COP 3540: Introduction to Database Structures

Fall 2017

Query Optimization II

M. Rathod

Query Optimization

The following schema is used for the queries:

Sailors(*sid:* **integer**, *sname:* **string**, *rating:* **integer**, *age:* **real**)
Boats(*bid:* **integer**, *bname:* **string**, *color:* **string**)
Reserves(*sid:* **integer**, *bid:* **integer**, *day:* **dates**, *rname:* **string**)

We will also assume:

- Reserves:
  - each tuple is 40 bytes long
  - a page can hold 100 tuples
  - 1,000 pages of such tuples
- Sailors:
  - each tuple 50 bytes long
  - a page can hold 80 tuples
  - 500 pages of such tuples

FAU

Decompostion

Decomposition of a Query into Blocks:

- SQL queries are decomposed (parsed) into smaller units called **query blocks**
- Query optimizer optimizes a single block at a time
- **Query block** is SQL query with no nesting and exactly one `SELECT` clause and one `FROM` clause and at most one `WHERE` clause, `GROUP BY` clause, and `HAVING` clause

Decompostion

Decomposition of a Query into Blocks:

For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat

```
SELECT    S.sid, MIN (R.day)
FROM      Sailors S, Reserves R, Boats B
WHERE     S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' AND
          S.rating = ( SELECT MAX (S2.rating)
                       FROM   Sailors S2 )
GROUP BY  S.sid
HAVING    COUNT (*) > 1
```

nested block

outer block

Decompostion

Decomposition of a Query into Blocks:

For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat

```
SELECT    S.sid, MIN (R.day)
FROM      Sailors S, Reserves R, Boats B
WHERE     S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' AND
          S.rating = Reference to nested block
GROUP BY  S.sid
HAVING    COUNT (*) > 1
```

outer block

```
SELECT MAX (S2.rating)
FROM   Sailors S2
```

nested block

Translating SQL into Algebra

A Query Block as a Relational Algebra Expression

For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat

outer block

```
SELECT   S.sid, MIN (R.day)
FROM     Sailors S, Reserves R, Boats B
WHERE    S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' AND
         S.rating = Reference to nested block
GROUP BY S.sid
HAVING   COUNT (*) > 1
```

$$\pi_{S.sid,MIN(R.day)}($$
$$HAVING_{COUNT(*)>2}($$
$$GROUP\ BY_{S.sid}($$
$$\sigma_{S.sid=R.sid \wedge R.bid=B.bid \wedge B.color='red' \wedge S.rating=value\_from\_nested\_block}($$
$$Sailors \times Reserves \times Boats))))$$

Translating SQL into Algebra

A Query Block as a Relational Algebra Expression

SQL query block can be expressed as an extended algebra expression in the following manner:

`SELECT` clause corresponds to π

`WHERE` clause corresponds to σ

`FROM` clause corresponds to cross-product of relations (x)

a query is essentially treated as a σ π x algebra expression

Translating SQL into Algebra

A Query Block as a Relational Algebra Expression

```
SELECT    S.sid, MIN (R.day)
FROM      Sailors S, Reserves R, Boats B
WHERE     S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' AND
          S.rating = Reference to nested block
GROUP BY  S.sid
HAVING    COUNT (*) > 1
```

outer block

- The σ π $\times$ expression for the query is

$$\pi_{S.sid,R.day}(\sigma_{S.sid=R.sid \wedge R.bid=B.bid \wedge B.color='red' \wedge S.rating=value\_from\_nested\_block}(Sailors \times Reserves \times Boats))$$

Translating SQL into Algebra

A Query Block as a Relational Algebra Expression

- `GROUP BY` and `HAVING` operations are added to the projection list
- `SELECT` clause aggregate operation `MIN(R.day)` is computed after first computing the σ π $\times$

```
SELECT    S.sid, MIN (R.day)
FROM      Sailors S, Reserves R, Boats B
WHERE     S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' AND
          S.rating = Reference to nested block
GROUP BY S.sid
HAVING    COUNT (*) > 1
```

$$\pi_{S.sid,MIN(R.day)}($$
$$HAVING_{COUNT(*)>2}($$
$$GROUP\ BY_{S.sid}($$
$$\pi_{S.sid,R.day}($$
$$\sigma_{S.sid=R.sid \wedge R.bid=B.bid \wedge B.color='red' \wedge S.rating=value\_from\_nested\_block}($$
$$Sailors \times Reserves \times Boats)))))$$

-

Estimating the Cost of a Plan

Cost of an evaluation plan for a query block is done by:

1. For each node in the tree, estimate the cost of performing the corresponding operation (determine if pipelining is used or temporary relations are created to pass the output of an operator to its parent)

2. For each node in the tree, estimate the size of the result, and if it is sorted

Estimating the Cost of a Plan

Estimating Result Sizes

```
SELECT  attribute list
FROM    relation list
WHERE   term_1 ∧ term_2 ∧ ... ∧ term_n
```

- maximum number of tuples in the result (without duplicate elimination) is the product of the cardinalities of the relations in the `FROM` clause.
- **reduction factor** - ratio of the (expected) result size to the input size considering only the selection represented by the term - can be associated to each term in the `WHERE` clause
- actual size of the result is estimated as

  maximum size x the product of the reduction factors for the terms in the `WHERE` clause.

Estimating the Cost of a Plan

Estimating Result Sizes

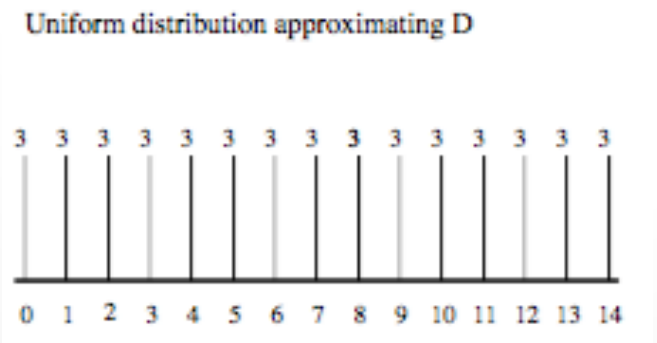
- column = value
  - If there is an index *I* on column for the relation, reduction factor = 1/Nkeys(*I*) (assuming uniform distribution in index key values)
  - If there is no index on column, an optimizer can arbitrarily assume a reduction factor of 1/10
- column1 = column2
  - if there are indexes I1 and I2 on column1 and column2, , reduction factor = I/MAX (Nkeys(*I1*), Nkeys(*I2*)) (assuming that each key value in the smaller index, say *I*1, has a matching value in the other index *I*2)
  - If only one of the two columns has an index *I*, reduction factor = 1/Nkeys(*I*)
  - if neither column has an index, approximate it to 1/10
  - formulas are used whether or not the two column appear in the same relation
- column > value
  - If there is an index *I* on column, reduction factor = High(*I*) – value/High(*I*) – Low(*I*)
  - If there is no index, or column is not of an arithmetic type, a fraction < ½ is chosen
- column IN (list of values)
  - Reduction factor is same as column= value multiplied by # of items in list (at most half)

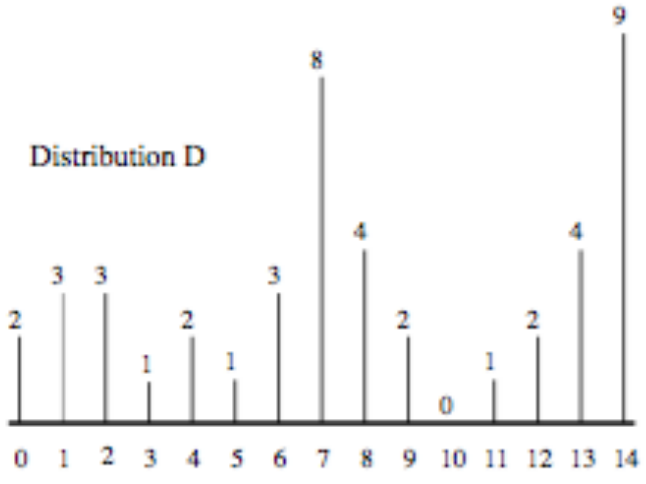
Estimating the Cost of a Plan

Improved Statistics: Histograms

approximate the distribution of key values *I* as accurately as possible

- say we are looking at age and how many tuples of that age value exist (frequency)
- for non-uniform distribution:
  - selection age >13 will return 9 tuples
- for uniform distribution:
  - the estimate is 1/15 * 45 = 3 tuples
  - Inaccurate!





Estimating the Cost of a Plan

Improved Statistics: Histograms

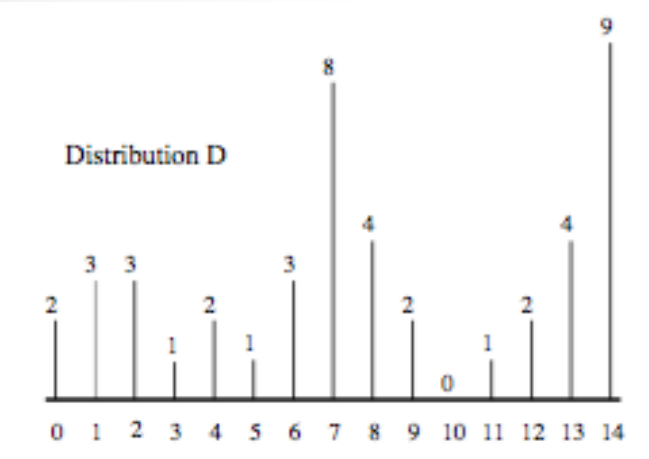
- **histogram** is a data structure maintained by a DBMS to approximate a data distribution
- data distribution can be approximated by dividing the range of age values into sub-ranges called buckets
- count the number of tuples with age values within that bucket
- two kinds of histograms
  - equiwidth
  - equidepth

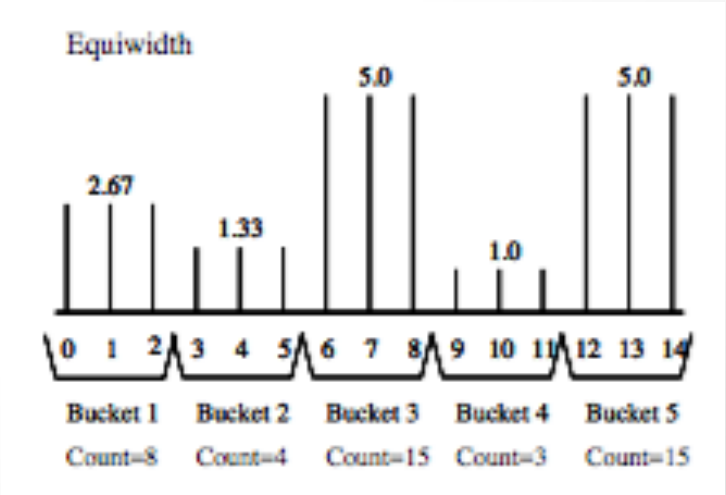
Estimating the Cost of a Plan

Improved Statistics: Histograms

equiwidth histogram

- assume distribution within the bucket is uniform
- divide the range into sub-ranges of equal size (in terms of the age value range)
- for query age > 13, estimate size of result to be 5 (bucket 5 represents 15 tuples, selected range corresponds to 1/3*15 = 5 tuples)
- small # of buckets = more accurate estimates
  - (cost of few hundred bytes per histogram)



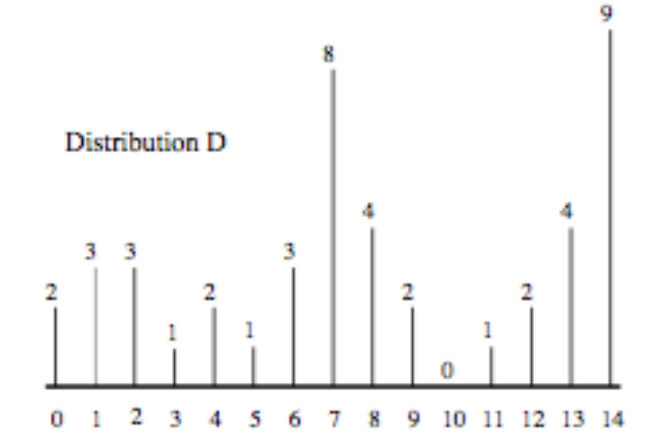


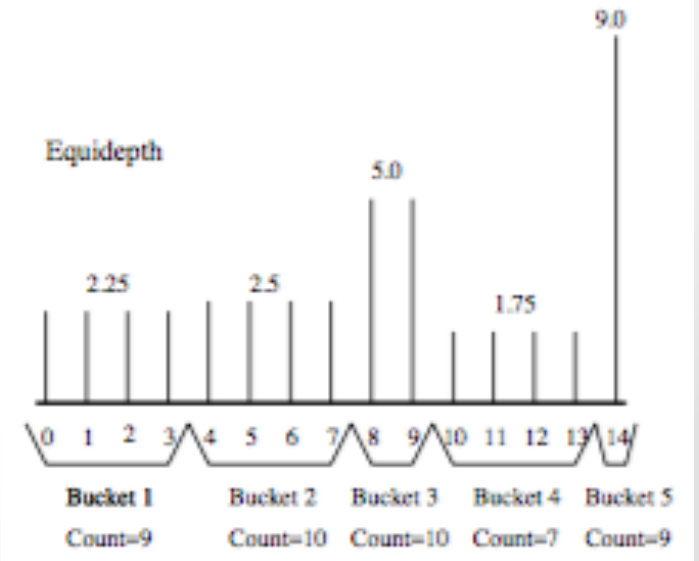
Estimating the Cost of a Plan

Improved Statistics: Histograms

equidepth histogram

- divide the range into sub-ranges where number of tuples within each sub-range (bucket) is equal
- frequently occurring values contain fewer values, and uniform distribution assumption is applied to a smaller range of values, leading to better approximations
- buckets with mostly infrequent values are approximated less accurately
- for query age > 13, we use bucket 5, which contains only the age value 15, and thus we arrive at the exact answer, 9
- equidepth histograms provide better estimates than equiwidth histograms

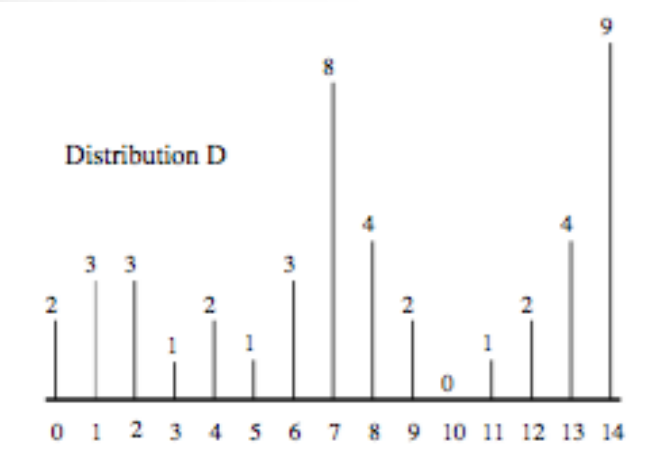


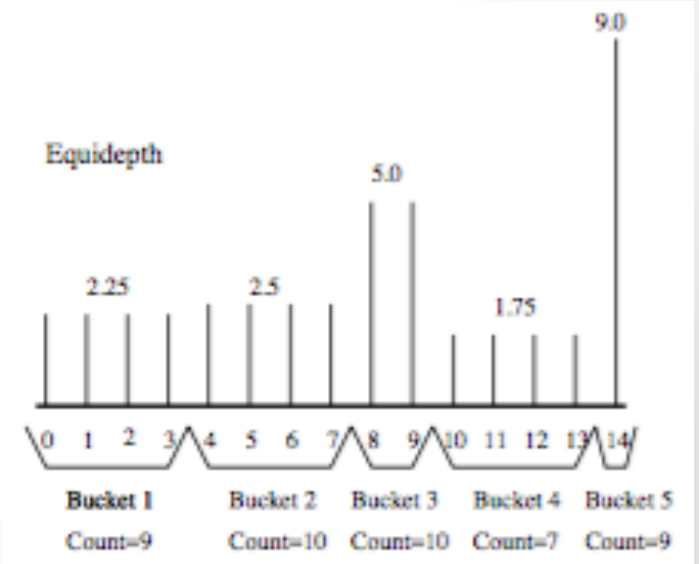


Estimating the Cost of a Plan

Improved Statistics: Histograms

- **compressed** histogram - separately maintain counts for a small number of very frequent values, say the age values 7 and 14
- most commercial DBMSs currently use equidepth histograms, and some use compressed histograms





Relation Algebra Equivalencies

equivalent – if two algebra expression over the same set of input relations produce the same result on all instances

algebra equivalences allow us to:

- convert cross-products to joins
- to choose different join orders
- to push selections and projections ahead of joins

Relation Algebra Equivalencies

Selections

- **cascading** of selections:

$$\sigma_{c_1 \wedge c_2 \wedge \ldots cn}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{cn}(R))\ldots))$$

from the right side to the left, this equivalence allows us to combine several selections into one selection

- selections are **commutative**:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

we can test the conditions c1 and c2 in either order

Relation Algebra Equivalencies

Projections

- **cascading** projections:

$$\pi_{a_1}(R) \equiv \pi_{a_1}(\pi_{a_2}(\ldots(\pi_{a_n}(R))\ldots))$$

- successively eliminating columns from a relation is equivalent to simply eliminating all but the columns retained by the final projection

- a_i is a set of attributes of relation R
- $a_i \subseteq a_i{+}1$ for $i = 1 \ldots n - 1$
- equivalence is useful in conjunction with other equivalences such as commutation of projections with joins

Relation Algebra Equivalencies

Cross-Products and Joins

commutative :

$$R \times S \equiv S \times R$$

$$R \bowtie S \equiv S \bowtie R$$

- allows us to choose which relation is to be the inner and which the outer in a join of two relations

associative:

$$R \times (S \times T) \equiv (R \times S) \times T$$

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

- regardless of the order in which the three relations are considered, the final result contains the same columns (selections specifying the join conditions can be cascaded)

Relation Algebra Equivalencies

Cross-Products and Joins

verify: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

1. from commutativity:

$$R \bowtie (S \bowtie T) \equiv R \bowtie (T \bowtie S)$$

2. From associativity:

$$R \bowtie (T \bowtie S) \equiv (R \bowtie T) \bowtie S$$

3. commutativity again:

$$(R \bowtie T) \bowtie S \equiv (T \bowtie R) \bowtie S$$

Relation Algebra Equivalencies

Selects, Projects, and Joins

commute a selection with a projection:

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

- every attribute mentioned in the selection condition c must be included in the set of attributes a

combine a selection with a cross-product to form a join:

$$R \bowtie_c S \equiv \sigma_c(R \times S)$$

Relation Algebra Equivalencies

Selects, Projects, and Joins

commute a selection with a cross-product or a join if the selection condition involves only attributes of one of the arguments to the cross-product or join:

$$\sigma_c(R \times S) \equiv \sigma_c(R) \times S$$

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

- attributes mentioned in c must appear only in R, and not in S
- similar equivalences hold if c involves only attributes of S and not R

Relation Algebra Equivalencies

Selects, Projects, and Joins

selection σ_c on R × S can be replaced by a cascade of selections σc_1 , σc_2 , and σc_3 such that c_1 involves attributes of both R and S, c_2 involves only attributes of R, and c_3 involves only attributes of S:

$$\sigma_c(R \times S) \equiv \sigma_{c_1 \wedge c_2 \wedge c_3}(R \times S)$$

using the cascading rule for selections:

$$\sigma_{c_1}(\sigma_{c_2}(\sigma_{c_3}(R \times S)))$$

using the rule for commuting selections and cross-products:

$$\sigma_{c_1}(\sigma_{c_2}(R) \times \sigma_{c_3}(S))$$

we can push part of the selection condition c ahead of the cross-product

Relation Algebra Equivalencies

Selects, Projects, and Joins

commute projection with **cross product:**

$$\pi_a(R \times S) \equiv \pi_{a_1}(R) \times \pi_{a_2}(S)$$

a_1 is the subset of attributes in a that appear in R

a_2 is the subset of attributes in a that appear in S

Relation Algebra Equivalencies

Selects, Projects, and Joins

commute projection with a **join** (if the join condition involves only attributes retained by the projection)**:**

$$\pi_a(R \bowtie_c S) \equiv \pi_{a_1}(R) \bowtie_c \pi_{a_2}(S)$$

a_1 is the subset of attributes in a that appear in R
a_2 is the subset of attributes in a that appear in S
every attribute mentioned in the join condition c must appear in a

If a does not include all attributes mentioned in c, generalize the commutation rules by

- project out attributes that are not mentioned in c or a performing the join
- project out all attributes that are not in a

$$\pi_a(R \bowtie_c S) \equiv \pi_a(\pi_{a_1}(R) \bowtie_c \pi_{a_2}(S))$$

a_1 is the subset of attributes of R that appear in either a or c
a_2 is the subset of attributes of S that appear in either a or c

Relation Algebra Equivalencies

other equivalences

- Union and intersection are associative and commutative
- Selections and projections can be commuted with each of the set operations (set-difference, union, and intersection)

Enumeration of Alternative Plans

Single-Relation Queries

For each rating greater than 5, print the rating and the number of 20-year-old sailors with that rating, provided that there are at least two such sailors with different names

```
SELECT    S.rating, COUNT (*)
FROM      Sailors S
WHERE     S.rating > 5 AND S.age = 20
GROUP BY  S.rating
HAVING    COUNT DISTINCT (S.sname) > 2
```

$$\pi_{S.rating,COUNT(*)}($$
$$HAVING_{COUNTDISTINCT(S.sname)>2}($$
$$GROUP\ \ BY_{S.rating}($$
$$\pi_{S.rating,S.sname}($$
$$\sigma_{S.rating>5 \wedge S.age=20}($$
$$Sailors)))))$$

Enumeration of Alternative Plans

Single-Relation Queries

Plans without Indexes

- scan the Sailors relation
- apply the selection and projection (without duplicate elimination) operations to each retrieved tuple

$$\pi_{S.rating,S.sname}(\sigma_{S.rating>5 \wedge S.age=20}(Sailors))$$

- resulting tuples are then sorted according to the `GROUP BY` clause
- one answer tuple is generated for each group that meets the condition in the `HAVING` clause

Enumeration of Alternative Plans

Single-Relation Queries

Plans without Indexes

Cost consists of:

1. Perform a file scan to retrieve tuples and apply the selections and projections
2. Write out tuples after the selections and projections
3. Sort these tuples to implement the `GROUP BY` clause

`HAVING` clause does not cause additional I/O (aggregate computations can be done on-the-fly wrt I/O as we generate the tuples in each group at the end of the sorting step for the `GROUP BY` clause

Enumeration of Alternative Plans

Single-Relation Queries - Plans without Indexes

Cost consists of:

- cost of a file scan on Sailors + the cost of writing out ⟨S.rating, S.sname⟩ pairs + the cost of sorting as per the `GROUP BY` clause
- cost of the file scan is NPages(Sailors) is 500 I/Os
- cost of writing out ⟨S.rating, S.sname⟩ pairs is
- NPages(Sailors) x size of such a pair/size of a Sailors tuple x reduction factors of the two selection conditions = 20 I/Os
  - tuple size ratio is about 0.8
  - rating selection has a reduction factor of 0.5
  - default factor of 0.1 for the age selection
- cost of sorting intermediate relation (Temp) = 3*NPages(Temp)= 60 I/Os (assume that enough pages are available in the buffer pool to sort it in two passes)
- The total cost of the example query is therefore 500 + 20 + 60 = 580 I/Os

Enumeration of Alternative Plans

Single-Relation Queries

Plans Utilizing an Index

1. Single-index access path:

- If several indexes match the selection conditions in the `WHERE` clause, each matching index offers an alternative access path
- optimizer can choose the access path resulting in retrieving the fewest pages, apply any projections and non-primary selection terms (parts of the selection condition that do not match the index), then proceed to compute the grouping and aggregation operations (by sorting on the `GROUP BY` attributes)

2. Multiple-index access path:

- If several indexes using (2) or (3) for data entries match the selection condition, each such index can be used to retrieve a set of rids
- intersect these sets of rids, then sort the result by page id, retrieve tuples that satisfy the primary selection terms of all the matching indexes
- projections and non-primary selection terms can then be applied, followed by grouping and aggregation operations

Enumeration of Alternative Plans

Single-Relation Queries

Plans Utilizing an Index

3. Sorted index access path:

- if the list of grouping attributes is a prefix of a tree index, the index can be used to retrieve tuples in the order required by the `GROUP BY` clause
- selection conditions applied on each retrieved tuple, unwanted fields can be removed, and aggregate operations computed for each group
- strategy works well for clustered indexes

4. Index-Only Access Path:

- If all the attributes mentioned in the query (in the `SELECT`, `WHERE`, `GROUP BY`, or `HAVING` clauses) are included in the search key for some dense index on the relation in the `FROM` clause, an index-only scan can be used to compute answers.
- apply selection conditions, remove unwanted attributes, sort the result to achieve grouping, and compute aggregate functions within each group
- if the index is a tree index and the list of attributes in the `GROUP BY` clause forms a prefix of the index key, retrieve data entries in the order needed for the `GROUP BY` clause (avoid sorting)

Enumeration of Alternative Plans

Multiple-Relation Queries

- Query blocks that contain two or more relations in the `FROM` clause require joins (or cross-products)

- size of the final result can be estimated by taking the product of the sizes of the relations in the `FROM` clause and the reduction factors for the terms in the `WHERE` clause

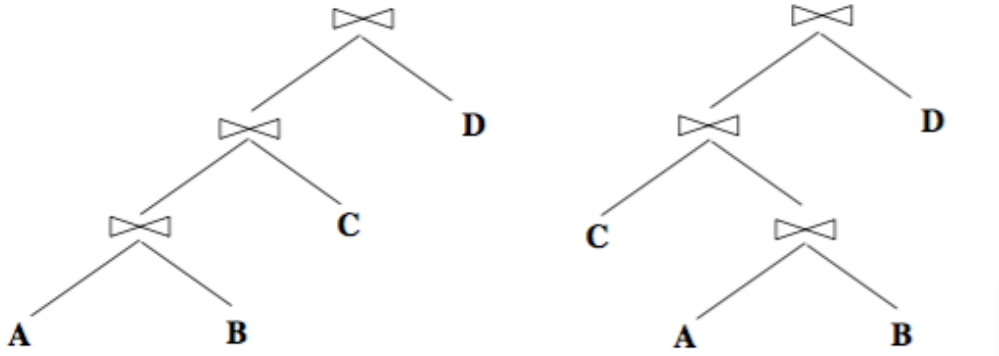
- based on the order in which relations are joined, intermediate relations of widely varying sizes can be created, leading to plans with very different costs

Enumeration of Alternative Plans

Multiple-Relation Queries

Left-Deep Plans

- a query with a natural join of 4 relations: $A \bowtie B \bowtie C \bowtie D$
- relational algebra operator linear trees that are equivalent to this query

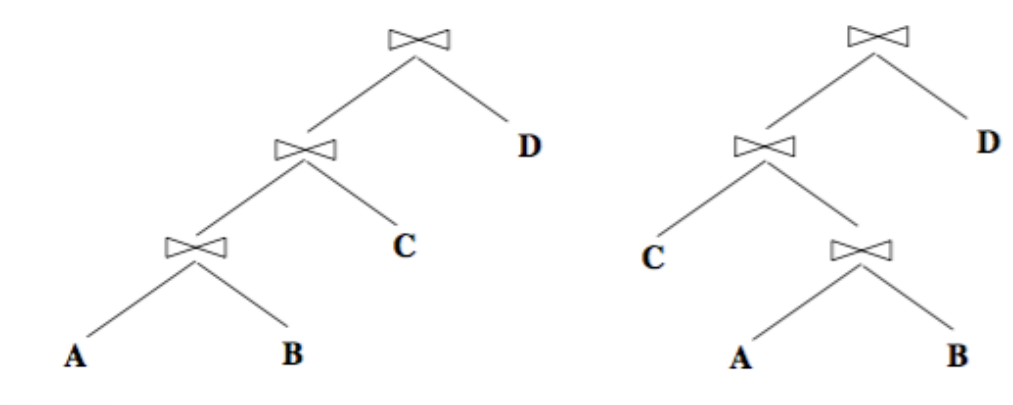


- left child of a join node is the outer relation and the right child is the inner relation
- by adding details such as the join method for each join node we obtain several query evaluation plans from these trees
- equivalence of these trees is based on the relational algebra equivalences

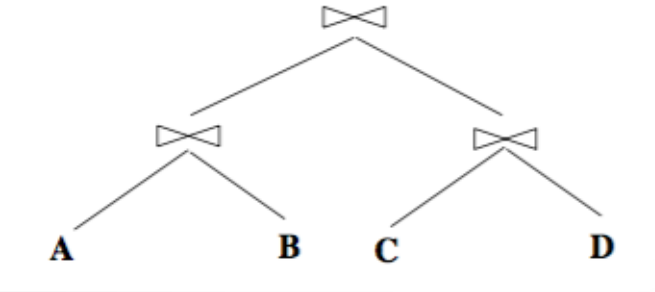
Enumeration of Alternative Plans

Multiple-Relation Queries

- relational algebra operator linear trees



- bushy trees are non linear



Enumeration of Alternative Plans

Multiple-Relation Queries

Left-Deep Plans

1. As the number of joins increases, the number of alternative plans increases rapidly and some pruning of the space of alternative plans becomes necessary
2. Left-deep trees allow us to generate all fully pipelined plans

- Inner relations must always be materialized fully because we must examine the entire inner relation for each tuple of the outer relation
- a plan in which an inner relation is the result of a join forces us to materialize the result of that join - motivates the heuristic decision to consider only left-deep trees.

Enumeration of Alternative Plans

Multiple-Relation Queries

Left-Deep Plans

```
SELECT  attribute list
FROM    relation list
WHERE   term_1 ∧ term_2 ∧ ... ∧ term_n
```

System R style query optimizer enumerates all left-deep plans with selections and projections considered as early as possible

Pass 1: enumerate all single-relation plans (over some relation in the `FROM` clause) - each single-relation plan is a partial left-deep plan for evaluating the query

Pass 2: generate all two-relation plans by considering each single-relation plan that is retained after Pass 1 as the outer relation and (successively) every other relation as the inner relation (tuples generated by the outer plan are assumed to be pipelined into the join)

Enumeration of Alternative Plans

Multiple-Relation Queries

Left-Deep Plans

```
SELECT  attribute list
FROM    relation list
WHERE   term_1 ∧ term_2 ∧ ... ∧ term_n
```

System R style query optimizer enumerates all left-deep plans with selections and projections considered as early as possible

Pass 3: generate all three-relation plans, proceed as in Pass 2, except consider plans retained after Pass 2 as outer relations, instead of plans retained after Pass 1

Additional passes: process is repeated with additional passes until plans are produced that contain all the relations in the query

- We now have the cheapest overall plan for the query

Enumeration of Alternative Plans

Nested Queries

nested queries are dealt with using some form of nested loops evaluation

simple query

Find the names of sailors with the highest rating

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.rating = ( SELECT MAX (S2.rating)
                     FROM   Sailors S2 )
```

- nested sub-query can be evaluated just once, yielding a single value
- value is incorporated into the top-level query as if it had been part of the original statement of the query

Enumeration of Alternative Plans

Nested Queries

nested queries are dealt with using some form of nested loops evaluation

query returning a relation

Find the names of sailors who have reserved boat number 103

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN ( SELECT R.sid
                  FROM   Reserves R
                  WHERE  R.bid = 103 )
```

- nested sub-query can be evaluated just once yielding a collection of sids
- For each tuple of Sailors, check whether the sid value is in the computed collection of sids

Enumeration of Alternative Plans

Nested Queries

nested queries are dealt with using some form of nested loops evaluation

correlated queries

```
SELECT S.sname
FROM   Sailors S
WHERE  EXISTS ( SELECT *
                FROM   Reserves R
                WHERE  R.bid = 103
                       AND S.sid = R.sid )
```

- query is correlated—the tuple variable S from the top-level query appears in the nested sub-query
- For each tuple of Sailors, check whether the sid value is in the computed collection of sids
- we cannot evaluate the sub-query just once

Other Approaches

Complex Queries

rule-based optimizers: use a set of rules to guide the generation of candidate plans

randomized plan generation: uses probabilistic algorithms such as simulated annealing to explore a large space of plans quickly, with a reasonable likelihood of finding a good plan

techniques for estimating the size of intermediate relations more accurately:

- **parametric query optimization**: seeks to find good plans for a given query for each of several different conditions that might be encountered at run-time
- **multiple-query optimization**: the optimizer takes concurrent execution of several queries into account

FAU

SUMMARY

- optimizing SQL queries are first decomposed into small units called **blocks**
- outermost query block is often called **outer block**
- other blocks are called **nested blocks**
- **histogram**: a data structure that approximates a data distribution by dividing the value range into buckets and maintaining summarized information about each bucket
- **equiwidth** histogram: the value range is divided into sub-ranges of equal size
- **equidepth** histogram: the range is divided into sub-ranges such that each sub-range contains the same number of tuples
- generating plans for multiple relations – heuristically only **left-deep plans** are considered
- **nested sub-queries** within queries are usually evaluated using some form of nested loops join