Query languages 1 (NDBI001)
Query Optimization

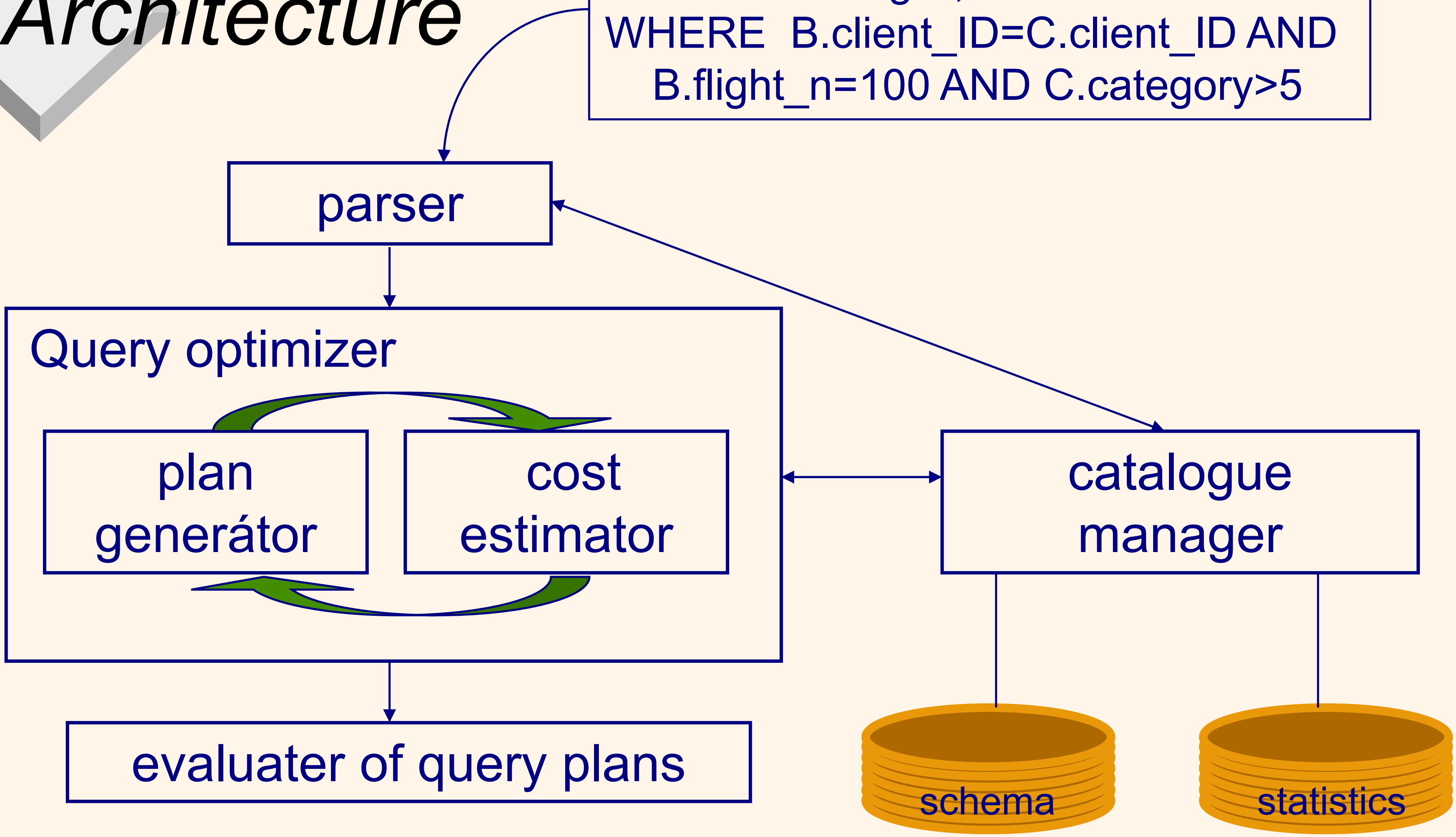
Jaroslav Pokorný
MFF UK, Praha
pokorny@ksi.mff.cuni.cz

DBMS context

- A key module of DBMS
- Goal: to make optimization independent on strategy query expression
  - Counterexamples: navigational languages, SQL interpreter
- Parallel to evaluation of arithmetic expressions
  - Here: time complexity of operations of A_R using I/O operations
  - Crucial factors: size of relations, size of active domains, indexes, hashing, bitmaps etc.

Architecture

```
SELECT  C.name
FROM  Booking B, Clients C
WHERE  B.client_ID=C.client_ID AND
   B.flight_n=100 AND C.category>5
```



Optimizer

- Phases of query processing
  - Transfer into the internal form
    - SQL $\to$ A_R
    - linear expression $\to$ tree

    Remark: calculi $\leftarrow \to$ A_R in polynomial time depending on the expression length
  - conversion into canonic form
  - optimization
  - evaluation plan
  - code generating

Overview of the problem

- *Evaluation plan*: query tree + algorithm for each operation.
- Two main ideas:
  - which plans are considered for given query?
  - how to estimate the plan cost ?
- From the plans considered it is chosen the one with the least cost.

Ex..: System R

- using statistic data for cost estimation,
- using equivalent algebraic expressions,
- restriction to *left-deep* plans.

Example of a schema

Clients (<u>*client_ID*: int</u>, *name*: string, *category*: int, *age*: real)
Booking(<u>*client_ID*: int, *flight_n*: int,</u> *date*: date, *remark*: string)

Semantics: Clients book their flights until a given date.

Parameters: B = 4 KByte

- Booking:

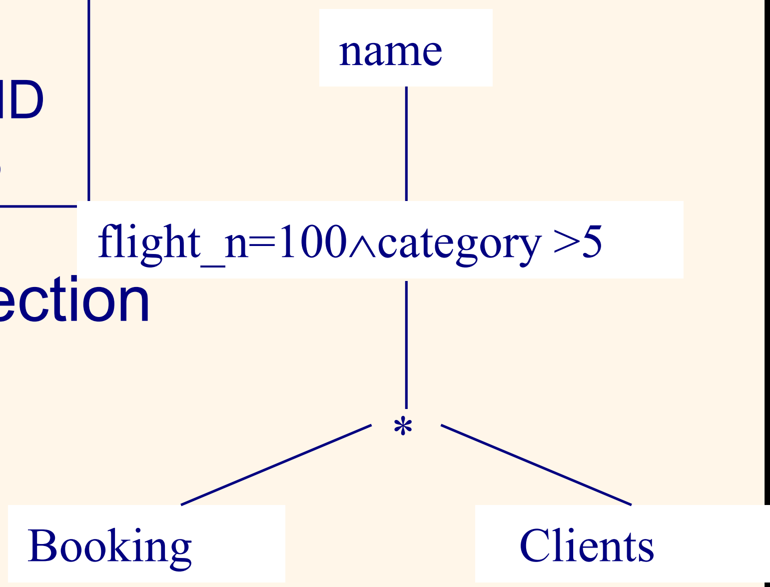
  R = 40 Byte, b = 100, p_B = 1000 pages.

- Clients:

  R = 50 Bytes, b = 80, p_C = 500 pages.

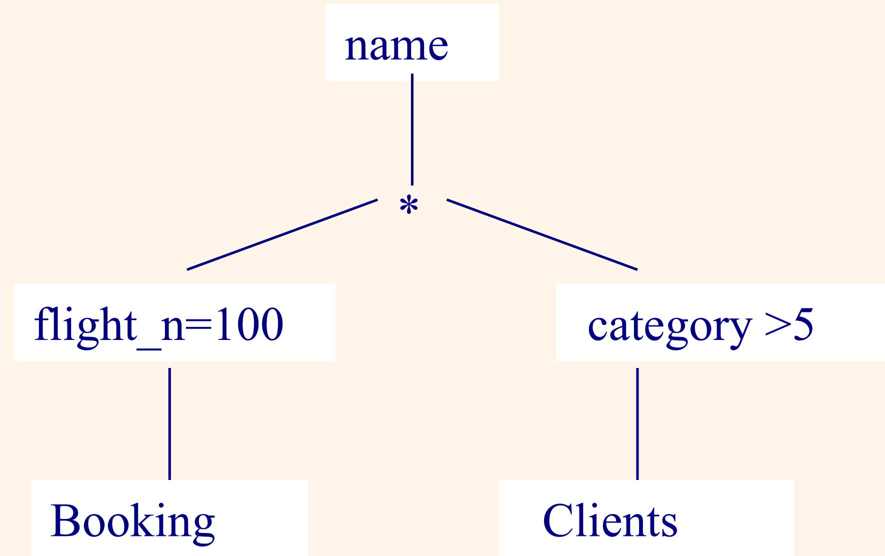
Alternative 1

```
SELECT  C.name
FROM  Booking B, Clients C
WHERE  B.client_ID=C.client_ID AND
   B.flight_n=100 AND C.category>5
```



- Plan: join by nested-loops, selection+projection when the result is generated
- Cost: 500+500*1000 I/Os
- Apparently, the worst plan!
- Use possibilities: selections should be evaluated earlier, available indexes, etc.
- Optimization goal: To find the most effective plans, which lead to the same result (answer).

Alternative 2 (without indexes)



- The main difference: selections first.
- Assumption: M=5 (5 buffers. Calculation of the plan cost:
  - Scanning Booking (1000) + writes into T1 (10 pages, if we have 100 flights and uniform distribution).
  - Scanning Clients (500) + writes into T2 (250 pages, if we have 10 categories, uniform distribution).
  - Sort(T1) (2*2*10), Sort(T2) (2*4*250), Merge(T1,T2) (10+250)
  - Sum: 1000+10+500+250+40+2000+260= 4060 I/O operations.

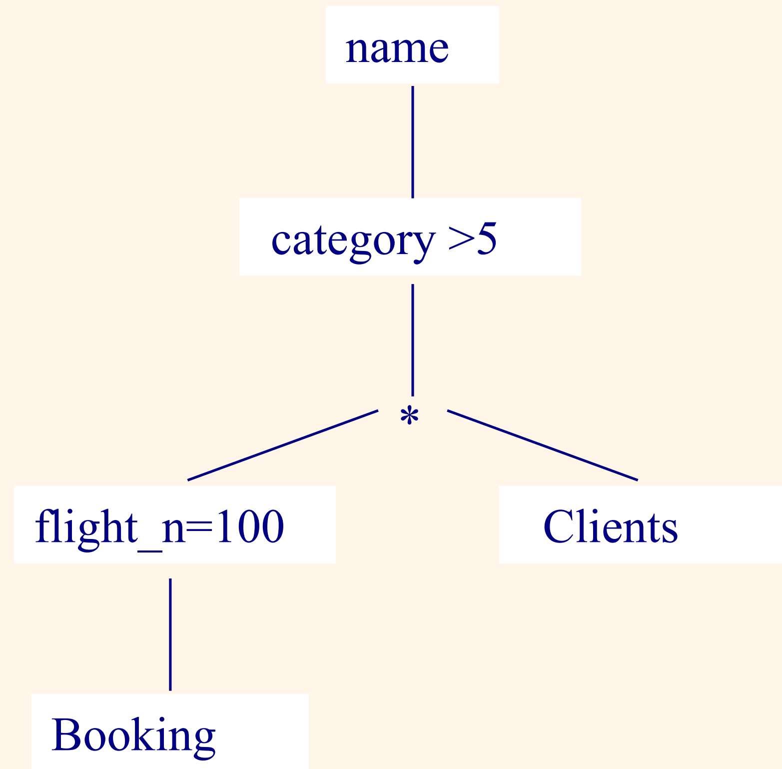
Remark: sorting - by n-way sorting algorithm (T1 with 2 passes, T2 with 4 passes)

Improvement: projections before sorting - T1[client_ID], T2[client_ID, name]:

$2p_T$*#passes

- T1 (1 page), T2 (166 pages),
- Sum: 1000+1+500+166 + 2*1*1 + 2*4*166 + 1 + 167 = 3027 I/O operations.

Alternative 3 (with indexes)



- With clustered index *flight_n* in Booking, we obtain 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- Join attribute is a key in Clients
  - at most one tuple, unclustered index on *client_ID* OK.
- The decision not to propagate *category >5* before join is based in the availability of index *client_ID* in table Clients.
- Cost: reading pages from Booking (10); for each Booking tuple 1 page from Clients is read (1000 ×);

  Sum: 1010 I/O operations

Algebraic optimization

Enables to use various strategies for join and propagate selections and projection before operation join.

- Commutativity of join and Cartesian product

$$E_1 \, [\theta] \, E_2 \approx E_2 \, [\theta] \, E_1$$

$$E_1 * E_2 \approx E_2 * E_1$$

$$E_1 \times E_2 \approx E_2 \times E_1$$

- Associativity of (theta) join and Cartesian product

$$(E_1 \, [\theta_1] \, E_2) \, [\theta_2 \wedge \theta_3] \, E_3 \approx E_1 \, [\theta_1 \wedge \theta_3] \, (E_2 \, [\theta_2] \, E_3),$$

where θ_2 includes attributes only from E_2 and E_3

$$(E_1 * E_2) * E_3 \approx E_1 * (E_2 * E_3)$$

$$(E_1 \times E_2) \times E_3 \approx E_1 \times (E_2 \times E_3)$$

Algebraic optimization

- Commutativity of selection and projection

If all the attributes from ϕ are ín $\{A_1,...,A_k\}$, then

$$E_1[A_1...A_k](\phi) \approx E_1(\phi)[A_1...A_k]$$

If $B_1,...,B_s$ are not in ϕ, then

$$E_1(\phi)[A_1...A_k] \approx E_1[A_1...A_kB_1...B_s](\phi)[A_1...A_k]$$

Remark: Propagation of selection to (basic) relations can be used also for operations $\cup$, -, $\times$.

- Commutativity of selection and Cartesian product

If all attributes are ϕ are involved in E_1, then

$$(E_1 \times E_2)(\phi) \approx E_1(\phi) \times E_2$$

Algebraic optimization

- Commutativity of selection and union

$$(E_1 \cup E_2)(\phi) \approx E_1(\phi) \cup E_2(\phi)$$

- Commutativity of selection and difference

$$(E_1 - E_2)(\phi) \approx E_1(\phi) - E_2(\phi)$$

Remark: Similarly, it is possible to use a projection.

- Commutativity of projection and Cartesian product

$$(E_1 \times E_2)[A_1 \ldots A_n] \approx E_1[B_1 \ldots B_k] \times E_2[C_1 \ldots C_m]$$

where $\cup_i B_i \cup \cup_i C_i = \cup_i A_i$, B_i concern E_1 and C_j concern E_2.

- Commutativity of projection and union

$$(E_1 \cup E_2)[A_1 \ldots A_n] \approx E_1[A_1 \ldots A_n] \cup E_2[A_1 \ldots A_n]$$

Heuristics for query optimization

1. Selections as soon as possible. Use cascades of selections, commutativity of selections with projections and $\times$, distributiveness of selection over $\cup$, $\cap$, - in such way, to get selections as close at possible to leafs.
2. Projections as soon as possible. Use cascades of projections, distributiveness of projection over $\times$, $\cup$, $\cap$, - and commutativeness of selection and projection in such way, to get projections as close as possible to leafs. Remove unnecessary projections.
3. If possible, transform $\times$ into *. Selection on 1 argument in $\times$ apply earlier.
4. Sequence of selections and/or projections replace by one selection, one projection. Use possibilities to do more operations altogether! (pipeline: e.g., if * follows, generate tuples of join)

Heuristics for query optimization

5. Use associativity of *, $\times$, $\cup$, $\cap$ to regrouping relations in the query tree in such way, so that selections producing smaller relations were called earlier.
6. Store results of common subqueries (if they are not too big).

   Remark: appropriate for queries on views

Algebraic optimization - example

D: Find titles of books having copies, which should be returned back until 30.9.2015.

D_{RA}:

(LOANS * READERS * COPIES * BOOKS) [TITLE, AUTHOR, ISBN, COPY_ID, NAME, ADDRESS, READER_ID, DATE_BACK]
(DATE_BACK < 30.9.2015) [TITLE]

Remark: D could originate as the query on view LOANS_INFO

```
SELECT TITLE
FROM LOANS_INFO
WHERE DATE_BACK < 30.9.2015
```

Algebraic optimization - example

Transformations:

(1) 2 joins from 3 joins replace by ×

((LOANS × READERS)(L.READER_ID = R.READER_ID) [COPY_ID, READER_ID, DATE_BACK, NAME, ADDRESS]

* ((COPIES × BOOKS)(C.ISBN = B.ISBN) [TITLE, AUTHOR, ISBN, COPY_ID, PURCHASE_DATE])) [TITLE, AUTHOR, ISBN, COPY_ID, NAME, ADDRESS, READER_ID, DATE_BACK] (DATE_BACK < 30.9.2015) [TITLE]

(2) remove the last * and omit PURCHASE_DATE from [ ]

(A×B)(COPY_ID = COPY_ID) [TITLE, AUTHOR, ISBN, COPY_ID, NAME, ADDRESS, READER_ID, DATE_BACK] (DATE_BACK < 30.9.2015) [TITLE]

Algebraic optimization - example

(3) Because DATE_BACK is in [] and conditions of selections commutate ⇒

(A×B)(DATE_BACK < 30.9.2015)(COPY_ID = COPY_ID)[TITLE]

Remark: unnecessary projections were removed

(4) Because DATE_BACK is only in A in relation LOANS ⇒

((LOANS(DATE_BACK < 30.9.2015) × READERS)(L. READER_ID = R. READER_ID)[COPY_ID, READER_ID, DATE_BACK, NAME, ADDRESS] × B) (COPY_ID = COPY_ID)[TITLE]

(5) Reduction of projections in () to [COPY_ID] and [COPY_ID,TITLE] ⇒ (LOANS(DATE_BACK < 30.9.2015)[COPY_ID] × (COPIES × BOOKS)(C.ISBN = B.ISBN)[COPY_ID, TITLE]) (COPY_ID = COPY_ID)[TITLE] ⇒ relation READERS disappears

Algebraic optimization - example

(6) Result in operations selection, projections, and * ⇒

(LOANS(DATE_BACK < 30.9.2015)[COPY_ID] * (COPIES * BOOKS) [COPY_ID, TITLE])[TITLE]

The query belongs to the class of SPJ-queries.

It is possible to optimize them in way to minimalize the number of joins.

(It is an NP-complete problem.)

Statistics-driven optimization

- Cost estimation for each plan: for each operation, *cost* and *size result* estimations are performed
- Information about R* size and indexes is needed.
- *Data catalogues* typically contain a description of relation R and indexes:
  - n_R (# tuples) and p_R (# pages)
  - $V(A,R) = |R[A]|$ (tj. $|adom_A|$) through the index
  - $p_{R.A}$ (# leaf pages B^+-tree index for R.A).
  - $I(A,R)$ – the depth of B^+-tree for index R.A, min/max values for each B^+-tree index, $2min_A$, $2max_A$ (the second lowest, resp. highest value in $adom_A$)
- More detailed information (e.g., histograms for $adom_A$)

Result size estimation and reduction factors

```
SELECT list_of_attributes
FROM list_of_relations
WHERE atom_1 AND ... AND atom_k
```

- Maximum # tuples in result is given by a product of relations cardinalities being in the FROM clause.
- *Reduction factor (RF)* associated with each *atom* reflects the impact of the atom in reducing result size.
- *Result cardinality* = Max # tuples * product of all RF.
- Implicit assumption: terms are independent!

Estimation of size result and RFs

SELECT list of attributes
FROM list of relations
WHERE atom_1 AND ... AND atom_k

- atom A=k
  - RF = 1/V(A.R), given index on A
  - RF = 1/10 index does not exist
- atom A=B
  - RF = 1/MAX(V(A,R), V(B,S)), given indexes on A and B
  - RF = 1/V(A,R) given an index on A
  - RF = 1/10 no index exist
- atom A>k
  - RF = (2max-k)/(2max-2min), given an index A
  - RF < ½ if A is not of integer type or index does not exist

Optimization using rough estimation of RFs

Strategy: estimations of RF for operators

Ex..: rough estimations by constants

$RF_{=} = 20\%$, $RF_{>} = 40\%$

$\Rightarrow$ FLIGHT.Cost > 26.000 (1)

FLIGHT.Cost > 7.000 (2)

have the same RF, because evidently

$RF1_{>real} < RF2_{>real}$

Example: Informix Online

Assumptions: i-attribute is attribute with index, *k* is constant, *m* is estimation of subquery size.

selekční condition	RF
R.i-attribute = k	
R.i-attribute IS NULL	1/V(R.i-attribute, R)
R.i-attribute = S.i-attribute	1/max(V(R.i-attribute,R), V(S.i-attribute,S))
i-attribute > k	(2max - k)/(2max -2min)
i-attribute < k	(k -2min)/(2max -2min)
attribute = expression	
attribute = NULL	1/10
attribute LIKE expression	1/5

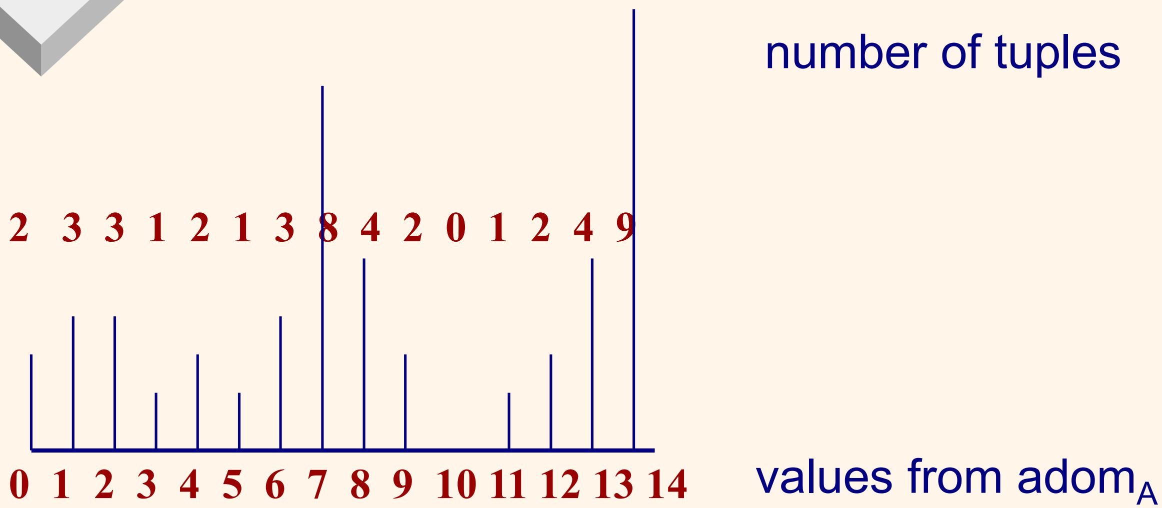
Example: Informix Online

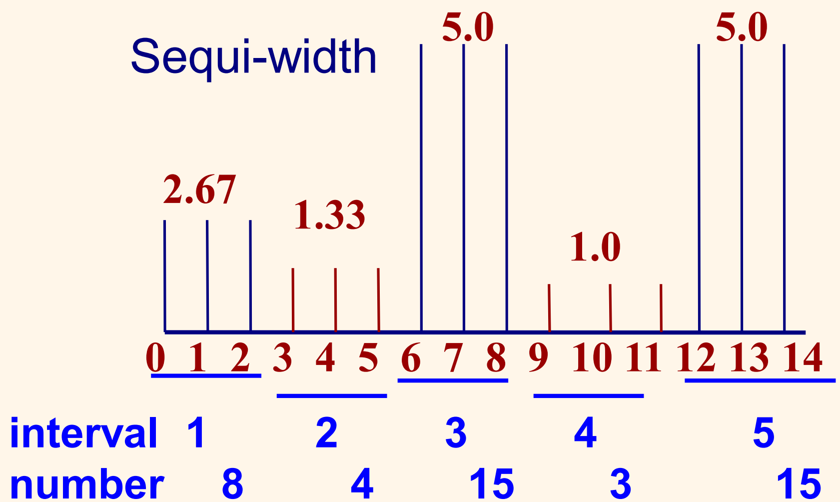
Selection condition	RF
attribute > expression	
attribute < expression	1/3
EXISTS subquery	1, if there is estimation, that TRUE
	0, otherwise
NOT selection	$1\text{-}RF_{selection}$
$selection_1$ AND $selection_2$	$RF_{selection1} * RF_{selection2}$
$selection_1$ OR $selection_2$	$RF_{selection1} + RF_{selection2} -$
	$RF_{selection1} * RF_{selection2}$
attribute IN list-of-values	$\Leftrightarrow$ attribute = k_1 OR … attribute = k_m
attribute θ ANY subquery	$\Leftrightarrow$ attribute θ k_1 OR … attribute θ k_m

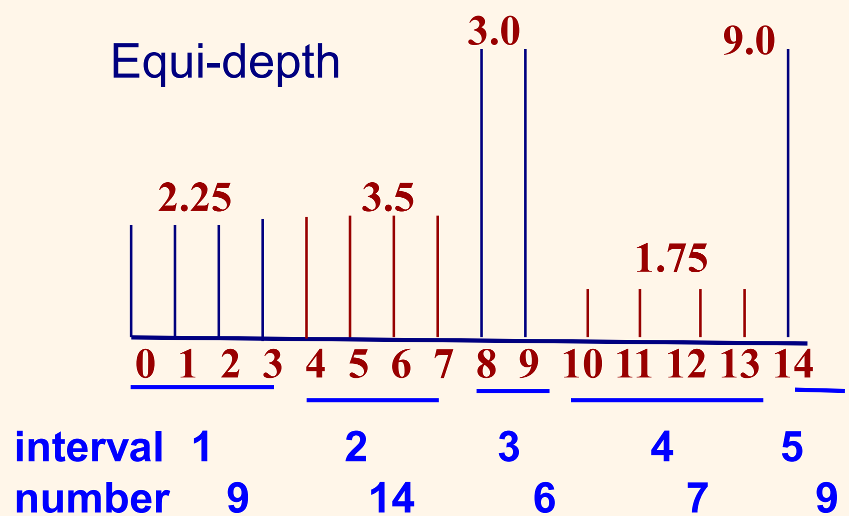
Statistics driven optimization

- *Histograms*
  - the assumption of uniform distribution is not real in applications
  - a histogram on attribute is constructed by partitioning the data distribution D into mutually disjoint subsets called buckets and approximating the frequencies f and values V in each bucket in some common fashion, i.e., histograms approximate real data distribution
  - they are maintained by DBMS
- Kinds of histograms
  - Equi-width: divides value range of the column into intervals supposing, that value distribution in interval is uniform
  - Equi-depth: number of tuples in interval is appr. of the same size

Statistics driven optimization







Enumeration of alternative paths

- Two main cases:
  - plans for a single relation
  - plans for more relations
- queries over single relation are composed from operations selection, projection (and aggregation operations):
  - each available access path (scanning file/index) is considered and the one with the least estimated cost is chosen.
  - Two different operations can be performed altogether (e.g., when an index is chosen for selection, projection is done for each selected tuple and tuples are moved (pipelined) into aggregation calculation).

Example: System R

Assumptions: Simple query q over relation R, some attributes with index, V(A, R)

- indexes:
  - clustered (R(A=c) is ≈ in minimal amount of pages)
  - unclustered (R(A=c) is ≈ in n_R/V(A,R) pages)

Method: choose the cheapest strategy from (1)-(8) and on the result use the rest of conditions from *q*

(1) A = c, where there is a clustered index for A

Cost: p_R/V(A,R)

(2) A θ c, where $\theta \in \{\geq, \leq, <, >\}$ and there is a clustered index for A

Cost: p_R/2

Remark: for ≠ it is necessary to read ≈ entire R ⇒ (5)

Example: System R

(3) A = c, where for A there is unclustered index

Cost: $n_R/V(A,R)$

(4) When R is a sequential file, then the entire R is read.

Cost: p_R

(5) when R „mixed“ with other relations and there is a clustered index for arbitrary attribute (group of attributes), then the whole R is read „over“ it.

Cost: p_R

(6) A θ c, where $\theta \in \{\geq, \leq, <, >\}$ and for A there is unclustered index

Cost: $n_R/2$

Example: System R

(7) If there is any unclustered index, the entire R is read „over" it.

Cost: n_R

(8) (1)-(7) are not applicable. Then all pages potentially containing R are read.

Cost: $\geq n_R$

Remark: A = c AND B=d and there is index on A and B as well.

A better strategy would be „over both indexes" $\Rightarrow$ intersection of two lists of pointers

Estimation of the plan cost for one relation – more precisely with RF

- Index on primary key A satisfies an equality:
  Cost: I(A,R)+1 for B^+-tree, about 1.2 for hashed index.
- Clustered index I satisfies 1 or more comparisons:
  $(p_{R.A} + p_R)$ * product RF of satisfying selections.
- Non-clustered index I satisfies 1 or more selections:
  $(p_{R.A} + n_R)$ * product RF of satisfying selections.

projections were performed without elimination of duplicates!

Example

```
SELECT  C.client_ID
FROM Clients C
WHERE C.category=8
```

- There is an index on *category*:

$(1/V(A,R))*n_C = (1/10) * 40000$ tuples should be selected.

  - clustered index: $(1/V(A,C)) * (p_{C.category} + p_C) = (1/10) * (50+500)$ pages are selected.
  - unclustered index: $(1/V(A,C)) * (p_{C.category} + n_C) = (1/10) * (50+40000)$ pages are selected.

- There is an index on *client_ID*:
  - All tuple/pages should be read. Index is not usable. The whole file C (500) is scanned.

Queries involving more relations

- Since the number of joins is increasing, the number of alternative plans is quickly increasing; it is necessary to restrict the search space.
  - For n relations $R_1,...,R_n$ the number of plans is $(2(n-1))!/(n-1)!$, e.g., for $n=7$ it is 665280.
- Solution: using dynamic programming;
- S contains n relations. For finding the best plan for S, consider all possible plans of form: $S_1 * (S – S_1)$, where S_1 is any non-empty subset of S.
- Recursively calculate cost of each plan. Choose the cheapest of the $2^n - 2$ alternatives.
- Basic case for recursion: Access plan for particular relation.
  - apply all selections on R_i using the best choices of indexes on R_i.
- When the plan for any subset is computed, store it and reuse, when it is required again. Thus, it is not necessary to generate all join orders .

Queries involving more relations

procedure findbestplan(*S*)
    if (*bestplan*[*S*].*cost* $\neq \infty$) **return** *bestplan*[*S*]
                //else: *bestplan*[*S*] ještě was not calculated, calculate it now
    if (*S* contains only 1 relation)
        set the *bestplan*[*S*].*plan* and *bestplan*[*S*].*cost* according to the best access to *S*
    else for each *S*1 $\subseteq$ *S* such that *S*1 $\neq \emptyset$ and *S*1 $\neq$ *S*
        P1= findbestplan(*S*1)
        P2= findbestplan(*S* - *S*1)
        A = the best algorithm for join of results *P*1 and *P*2
        cost = *P*1.*cost* + *P*2.*cost* + cost *A*
        if *cost* < *bestplan*[*S*].*cost* **then**
            bestplan[*S*].*cost* = cost
            bestplan[*S*].*plan* = "call *P*1.*plan*; call *P*2.*plan*;
                join results *P*1 and *P*2 by algorithm *A*"
    return *bestplan*[*S*]

- Complexity: $O(3^n)$

Queries involving more relations

- Essential decision in System R: for * only those *linear trees* are considered, which are of type *left-deep*.

  Df.: linear: each non-leaf node has at least one child from **R**

  Df.: left-deep: each right-hand-side child is from **R**

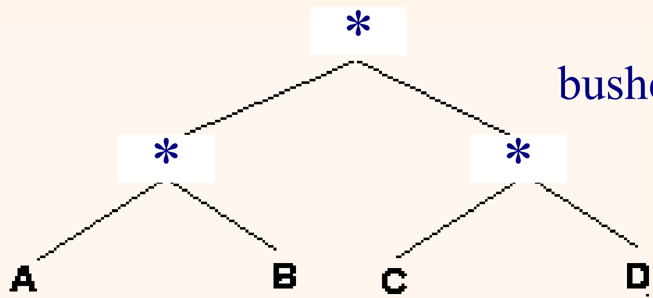
- left-deep joins enable generate *fully piplined* plans.
  - Intermediate results have to be not stored into temporary files

  Remark: not all left-deep plans are fully pipelined (depends on the algorithm of join operation, e.g., sort-merge)

- It is not necessary to generate all join orders. Using dynamic programming, the cheapest alternative is generated only once for each subset $\{R_1, \ldots, R_n\}$ and stored.

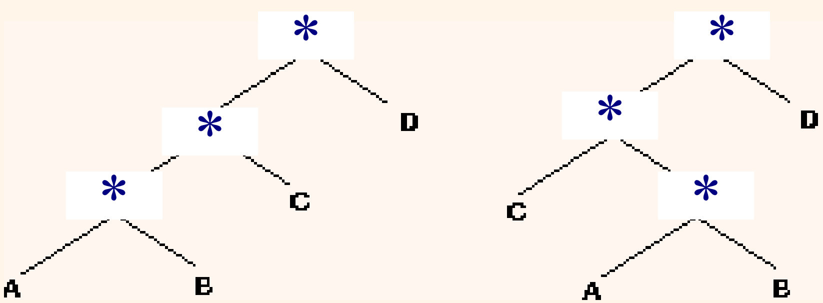
  Remark: There are $O(n*2^n)$ left-deep plans.

Queries involving more relations



bushes

nelinear trees



linear trees

left-deep

Enumeration on left-deep plans

Left-deep plans distinguish only in order of relations, access method for each relation and method of a join for each relation.

- Algorithm modification:
  - replace "**for each** $S1 \subseteq S$ such that $S1 \neq \emptyset$ and $S1 \neq S$"
  - by expression „**for** r $\in$ S
    Let S1 = S – r"
- Enumerated using *n* passes (if *n* relations joined):
  - Pass 1: Find the best 1-relation plan for each relation.
  - Pass 2: Find the best way to join result of each 1-relation plan (as outer) to other relation (all 2-relation plans)
  - Pass n: Find the best way to join the result of the (n-1)-relation plan (outer) to the *n*th relation (all *n*-relation plans)
- Time complexity is $O(n2^n)$

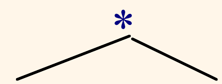
Finding „the best“ left-deep plan

procedure findbestplan(*S*)
  if (*bestplan*[*S*].*cost* ≠ ∞) **return** *bestplan*[*S*]
    //else: *bestplan*[*S*] has not yet been calculated, calculate it now
  if (*S* contains onlyt 1 relation)
    set *bestplan*[*S*].*Plan* and *bestplan*[*S*].*cost* according to the best access to *S*
  else for r ∈ S
    let S1 = S – r
    P1= findbestplan(*S*1)
    P2= findbestplan(*S* - *S*1)
    A = the best algorithm for join of results *P*1 and *P*2
    cost = *P*1.*cost* + *P*2.*cost* + cost *A*
    if *cost* < *bestplan*[*S*].*cost* **then**
      bestplan[*S*].*cost* = cost
      bestplan[*S*].*Plan* = “call *P*1.*plan*; call *P*2.*plán*; join results *P*1 and *P*2 by algorithm *A*”
  return *bestplan*[*S*]

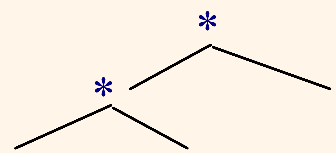
Calculation of left-deep plans

Pass 1 **A B C D**

Pass 2



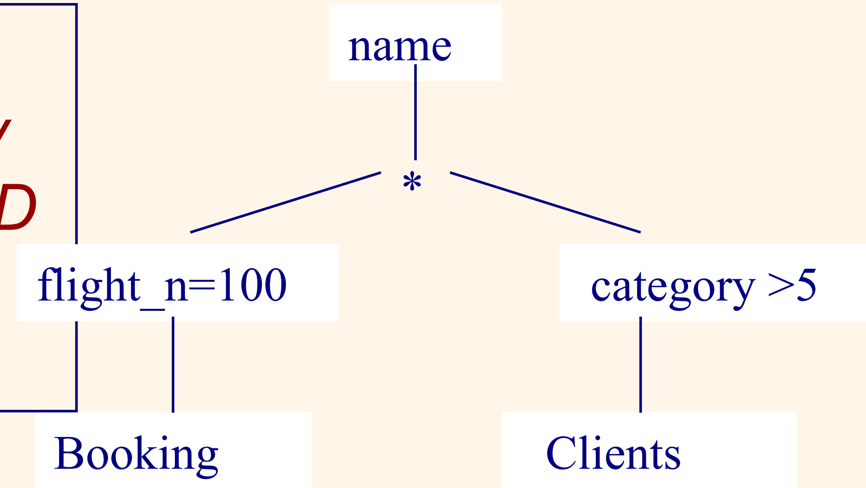
Pass n



- For each subset of relations only the cheapest plan (for each *interesting tuple ordering* - see sorting, merging, group by).

Example

<u>Clients:</u>
B^+-tree on *category*
hashing on *client_ID*
<u>Booking:</u>
B^+-tree on *flight_n*



Pass 1:

- *Clients*: B^+-tree matches on *category>5*, and is probably cheapest. But the result is a set of tuples, index is unclustered, scanning the file can be cheaper.
  - The plan with B^+-tree (sorted by *category*) is held
- *Booking*: B^+-tree agrees on *flight_n*=100; the cheapest.

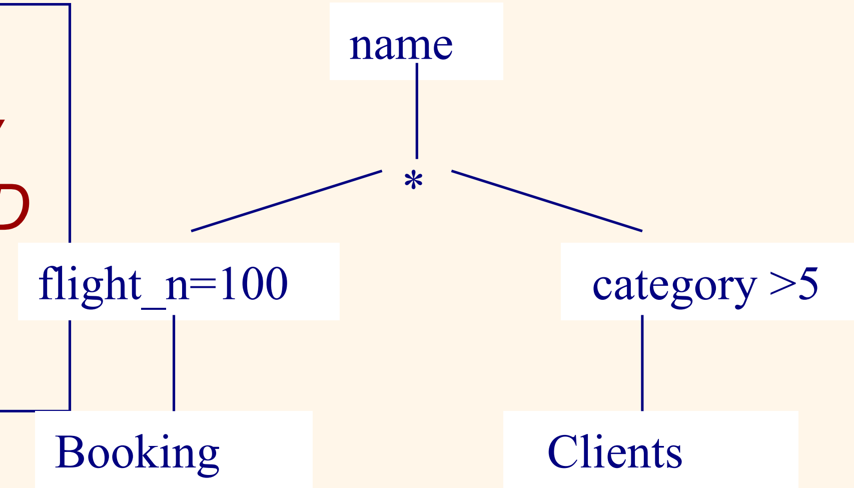
Pass 2:

We consider each plan retained from Pass 1 as the outer and consider, how to join it with the other relation

- *Booking* as the outer: by hashing to Clients tuples, that satisfy *client_ID* = value of *client_ID* of outer tuple.

Example

Clients:
B+-tree on *category*
hashing on *client_ID*
Booking:
B+-tree on *flight_n*



Pass 1:

- *Clients*: B+-tree matches on *category>5*, and is probably cheapest. But the result is a set of tuples, index is unclustered, scanning the file can be cheaper.
  - The plan with B+-tree (sorted by *category*) is held
- *Booking*: B+-tree agrees on *flight_n*=100; the cheapest.

Pass 2:

We consider each plan retained from Pass 1 as the outer and consider, how to join it with the other relation

Query blocks: units of optimization

```
SELECT  C.name
FROM  Clients S
WHERE  C.age IN
    (SELECT  MAX (S2.age)
     FROM  Clients S2
     GROUP BY  S2.category)
```

outer block *nested block*

- Query in SQL is split into a collection of *block queries*, which are optimized always 1 block in time.
- A nested block corresponds (simply) to a procedure call for each tuple from outer block
  – for each block, the following plans are considered:
  – all available access methods for ∀ relation in the FROM clause.
  – all trees for left-deep joins (how to join with relations in inner FROM (permutations and join methods are considered)

Nested queries

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
- Outer block is optimized with the cost of `calling' nested block computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered. The non-nested version of the query is typically optimized better.

```
SELECT  C.name
FROM  Clients C
WHERE EXISTS
   (SELECT  *
    FROM  Booking B
    WHERE  B.flight_n=103
    AND  B.client_ID=C.client_ID)
```

nested block k optimization:

```
SELECT  *
FROM  Booking B
WHERE B.flight_n=103 AND
C.client_ID = outer value
```

Equivalent non-nested query:

```
SELECT  C.name
FROM Clients C, Booking B
WHERE  C. client_ID =B. client_ID
AND B.flight_n=103
```

Syntax driven optimization

Ex.: SELECT * FROM Copies (1)
WHERE Cost >'40' AND Issue_country = 'GB'

SELECT * FROM Copies (2)
WHERE Issue_country = 'GB' AND Cost >'40'

In some DBMS the evaluation depends of the order of conditions:

The one with the lowest RF is evaluated first.

$\Rightarrow$ variant (2) is more effective than (1).

Syntax driven optimization

How to bypass the optimizer?

Ex.: SELECT * FROM Copies (1)
WHERE (Purchase_date >'23.04.99' AND
Issue_country = 'GB') OR ISBN = '486';

(SELECT * FROM Copies (2)
WHERE Purchase_date >'23.04.99' AND
Issue_country = 'GB')
UNION
(SELECT * FROM Copies WHERE ISBN = '486');

Tendency of optimizer: (1) sequentially, (2) with indexes for subqueries

Summary

- Query optimization is an important task solved by DBMS.
- Other approaches:
  - based on rules
  - probability algorithms
  - parametrized optimization
- It is necessary to understand optimization, to understand an influence of DB design (relations, indexes) on the load (set of queries).
- Trend: autonomous DBMS with AI.
  - Ex..: platform Oracle 18c based on machine learning. DB is automatically upgraded, optimizes at run time, DBA is not necessary.