Databases

http://www.dcs.fmph.uniba.sk/~plachetk

/TEACHING/DB2

http://www.dcs.fmph.uniba.sk/~sturc/databazy/rldb

Tomáš Plachetka, Ján Šturc

Faculty of mathematics, physics and informatics,
Comenius University, Bratislava

Summer 2024–2025

Database tuning

Engineering aspects of query optimisation

- Estimation of the output size of a query
- Tuning of schema: indexes, denormalisation, partitioning

Estimation of the output size

The evaluation of a query plan (e.g. choosing the order of joins) is measured in the total number of output tuples. However**, how to compute the number of output tuples *before* computing them?**

For example, the number of tuples produced by a join R S lies in general between 0 (an empty relation) and |R||S| (a cartesian product). **It depends on the data** in R and S (and the join condition)

The query optimiser collects some statistics on the relations in a special table called the **catalogue**. This allows for a better estimation than, say, |R||S| / 2 for the output size of R S

Estimation of the output size

Statistics for a relation R:

B(R): the number of blocks

T(R): the number of tuples

V(R, A): the number of distinct values of the attribute A

MAX(R, A): the maximum value of A

MIN(R, A): the minimum value of A

We will assume SQL queries of form

select AttrList

from R_1, ..., R_N

where Cond

The **reduction factor** for a query Q (i.e. the output relation) is defined as rf(Q) = B(Q) / (B(R_1) * ... * B(R_N))

Estimation of the output size

Idea: **The reduction factor for a query Q will be estimated by induction on the structure of Q** (without actually computing Q):
rf(Q) = rf(AttrList) * rf(Cond) /* assumption of independence */

rf(AttrList) = #attr(AttrList) / $\sum_i$ #attr(R_i)
/* assumption that all attributes contribute equally to the output size */

rf(Cond1 and Cond2) = rf(Cond1) * rf(Cond2)
/* assumption of independence */

rf(Cond1 or Cond2) = min(1, rf(Cond1) + rf(Cond2))
/* assumption of independence */

Estimation of the output size

$rf(R_i.A=const) = 1 / V(R_i.A)$
/* assumption of uniform distribution */

$rf(R_i.A>const) = (MAX(R_i.A) – const) / (MAX(R_i.A) – MIN(R_i.A))$
/* assumption of uniform distribution */

$rf(R_i.A=R_j.B) = 1 / (max(V(R_i.A), V(R_j.B)))$
/* assumption of uniform distribution */

etc.

E.g. the number of the output tuples for a selection $\sigma_{R.A=const}(R)$ can then be estimated as T(R) * rf(R.A=const)

Schema tuning: a sample database

A sample university database (Kifer et al.):

students(StudentId, SName, SAddr, SStatus)

professors(ProfId, PName, DeptId)

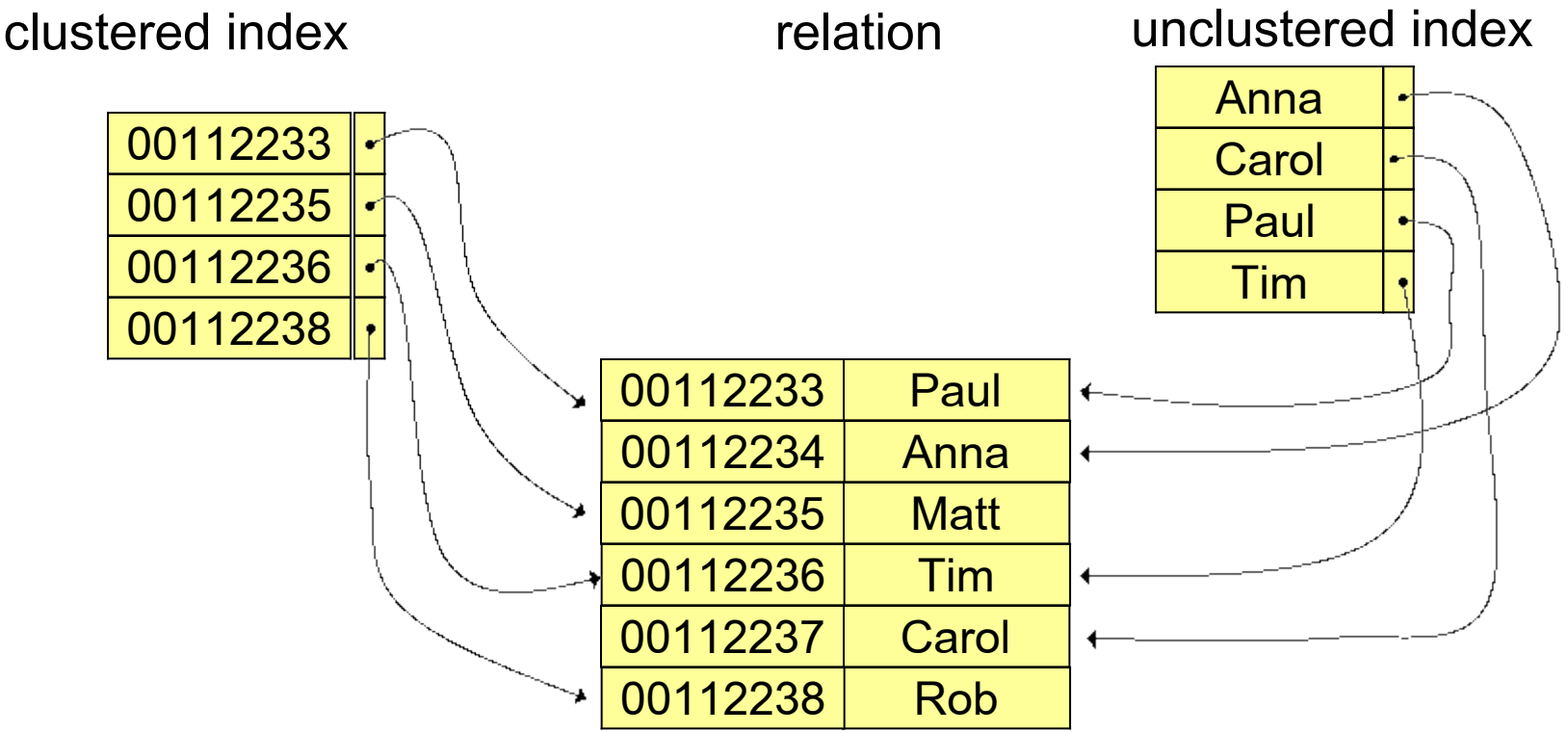
courses(CourseId, DeptId, CName, CDescr)

transcripts(StudentId, CourseId, Semester, Grade)

teaching(ProfId, CourseId, Semester)

Schema tuning: indexes

A clustered index (usually created on the primary key) reflects the physical ordering of tuples in a relation. As there is only one physical ordering of tuples, **there is at most one clustered index** There may be **arbitrarily many non-clustered indexes** (hash tables or search trees, e.g. B^+-trees) attached to a relation



Schema tuning: indexes

professors(ProfId, PName, DeptId)

Assume that ProfId is the primary key. Hence, the system probably automatically creates a clustered index on ProfId (unless the schema states otherwise)

select p.DeptId
from professors p
where p.PName = :name

If names of professor tend to differ, create an unclustered index on PName

However, what **if there are many professors with the same name** (e.g. 'Johanson' in Island)? If this is a frequent query, **create a *clustered index* on PName, not on the primary key (and let the index on the primary key be unclustered, which guarantees the uniqueness of ProfId anyway)**

Schema tuning: indexes

transcripts(StudentId, CourseId, Semester, Grade)

select t.StudentId, t.CourseId
from transcripts t
where t.Grade = :grade

Do not create an unclustered index on Grade, because grades are from a **small domain** (A-Fx). For a concrete :grade, the output will be large. An index scan may therefore be slower than a simple sequential scan

Do not create a clustered index on Grade, save it for other queries. **The best is not to create any index at all**

Do not create indexes for queries which access (e.g. output) more than 20% of tuples

Schema tuning: indexes

transcripts(StudentId, CourseId, Semester, Grade)

Presumably, the queries to this table frequently use conditions on StudentId and CourseId. Less frequent queries use a condition on Semester. It seems therefore logical to create a clustered index on [StudentId, CourseId]. However, it is not a good idea, because there is seldomly more than 1 record for a given [StudentId, CourseId] (only when a student repeats a course)

It is appropriate to create an **unclustered hash index on [StudentId, CourseId]** (a hash index has a smaller overhead than a B^+-tree and we are discussing non-range queries). In addition, create a **clustered B^+-tree index on Semester** (for which range queries are frequent)

Use clustering for grouping tuples which are likely to appear in the output of queries

Schema tuning: indexes

teaching(ProfId, CourseId, Semester)

Suppose that we have already created a clustered B^+-tree index on Semester in order to tune a query. We are tuning another query which needs a *clustered* index on [ProfId, CourseId]. What now?

In this case (a small number of attributes), we are lucky, beause we can create an **unclustered B^+-tree index on [ProfId, CourseId, Semester]**. The computation of output tuples will then not access the table teaching at all, because the values will be fetched directly from the index! (This is called an index cover.)

Schema tuning: indexes

professors(ProfId, PName, DeptId)
courses(CourseId, DeptId, CName, CDescr)
teaching(ProfId, CourseId, Semester)

```
select
from professors p, courses c
where p.DeptId = ‘cs’ and c.DeptId = ‘math’ and c.CourseId in
(
    select t.CourseId
    from teaching t
    where t.Semester = ‘s2015’ and t.ProfId = p.ProfId
)
```

Nested queries are (usually) optimised separately. Hence, a clustered index on CourseId in teaching does not help at all, because CourseId does not appear in the where clause of the subquery. It is better to **rewrite the query without nesting**

Schema tuning: indexes

professors(ProfId, PName, DeptId)
courses(CourseId, DeptId, CName, CDescr)
teaching(ProfId, CourseId, Semester)

select
from professors p, courses c, teaching t
where p.DeptId = ‘cs’ and c.DeptId = ‘math’ and c.CourseId = t.CourseId and t.Semester = ‘s2015’ and p.ProfId = t.ProfId

A good evaluation plan is then (perhaps extended with some preprocessing)
$\sigma_{Dept='cs'}$(professors) $\sigma_{Semester='s2015'}$(teaching $\sigma_{DeptId='math'}$courses)
which benefits from a **clustered index on CourseId in teaching**. (The table **professors is covered by indexes**.)

Schema tuning: indexes

transcripts(StudentId, CourseId, Semester, Grade)

```
select t.Semester, count(*) as Cnt
from trancripts t
where t.Grade <= :grade
group by t.Semester
```

An intuition tells us to create a clustered B^+-tree index on Grade, because the condition expresses a range of values. However, this condition is not very selective...

It is better to create a **clustered index on Semester** (B^+-tree or hash) which is needed for creating the groups. Each group is then sequentially scanned, counting the number of tuples

Schema tuning: indexes

transcripts(StudentId, CourseId, Semester, Grade)
students(StudentId, SName, SAddr, SStatus)

```
select t.Semester, count(*) as Cnt
from students s, transcripts t
where s.StudentId = t.StudentId and t.CourseId = 'cs'
group by t.Semester
```

With no indexes, the plan may be a nested-loop-join or a sort-merge-join. Such plans are inefficient, because the **output is probably small and intermediate results are probably large**.
It helps to create an index on CourseId (which is the primary key anyway) in the table transcripts

Schema tuning: indexes

transcripts(StudentId, CourseId, Semester, Grade)
teaching(ProfId, CourseId, Semester)

```
select t.ProfId, r.StudentId
from teaching t, transcripts r
where t.Semester = r.Semester and t.CourseId = r.CourseId
```

The output is probably much larger than any of the tables involved. An appropriate plan is a sort-merge-join for the computation of the join. We can help the optimiser with the choice of this plan by the creation of a clustered index [Semester, CourseId] for the table transcript (which saves a large part of the sorting)

Schema tuning: indexes

students(StudentId, SName, SAddr, SStatus)
professors(ProfId, PName, DeptId)
courses(CourseId, DeptId, CName, CDescr)
transcripts(StudentId, CourseId, Semester, Grade)
teaching(ProfId, CourseId, Semester)

ProfId and CourseId are **foreign keys** in teaching (a non-existing professor does not teach, a non-existing course is not taught). E.g. each deletion of e.g. a professor from professors fires an integrity check in the table teaching (deleting all the tuples with that ProfId)
It is therefore advisable to **have an index on foreign keys**

Schema tuning: form of queries

The choice of built-in operators is important

courses(CourseId, DeptId, CName, CDescr, **Hours**)

```
select c.CName
from courses c
where c.Hours <> 2
```

Avoid <> when possible, it usually leads to a sequential scan. If '2' is a very frequent value for hours, then this is perhaps better (a union of all other values):

```
select c.CName
from courses c
where c.Hours = '1' or c.Hours = '3' or c.Hours = '4' or c.Hours = '5'
```

Schema tuning: form of queries

professors(ProfId, PName, DeptId, HatSize)

```
select p.DeptId, max(p.HatSize)
from professors p
group by p.DeptId
having p.DeptID in ('cs', 'math')
```

It is better to move the selection to the where clause, it will be applied earlier:

```
select p.DeptId, max(p.HatSize)
from professors p
where p.DeptID = 'cs' or p.DeptID = 'math'
group by p.DeptId
```

Although these queries are equivalent, the optimiser may apply the selection after aggregation in the plan for the first query, which unnecessarily increases the size of the intermediate results

Schema tuning: denormalisation

students(StudentId, SName, SAddr, SStatus)
professors(ProfId, PName, DeptId)
courses(CourseId, DeptId, CName, CDescr)
transcripts(StudentId, CourseId, Semester, Grade)
teaching(ProfId, CourseId, Semester)

Denormalisation increases the performance of frequent queries by the violation of a normal form of some table or tables. It often has the form of adding new (redundant) attribute to a table

Schema tuning: denormalisation

students(StudentId, SName, SAddr, SStatus, **AvgGrade**)
transcripts(StudentId, CourseId, Semester, Grade)

For example, if a query concerning the overall averages over students' grades is frequent, **the table student can be extended with an attribute AvgGrade**

A common problem with such a denormalisation are updates to the database. In this case, **when the table transcript changes, the value of AvgGrade in students must be updated as well. This is what triggers are used for**

A general advice is: do not use denormalisation (unless you know very well what you are doing)

Schema tuning: denormalisation (horizontal partitioning)

transcripts(StudentId, CourseId, Semester, Grade)

select t.StudentId, t.CourseId
from transcripts t
where t.Grade = :grade

If this is a very frequent query, the table transcript could be explicitly grouped:
transcriptsA(StudentId, CourseId, Semester)
transcriptsB(StudentId, CourseId, Semester)
...
transcriptsF(StudentId, CourseId, Semester)
Then the query becomes e.g.
select t.StudentId, t.CourseId
from transcriptsA t

Again, **a general advice is: do not do this**

Schema tuning: denormalisation (vertical partitioning)

students(StudentId, SName, SAddr, SStatus, Photo, Phone, ...)

If most attributes are usually unused in queries, the table can be split into two tables, e.g.

students1(StudentId, SName)

students2(StudentId, SStatus, Photo, Phone, ...)

Some systems do this internally (it saves the transfer costs for retrieving tuples from the disk and writing tuples to the disk)

Summary

This is (an incomplete) collection of tuning techniques of an engineering character. A general advice:

- Do not fiddle with the schema denormalisation prematurely
- Choose the clustered indexes with care (the change of a clustered index requires resorting the relation), reflect the structure of intermediate results of frequent queries
- Think twice before you create an additional unclustered index. Indexes can significantly speed up queries, but slow down updates (as all indexes attached to relations must also be updated)
- When importing a large amount of data into the database, consider to remove all indexes; and recreate the indexes back again after the database has been populated
- Do not use all the features of the SQL language. Write queries in a "canonical" form which does not unnecessarily limit the optimiser
- When a machine-generated plan for an SQL query differs from your expectation, manually rewrite the query directly in the relational algebra. If your SQL system does not allow you to do this, learn the details of its optimiser!