Data Bases II
Query optimization

Michele Beretta

michele.beretta@unibg.it

Exam from december 2017

You have the following schema

```
Measure(PlaceId, Time, Temperature)
Place(Id, Name, Region, Longitude, Latitude)
```

And the following structures

Table	Tuples	Blocks	Primary index	Secondary indeces
Measure	10G	10M	Hash(PlaceId, Time)	B+(Temperature, PlaceId, Time) B+(PlaceId, Temperature, Time)
Place	10K	1K	Sequential	B+(Region, Name)

The trees on `Measure` have 10M leaves at distance 3 from the root, while the one on `Place` has 100 leaves at distance 1.

Consider the following query

```
SELECT PlaceId, MAX(Temperature), MIN(Temperature)
FROM Measure
GROUP BY PlaceId
```

Describe the best execution strategy, with an estimate of number of disk accesses.

A first approach may be to read all 10M blocks of `Measure`, and then work in memory. The cost is then simply

$$C_1 = 10M$$

Another (even more naïve) approach would be to find the minimum and maximum starting from `Place`: for every place, read all its measures and get the two values

$$C_2 = \underbrace{1K}_{\text{Place's blocks}} + \underbrace{10K}_{\text{Place's tuples}} \cdot \underbrace{10M}_{\text{Measure's blocks}}$$

This is clearly worse. We have to use indeces.

With what we have, we can use the second B+ tree on `Measure`, ordered by `PlaceId`, `Temperature`, and `Time`, and find the minimum and maximum values of `Temperature` for every `PlaceId`.

We can access the data for a given `PlaceId` with cost 4: 3 to navigate the tree, 1 to access the disk.

Moreover, since the tree is ordered by `PlaceId` and then by `Temperature`, we can directly find the minimum and maximum values for every place by accessing this tree twice.

If we wanted to extract even more performance, we know that given a place (say `A`) its maximum temperature value is adjacent to the minimum of the next place in the order (say `B`). This is true even for different leaves, since the tree is a B+ tree and its leaves are in a linked list. We can then "share" the tree access for the maximum value of `A` and the minimum value of `B`. We can then have only 4 accesses per place instead of 8.

The cost of this last strategy would be

$$C = \underbrace{4}_{\text{B+ tree usage}} \cdot \underbrace{10K}_{\text{Place's tuples}} = 40K$$

Exam from december 2016

```
Student(StudId, Surname, Name, Birthday, Address)
Exam(StudId, CourseId, Date, Grade)
Course(CourseId, Title, Professor)
```

We have the following

Table	Tuples	Blocks	Primary index	Secondary indeces
Student	10K	1K	Sequential	B+(StudId), fanout 100, leaves are 1/10 of blocks
Exam	100K	10K	Sequential	B+(StudId, CourseId), fanout 100, leaves are 1/10 of blocks
Course	1K	100	Sequential	B+(CourseId), fanout 100, leaves are 1/10 of blocks

Consider the following query

```
SELECT *
FROM Student S
  JOIN Exam E ON S.StudId = E.StudId
  JOIN Course C ON E.CourseId = C.CourseId
WHERE Surname = "Rossi"
  AND Title = "DB2"
```

Knowing that there are 100 students with surname `"Rossi"`, and a total of 1000 exams of DB2, describe the best execution strategy in these three scenarios:

1. Table `Exam` only has an index on `(StudId, CourseId)`
2. In addition to point 1, table `Exam` has two indeces for the different orderings of its primary key
3. In addition to points 1 and 2, we have and index on `Student` for the attributes `(Surname, Name, Birthday)`

All indeces are B+ trees unless explicitely stated.

1. Table `Exam` only has an index on `(StudId, CourseId)`

Without any hash and being the tables all *entry-sequenced*, we have to use secondary indeces. We can find the best *nested loop*.

We can do a scan of `Student`, obtain all the Rossi students by filtering in memory, and then use indeces to access `Exam` first and then `Course`.

However, we have to give an estimate of *how many exams per students* there are (on average at least). A sensible assumption would be a uniform distribution, i.e., 10 exams per student.

Hence, this is our total cost

$$C_1 = \underbrace{1K}_{\text{Student scan}} + \underbrace{100}_{\text{'Rossi' students}} \cdot \left(\underbrace{3}_{\text{B+tree access to Exam}} + \underbrace{10}_{\text{Exams per student}} \cdot \underbrace{2}_{\text{B+tree access to Course}} \right) = 3.3K$$

Since the leaves are 1/10 of blocks (i.e., 1000 for `Exam`), accessing the tree costs $\lceil \log_{\text{fanout}} \text{leaves} \rceil = \lceil \log_{100} 1000 \rceil = 2$, to which we add 1 to access the disk.

2. In addition to point 1, table `Exam` has two indeces for the different orderings of its primary key

The previous strategy is still valid. Let's see if we can do better.

If we exploit the second index on `Exam`, we could first find all DB2 exams and then filter the students.

Hence

$$C_2 = \underbrace{100}_{\text{Course's scan}} + \underbrace{3}_{\text{B+tree to Exam}} + \underbrace{1000}_{\text{Number of DB2 exams}} \cdot \underbrace{2}_{\text{B+tree to Student}} \approx 2.1K$$

The B+tree access to `Student` is obtained through the already shown formula.

3. In addition to points 1 and 2, we have and index on `Student` for the attributes `(Surname, Name, Birthday)`

Both previous strategies are still valid.

Let's see what happens if we use this new index on `Student`. We can obtain all `"Rossi"` students directly through this index with a single access to the B+tree, and then get all their exams and courses as we did in C_1.

Hence

$$C_3 = \underbrace{2}_{\text{B+tree access to Student}} + \underbrace{100}_{\text{'Rossi' students}} \cdot \left(\underbrace{3}_{\text{B+tree access to Exam}} + \underbrace{10}_{\text{Examps per student}} \cdot \underbrace{2}_{\text{B+tree to Course}} \right) \approx 2.3K$$

The second strategy is still better (by about 200 disk accesses).