\lceil log_{455}(22026432) \rceil) * 15\,\text{ms} = 3 * 15\,\text{ms} = 45\,\text{ms}$.

(b) Specify (in ms) the maximum time to insert a record with key k in the B$^+$-tree assuming that this record was not already in the data file.

Solution: The maximum time results when we have the minimum branching factor, i.e. $\lceil \frac{454}{2} \rceil + 1 = 227 + 1 = 228$ at non-root nodes.

Note that this time, there will be $\frac{10^{10}}{227}$ leaf nodes of the B$^+$-tree that store the keys of the 10^{10} keys of the records.

Since the minimum branching factor at the root is 2, we must determine the height of a tree that has half of the nodes, i.e., $\lceil \frac{\frac{10^{10}}{227}}{2} \rceil =$ 22026432 nodes.

The height of such a the tree will be $\lceil log_{121}(4166667) \rceil = \lceil 3.114 \cdots \rceil =$ 4.

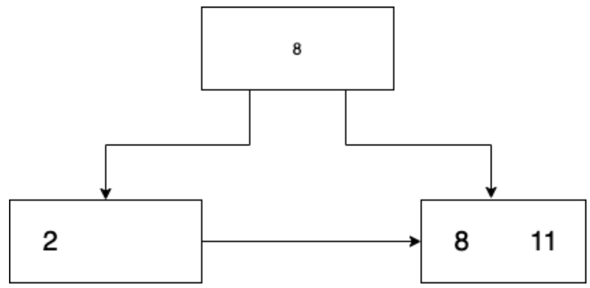
We then also need one more block access to insert the record and another to write that block back to the data file, so the maximum time is $(4 + 2 + 1) * 15\,\text{ms} = 105\,\text{ms}$. The 1 in this sum is present because we also need to read the block containing the root of the B$^+$ tree.

(c) How large must main memory be to hold the first two levels of the B$^+$-tree? How about the first three levels? In what manner does storing these levels in main memory affect the answers in Problem 7a and Problem 7b.

Solution We must store the root and the next level which may have as many as 455 subtrees (i.e., when we have maximum branching at the root). This give a total of $1 + 455$ blocks which is about 3.75 MB of space in main memory to store the 2 top levels of the B$^+$ tree

If we need to store the first 3 levels, we have a need for $1+455+455^2 =$ 207481 blocks which is about 1.7 GB.

8. • Consider the following B$^+$-tree of order 2 that holds records with keys 2, 8, and 11.[13]



(a) Show the contents of your B^+-tree after inserting records with keys 5, 7, 10, 4, 10, 12, and 1 in that order.

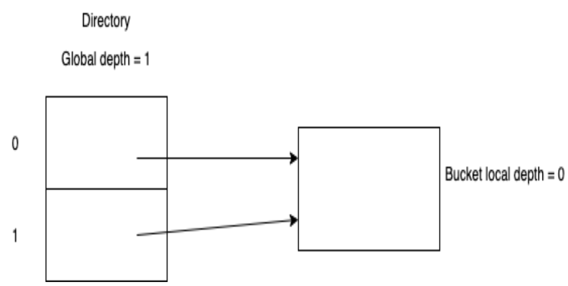
Strategy for splitting leaf nodes: when a leaf node n needs to be split into two nodes n_1 and n_2 (where n_1 will point to n_2), then use the rule that an even number of keys in n is moved into n_1 and an odd number of keys is moved into n_2. So if n becomes conceptually $\boxed{k_1|k_2|k_3}$ then n_1 should be $\boxed{k_1|k_2}$ and n_2 should be $\boxed{k_3|\;\;}$ and $n_1 \rightarrow n_2$.

(b) Starting from your answer in question 8a, show the contents of your B^+-tree after deleting records with keys 12, 2, and 4 in that order.

(c) Starting from your answer in question 8b, show the contents of your B^+-tree after deleting records with keys 10, 7, 1, and 5 in that order.

[13]Recall that (a) an internal node of a B$^+$-tree of order 2 can have either 1 or 2 keys values, and 2 or 3 sub-trees, and (b) a leaf node can have either 1 or 2 key values.

9. • Consider an extensible hashing data structure wherein (1) the initial global depth is set at 1 and (2) all directory pointers point to the same **empty** bucket which has local depth 0. So the hashing structure looks like this: Assume that a bucket can hold at most two records.



(a) Show the state of the hash data structure after each of the following insert sequences:[14]

i. records with keys 9 and 4.
ii. records with keys 1 and 2.
iii. records with keys 8 and 3.
iv. records with keys 6 and 7.

(b) Starting from the answer you obtained for Question 9a, show the state of the hash data structure after each of the following delete sequences:

i. records with keys 4 and 1.
ii. records with keys 9 and 6.
iii. records with keys 7 and 3.

As in the text book, the bit strings are interpreted starting from their left-most bit and continuing to the right subsequent bits.

[14] You should interpret the key values as bit strings of length 4. So for example, key value 7 is represented as the bit string 0111 and key value 2 is represented as the bit string 0010.

The goal in the next problems is to study the behavior of indexes for various different sized instances[15] of the `Person`, `personSkill`, `worksFor`, and `Knows` relations and for various queries:

10. • Create an appropriate index on the `personSkill` relation that speedups the lookup query

```
select pid from personSkill where skill = s;
```

Here s is a skill.

Illustrate and discuss this speedup by finding the execution times for this query for 3 different but sufficiently large sizes of the `personSkill` relation.

You should compare the execution times for running the query without the index versus the the execution times for running the query with indexes.

Solution

We use a hash secondary index on the `skill` attribute and compare scanning and hash index search for different sizes of the `personSkill` relation.

```
create index index_on_skill on personSkill using hash (skill);

We show the result for 2 sizes of personSkill

personSkill      |sequential search| index search
1000000          |   81.405        | 6.17
10000000         |  886.790        | 273.669
```

We observe improvement with hash index search as expected.

11. • Create an appropriate index on the `worksFor` relation that speedups the range query

```
select pid
from   worksFor
where  s1 <= salary and salary <= s2;
```

Here s1 and s2 are two salaries with s1 $\leq$ s2.

Illustrate and discuss this speedup by finding the execution times for this query for 3 different but sufficiently large sizes of the `worksFor` relation.

[15](Three different sizes, small, medium, large suffice for your experiment.

In addition, consider different size ranges [s1,s2]: (a) a small range [s1,s1], (b) an intermediate range [lowest salary,average salary], and (3) the maximum range [smallest salary, highest salary].

You should compare the execution times for running the query without the index versus the the execution times for running the query with indexes.

Solution

We use a B^+-tree and compare scanning and range index search for different size of the worksFor relation and for different ranges.

```
create index index_on_salary on worksFor using btree (salary);

worksFor     |sequential search| index search (range 10000,10000)
1000000      |   80.412        | 11.81
10000000     |  688.381        | 114.81

worksFor     |sequential search| index search (range 10000,100000)
1000000      |  125.915        | 104.367
10000000     |  1218.181       | 1073.840

worksFor     |sequential search| index search (full range)
1000000      |  174.212        | 173.918
10000000     |  1688.78        | 1694.779 ms
```

We observe significant improvement with B^+-tree index search for small ranges, small improvement for medium ranges, and no improvement for full range (this is not unexpected since a full range index search can be supported with a simple scan).

12. • Create indexes on the `personSkill` relation that speedup the multiple conditions query

```
select pid, skill
from   personSkill
where  pid = p and skill = s;
```

Here p is a pid and s is a skill.

Illustrate and discuss this speedup by finding the execution times for this query for 3 different but sufficiently large sizes of the `personSkill` relation.

You should compare the execution times for running the query without the index versus the the execution times for running the query with indexes.

Solution With hash indexes on pid and skill.

```
create index index_on_pid on personSkill using hash (pid);
create index index_on_skill on personSkill using hash (skill);
```

We compare sequential scanning and index search for different sizes of the personSkill relation. We observe substantial improvement.

```
worksFor     |sequential search| index search
1000000      |   78.806        |  0.028
5000000      |  428.741        |  0.050
```

Here is the query plan which uses bitmaps.

```
               QUERY PLAN
-----------------------------------------
 Bitmap Heap Scan on personskill
   Recheck Cond: (pid = 1)
   Filter: (skill = 10)
   ->  Bitmap Index Scan on index_on_pid
         Index Cond: (pid = 1)
```

13. • Create indexes on the appropriate relations that speedup the semi-join [anti semi-join] query

```
select pid, pname
from   Person
where  pid [not] in (select pid from personSkill where skill = s);
```

Here s is a skill.

Illustrate and discuss this speedup by finding the execution times for this query for various, but sufficiently large sizes of the `Person` and `personSkill` relations.

You should compare the execution times for running the query without the index versus the the execution times for running the query with indexes.

Solution

We create a secondary index for `personSkill` on skill. We create a primary index for `Person` on pids.

We compare the performance of running the queries without indexes (an hash join algorithm is used) and with indexes. The performances are all very similar.

```
Person   | personSkill | Sequential (hash join) | With indexes (hash index)
100000   |1000000    | 105.882                | 32.049  (for IN query)
100000   |1000000    |  87.026                | 30.044  (for NOT IN query)
```

We compare the query plans:

- `IN` query
  The algorithm without index is a simple hash join algorithm with hash table creation on $\sigma_{skill=1}(personSkill)$.

```
Hash Join
  Hash Cond: (person.pid = personskill.pid)
  ->  Seq Scan on person
  ->  Hash
        ->  HashAggregate
              Group Key: personskill.pid
                  ->  Seq Scan on personskill
                          Filter: (skill = 1)
```

For the semi join algorithm with index, the query plan is almost the same as that one above, but is uses a bitmap on the personSkill index on skill. (Note however how this query plan ignores the index for person on pid).

```
Hash Join
  Hash Cond: (person.pid = personskill.pid)
  ->  Seq Scan on person
  ->  Hash
        ->  HashAggregate
              Group Key: personskill.pid
              ->  Bitmap Heap Scan on personskill
                    Recheck Cond: (skill = 1)
                    ->  Bitmap Index Scan on index_on_skill
                          Index Cond: (skill = 1)
```

- `NOT IN` query

  For hash-lookup algorithm with hash-table creation on $\sigma_{skill=1}(personSkill)$. This is essentially an anti-join algorithm. It is very similar to the semi-join algorithm.

```
Seq Scan on person
  Filter: (NOT (hashed SubPlan 1))
  SubPlan 1
    ->  Seq Scan on personskill
          Filter: (skill = 1)
```

  For join algorithm with index (note however how this query plan ignores the index for person on pid).

  This query plan is almost the same as that one above.

```
Seq Scan on person
  Filter: (NOT (hashed SubPlan 1))
  SubPlan 1
    ->  Bitmap Heap Scan on personskill
          Recheck Cond: (skill = 1)
          ->  Bitmap Index Scan on index_on_skill
                Index Cond: (skill = 1)
```

Looking at the 4 query plan, we note that they are very similar. The performance is improved by using the index on the personSkill relation.

14. • Create indexes that speedup the path-discovery query

```
select distinct k1.pid1, k3.pid2
from   knows k1, knows k2, knows k3
where  k1.pid2 = k2.pid1 and k2.pid2 = k3.pid1;
```

Illustrate this speedup by finding the execution times for this query for various sizes of the `Knows` relation.

Solution

```
create index index_on_pid1 on knows using btree (pid1);
create index index_on_pid2 on knows using btree (pid2);

Knows has 10000 pairs 1000 pid1, 10 pid2
Knows has 100000 pairs 10000pid2, 10 pid2

Knows   | sequential search|  index search
10000   |     626.538      |  514.615
100000  |    4068.140      | 2720.942
```

We observe marginal improvement.