MSc in Computer Science FHS Computer Science; Mathematics and Computer Science; Computer Science and Philosophy.

Database Systems Implementation

Hilary Term 2020

Submission deadline 12pm, Tuesday 14th April 2020, via weblearn.

There is a total of 100 marks available for this paper, you should attempt all parts of the paper.

NB: You must not discuss this examination paper with anyone.

Your task is to design, implement, and benchmark `MooDB`, an efficient in-memory query engine for computing a batch of group-by aggregate queries on top of a single relation.

A simple approach to this problem is given by the following naïve solution, which we call `NaïveDB`: Given a relation, `NaïveDB` computes the batch of aggregates by processing one tuple at a time from the input relation and accumulating its contribution to each of the aggregates.

`MooDB` improves over `NaïveDB` by using a *trie* data-structure to represent a relation; by assuming a given *attribute order*, each level of the trie data-structure corresponds to one of the attributes. For instance, a trie representation of a relation $R(A, B)$ under the order $[A, B]$ has a union of all distinct A-values in R and for each A-value it has a union of all distinct B-values occurring with that A-value in R. By using this data structure, `MooDB` achieves the following three improvements:

(1) Instead of using a single loop scanning over the entire relation, there are multiple loops scanning over the different levels of the trie structure, each one corresponding to an attribute of the input relation.

(2) Each aggregate computation can be decomposed into computational units, which we henceforth denote as *partial aggregates*, that can be executed at different loops. Instead of processing all aggregates in the most inner loop (corresponding to the last level of the trie), some (partial) aggregates can be moved towards the outer loops. This improvement reduces the number of times each (partial) aggregate is computed.

(3) A careful examining of the decomposed partial aggregates reveals that many of them perform the same computation. The performance can be further improved by sharing these repetitive computations across different aggregates.

Example Database. For instance, assume a database with the relation $R(A, B)$.

Aggregates. You will consider SQL aggregate queries of the following forms:

```
SELECT SUM(EXPR_1), ..., SUM(EXPR_n) FROM R;
SELECT ATTRS, SUM(EXPR_1), ..., SUM(EXPR_n) FROM R GROUP BY ATTRS;
```

where `ATTRS` is a list of group-by attributes and the expressions `EXPR_i` can be (1) the constant `1`, (2) an attribute from the schema of R, or 3) the multiplication of two attributes from the schema of R. Recall that the aggregate `SUM(1)` means `COUNT`.

Example Batch of Aggregates. The following SQL statements represent a batch of aggregates over R:

```
SELECT A, SUM(1), SUM(B)                 FROM R  GROUP BY A;
SELECT B, SUM(1), SUM(A)                 FROM R  GROUP BY B;
SELECT SUM(1), SUM(A), SUM(B), SUM(A*B)  FROM R;
```

Without using a trie data-structure, `NaïveDB` computes all these aggregates using a single scan over the relation R as follows:

```
// init aggs_gb_A & aggs_gb_B & aggs
for(r <- R) {
  aggs_gb_A[r.A][0] += 1
  aggs_gb_A[r.A][1] += r.B
  aggs_gb_B[r.B][0] += 1
  aggs_gb_B[r.B][1] += r.A
  aggs[0] += 1
  aggs[1] += r.A
  aggs[2] += r.B
  aggs[3] += r.A * r.B
}
```

`NaïveDB` creates the array `aggs` for aggregates without group-by, and the maps of arrays `aggs_gb_A` and `aggs_gb_B` for aggregates with group-by A and B, respectively. The maps corresponding to group-by aggregates can be implemented using hash-based, tree-based, or sorted-based data-structures. By using a trie data-structure organized using the attribute order $[A, B]$, `MooDB` computes the batch of aggregates as follows:

```
// init aggs_gb_A & aggs_gb_B & aggs
for(r_A <- R_TRIE) {
  for(r_B <- r_A.LIST) {
    aggs_gb_A[r_A.A][0] += 1
    aggs_gb_A[r_A.A][1] += r_B.B
    aggs_gb_B[r_B.B][0] += 1
    aggs_gb_B[r_B.B][1] += r_A.A
    aggs[0] += 1
    aggs[1] += r_A.A
    aggs[2] += r_B.B
    aggs[3] += r_A.A * r_B.B
  }
}
```

For the group-by aggregates over attribute `A`, `MooDB` does not need to update the associated map (`aggs_gb_A`) for each element in the inner loop. `MooDB` improves the evaluation of this batch of aggregates as follows:

```
// init aggs_gb_A & aggs_gb_B & aggs
for(r_A <- R_TRIE) {
  // init par_aggs_gb_A & par_aggs
  for(r_B <- r_A.LIST) {
    par_aggs_gb_A[0] += 1
    par_aggs_gb_A[1] += r_B.B
    aggs_gb_B[r_B.B][0] += 1
    aggs_gb_B[r_B.B][1] += r_A.A
    par_aggs[0] += 1
    par_aggs[1] += 1
    par_aggs[2] += r_B.B
    par_aggs[3] += r_B.B
  }
  aggs_gb_A[r_A.A][0] += par_aggs_gb_A[0]
  aggs_gb_A[r_A.A][1] += par_aggs_gb_A[1]
  aggs[0] += par_aggs[0]
  aggs[1] += r_A.A * par_aggs[1]
  aggs[2] += par_aggs[2]
  aggs[3] += r_A.A * par_aggs[3]
}
```

`MooDB` first computes the partial aggregates that are dependent on the attribute `B` in the inner level of the trie data-structure (`r_A.LIST`). Then, it uses these partial aggregates in the outer loop in order to compute the batch of aggregates. Furthermore, by careful examining of the partial aggregates, we observe that several of them compute the same values (e.g., `par_aggs_gb_A[0]`, `par_aggs[0]`, and `par_aggs[1]`). Thus, `MooDB` can further improve the performance by sharing the computation across the aggregates:

```
// init aggs_gb_A & aggs_gb_B & aggs
for(r_A <- R_TRIE) {
  // init par_aggs_gb_A
  for(r_B <- r_A.LIST) {
    par_aggs_gb_A[0] += 1
    par_aggs_gb_A[1] += r_B.B
    aggs_gb_B[r_B.B][0] += 1
    aggs_gb_B[r_B.B][1] += r_A.A
  }
  aggs_gb_A[r_A.A][0] += par_aggs_gb_A[0]
  aggs_gb_A[r_A.A][1] += par_aggs_gb_A[1]
  aggs[0] += par_aggs_gb_A[0]
  aggs[1] += r_A.A * par_aggs_gb_A[0]
  aggs[2] += par_aggs_gb_A[1]
  aggs[3] += r_A.A * par_aggs_gb_A[1]
}
```

Note on marking: These questions have parts of increasing complexity. Part (a) of Question 1 asks for the design of `MooDB` using the first improvement mentioned above. Part (a) of Question 2 asks for the implementation of `MooDB` with the first improvement. Similarly, Parts (b) and (c) of these two questions refer to more complex second and third improvements, respectively. It is advisable to address this exam paper layer by layer (e.g., first Part (a) of the first two questions and Part (a), (b), and (c) of question 3, then Part (b) of the first two questions and Part (d) of question 3, and so on) so as to maximize the number of marks you can get. You can obtain: 50 marks (pass) by solving Part (a) of the first two questions and proposing an extension to your design to support the second improvement in Question 1 Part (b) as well as performing the first three benchmarking tasks; and 70 marks (distinction) by attempting an implementation of the second improvement in Question 2 Part (b) and benchmarking it. Finally, note that your design and implementation must work for databases of arbitrary schema and content. Otherwise, a significant portion of marks will not be awarded.

You may assume the following simplifications: The tables are sets of distinct tuples, i.e., the tables do not accommodate duplicates. Furthermore, all values in the database are real numbers (double). Furthermore, you can assume the attribute order for the given relation is the order in which the attributes appear in the schema. For example, for relation $S(A, B, C)$, you can assume $[A, B, C]$ as the attribute order.

Question 1

This question has three parts and is worth 30 = 10+10+10 marks.

Your first task is to **design** `MooDB` that provides efficient computation of a batch of aggregates on top of a single relation. In case your design is inspired by existing research literature, you must appropriately cite the source and explain in detail the distinct novel aspects of your approach.

(a) (i) Give the pseudocode of an algorithm that builds a trie data-structure from a relation of any schema, and computes a batch of aggregates using the trie. The input to the algorithm is given by: (1) the schema of the input relation (and its attribute order) (2) a batch of aggregate queries, some of them with a group-by clause, as given by the SQL expressions mentioned above. The output of the algorithm is the answer to these queries over the input relation. The algorithm needs to build a trie data-structure out of the input relation and compute the aggregates while scanning over the different levels of the trie data-structure. The algorithm can use additional memory for storing the trie data-structure.

(ii) Succinctly describe your algorithm and explain the data structures it uses. Explain if your algorithm is superior to `NaïveDB`.

(iii) Assume the following batch of aggregates over relation $S(A, B, C)$:

```
SELECT A, SUM(1), SUM(B), SUM(C)      FROM S  GROUP BY A;
SELECT B, SUM(1), SUM(A), SUM(C)      FROM S  GROUP BY B;
SELECT C, SUM(1), SUM(A), SUM(B)      FROM S  GROUP BY C;
SELECT SUM(1), SUM(A), SUM(B), SUM(C)
       SUM(A*B), SUM(A*C), SUM(B*C)
FROM   S;
```

By showing some pseudocode similar to what was shown above, illustrate how the algorithm processes this batch of aggregates.

(10 marks)

(b) (i) Extend your previous algorithm to decompose aggregates into partial aggregates and push them towards the most possible outer level loop. The algorithm can use additional memory for storing the trie data-structure and partial aggregates.

(ii) Succinctly describe your algorithm and explain the data structures it uses. Explain why your algorithm is superior to `NaïveDB` and Part (a).

(iii) Show how the algorithm works for the batch of aggregates over $S(A, B, C)$ given in Part (a).

(10 marks)

(c) (i) Extend your previous algorithm to maximize the sharing of the computation across the aggregates in Part (b). The constraints on memory use from Part (b) hold here as well, i.e., the algorithm can use additional memory for storing the trie data-structure and partial aggregates.

(ii) Succinctly describe your algorithm and explain the data structures it uses. Explain why your algorithm is superior to `NaïveDB` and Part (b).

(iii) Show how the algorithm works for the batch of aggregates over $S(A, B, C)$ given in Part (a).

(10 marks)

Question 2

This question has three parts and is worth 45 = 15+15+15 marks.

Your second task is to **implement** `MooDB`. Your implementation must follow your design proposed for Question 1.

For this implementation task, you may use a programming language of your choice as long as the code is compilable and runnable on the lab machines. If needed, you may also use open-source implementations of standard data structures and algorithms presented in the course for sorting and indexing. You must state explicitly which simplifications you assumed in your prototype.

Your implementation must be accompanied by documentation on how to compile, run, and benchmark your `MooDB`. The code must also be sufficiently well-commented. Marks will not be awarded if your code is not compilable, runnable, or commented or if the documentation is missing.

(a) Implement your algorithm from Question 1 Part (a). (15 marks)

(b) Implement your algorithm from Question 1 Part (b). (15 marks)

(c) Implement your algorithm from Question 1 Part (c). (15 marks)

Question 3

This question has five parts and is worth 25 = 5+5+5+5+5 marks.

Your third task is to **benchmark** `MooDB` against `NaïveDB`. `NaïveDB` may be implemented by computing the aggregates using a scan over the input relation (without constructing a trie).

You will consider instances of a relation $R(A, B, C, D, E)$ with the attribute order $[A, B, C, D, E]$. You can find on the course web page, 20 instances of the dataset, one per each scale factor from 1 to 20. You will consider the following batch of aggregates:

```
SELECT A, SUM(1), SUM(B), SUM(C), SUM(D), SUM(E)  FROM R  GROUP BY A;
SELECT B, SUM(1), SUM(A), SUM(C), SUM(D), SUM(E)  FROM R  GROUP BY B;
SELECT C, SUM(1), SUM(A), SUM(B), SUM(D), SUM(E)  FROM R  GROUP BY C;
SELECT D, SUM(1), SUM(A), SUM(B), SUM(C), SUM(E)  FROM R  GROUP BY D;
SELECT E, SUM(1), SUM(A), SUM(B), SUM(C), SUM(D)  FROM R  GROUP BY E;
SELECT SUM(1), SUM(A), SUM(B), SUM(C), SUM(D), SUM(E),
       SUM(A*B), SUM(A*C), SUM(A*D), SUM(A*E),
       SUM(B*C), SUM(B*D), SUM(B*E),
       SUM(C*D), SUM(C*E),
       SUM(D*E)
FROM   R;
```

The following benchmarking tasks ask you to draw plots that show the time performance of MooDB and NaïveDB (this is on the y-axis) as the scale factor of the dataset is varied (this is on the x-axis). You should run each experiment five times and plot the mean wall-clock time in seconds of the last four times (optionally with error bars). You may decide on a reasonable timeout per experimental setting (say, one hour) in case one of the two systems takes too long.

You must document the hardware specification of the machine you use (CPU, memory), how to run your experiments, and provide a script to run them and generate the plots or the values to be plotted. You must explain each plot included in your answer – this means explaining why MooDB and NaïveDB behave the way they do in your experiments. Marks will not be awarded if the documentation, the script for running the experiments, or the explanation of experimental findings is not satisfactory.

(a) Plot the (wall-clock) time needed by NaïveDB to compute the batch of aggregates on relation R as the scale factor varies from 1 to 20. Also, plot the average time for computing a SUM aggregate for each scale factor. Comment on the relative performance of computing all aggregates together vs computing them independently. (5 marks)

(b) Plot the (wall-clock) time needed by MooDB to construct the trie data-structure for relation R as the scale factor varies from 1 to 20. Also, plot the time it takes to load relation R into the main memory. Comment on the relative performance of constructing the trie data-structure vs loading the relation into the main memory. (5 marks)

(c) Plot the (wall-clock) time needed by MooDB with only the first improvement enabled to compute the batch of aggregates on relation R (excluding the time needed to build the trie) as the scale factor varies from 1 to 20. Also, plot the average time for computing a SUM aggregate for each scale factor. Comment on the relative performance of computing all aggregates together vs computing them independently, as well as the relative performance of MooDB with the first improvement vs NaïveDB. (5 marks)

(d) Plot the (wall-clock) time needed by MooDB with the second improvement enabled to compute the batch of aggregates on relation R (excluding the time needed to build the trie) as the scale factor varies from 1 to 20. Also, plot the average time for computing a SUM aggregate for each scale factor. Comment on the relative performance of computing all aggregates together vs computing them independently, as well as the relative performance of MooDB with the second improvement vs MooDB with the first improvement. (5 marks)

(e) Plot the (wall-clock) time needed by MooDB with the third improvement enabled to compute the batch of aggregates on relation R (excluding the time needed to build the trie) as the scale factor varies from 1 to 20. Also, plot the average time for computing a SUM aggregate for each scale factor. Comment on the relative performance of computing all aggregates together vs computing them independently. (5 marks)