# Query Optimization Overview

- parsing, syntax checking
- semantic checking
  - check existence of referenced relations and attributes
  - disambiguation of overloaded operators
  - check user authorization
- query rewrites
- cost-based optimization
  - consider alternative plans for processing the query
  - select an efficient plan

# Query Rewriting

- rewrites transform a legal query into another, equivalent legal query
- why rewrite?
  - expose optimization opportunities to the cost-based optimizer, e.g., predicate expansion
  - eliminate redundant or unnecessary parts of the query
  - change query structure to suit cost-based optimizer

## Automatically-Generated Queries

Automatic query generators, e.g., object-relational mappers, are one common source of complex queries which may benefit from rewriting.

# Rewrite Example: Subquery to Join

Original query:

```
select ps.* from partsupp ps
where ps.ps_partkey in
    (select p_partkey from parts
     where p_name like 'forest%')
```

Rewritten query:

```
select ps.* from parts, partsupp ps
where ps_partkey = p_partkey and
      p_name LIKE `forest%'
```

## Query Blocks

Cost-based optimizer may optimize each query block (sub-query and outer query separately) in the original. The revised query has only a single block.

This example is from "The DB2 Universal Database Optimizer" by Guy Lohman, IBM Research (2003)

# Query Rewrites in DB2

Original query:

```
select firstnme,lastname from employee e
where not exists
   (select * from employee e2
   where e2.salary > e.salary)
```

Rewritten query:

```
SELECT Q3.$C0 AS "FIRSTNME", Q3.$C1 AS "LASTNAME"
FROM
   (SELECT Q2.FIRSTNME, Q2.LASTNAME, Q1.$RID$
   FROM KMSALEM.EMPLOYEE AS Q1
        RIGHT OUTER JOIN KMSALEM.EMPLOYEE AS Q2 ON
           (Q2.SALARY < Q1.SALARY)) AS Q3
WHERE Q3.$C2 IS NULL
```

# Query Rewrites in DB2 (cont'd)

Original query:

```
select empno from employee e
where exists (select *
              from employee e2
              where e2.salary > e.salary)
```

Rewritten query:

```
SELECT DISTINCT Q2.EMPNO AS "EMPNO"
FROM KMSALEM.EMPLOYEE AS Q1, KMSALEM.EMPLOYEE AS Q2
WHERE (Q2.SALARY < Q1.SALARY)
```

# Rewrite Example: `distinct` Elimination

Original query:

```
select distinct custkey, name
from Customer
```

Revised query:

```
select custkey, name
from Customer
```

This rewrite applies because `custkey` is the key of `Customer`. Thus, `custkey` values will be unique in the result, and the `distinct` is redundant.

This example is from "The DB2 Universal Database Optimizer" by Guy Lohman, IBM Research (2003)

# Rewrite Example: Predicate Translation

- distribution of `NOT`:

  **where** **not** `(col1 = 10` **or** `col2 > 3)`

  `becomes`

  **where** `col1 <> 10 and col2 <= 3`

- constant expression transformation:

  **where** `col = Year('1994-09-08')`

  `becomes`

  **where** `col = 1994`

- predicate transitive closure:

  `given predicates:`

  `T1.c1=T2.c2` **and** `T2.c2=T3.c3` **and** `T1.c1 > 5`

  `add these predicates`

  `T1.c1=T3.c3` **and** `T2.c2 > 5` **and** `T3.c3 > 5`

These examples are from "The DB2 Universal Database Optimizer" by Guy Lohman, IBM Research (2003)

# Other Rewrite Examples

- view merging
- redundant join elimination
- `distinct` pushdown
- predicate pushdown (e.g., into subqueries, views, unions)

# Optimization: Overview

- start with rewritten SQL query
- break the query into query blocks
- generate a plan for each query block (this is the expensive part!)
- generate a plan for the entire query by stiching together the plans for each block

# Query Blocks: Correlated Subquery

```
select empno from employee e
where exists (select *
              from employee e2
              where e2.salary > e.salary)
```

- generate a plan for the inner query (`e2`)
- generate a plan for the outer query (`e`)
- combine by executing the inner query plan once for each employee tuple

# Query Blocks: Uncorrelated Subquery

```
select ProjName
from Project P
where P.DeptNo in
    ( select WorkDept from Employee
      group by WorkDept
      having sum(()salary) > 1000000 )
```

- generate a plan for the inner query (e2)
- generate a plan for the outer query (e)
- combine by executing the inner query plan one time and storing the result

# Logical and Physical Plans

- a logical plan is essentially an algebraic (relational algebra) expression that can be used calculate the result of the query
  - there may be many (logically equivalent) alternative plans for calculating the result of a query
  - the alternatives are defined by a set of algebraic equivalences that are understood by the query optimizer
- a physical plans can be thought of as a refinement of a logical plan, in which
  - logical operations are replaced with DBMS-specific operators that implement those operations (e.g., hybrid hash or merge-sort to implement a join)
  - specific access methods are defined for each relation used in the query
  - additional physical operators (e.g., sorting) are added ensure that computed results (intermediate or final) have desirable physical properties

# Some Algebraic Rules Involving Selection

- $\sigma_{C1 \wedge C2}(R) = \sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C2}(\sigma_{C1}(R)) = \sigma_{C1}(R) \cap \sigma_{C2}(R)$
- $\sigma_{C1 \vee C2}(R) = \sigma_{C1}(R) \cup \sigma_{C2}(R)$

When $C$ involves only attributes of $R$:

- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$

Here, $R$ and $S$ are assumed to have the same schema:

- $\sigma_C(R - S) = \sigma_C(R) - S$
- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

# Pushing Selections Down

Find the last names of the employees responsible for projects in departments managed by employee number '00020'

```
select E.LastName, D.DeptName
from   Employee E, Department D, Project P
where  P.DeptNo = D.DeptNo
and    P.RespEmp = E.EmpNo and D.DeptNo = '00020'
```

The expression

$$\pi_{\mathbf{LastName},\mathbf{DeptName}}(\sigma_{\mathbf{DeptNo}='00020'}(E \bowtie_{RespEmp=EmpNo} (D \bowtie P)))$$

is equivalent to

$$\pi_{\mathbf{LastName},\mathbf{DeptName}}(E \bowtie_{RespEmp=EmpNo} ((\sigma_{\mathbf{DeptNo}='00020'}(D)) \bowtie P))$$

# Pushing Conjunctive Selections

**select** * **from** `Emp_Act A, Project P`
**where** `A.ProjNo = P.ProjNo`
**and** `A.ActNo = 10` **and** `P.DeptNo = 'D01'`

The expression

$$\sigma_{\mathrm{A.ActNo=10} \wedge \mathrm{P.DeptNo='D01'}}(P \bowtie A)$$

can be transformed to

$$\sigma_{\mathrm{A.ActNo=10}}(\sigma_{\mathrm{P.DeptNo='D01'}}(P \bowtie A))$$

With two pushdowns, this becomes

$$(\sigma_{\mathrm{ActNo=10}}(A)) \bowtie (\sigma_{\mathrm{DeptNo='D01'}}(P))$$

# Some Algebraic Rules Involving Projection

- $\pi_M(\pi_N(R)) = \pi_M(R)$

where $M \subseteq N$

- $\pi_{M \cup N}(R \bowtie S) = \pi_{M \cup N}(\pi_{\hat{M}}(R) \bowtie \pi_{\hat{N}}(S))$

where $M$ consists of attributes of $R$, $N$ consists of attributes of $S$ and

- $\hat{M}$ includes the attributes in $M$ plus the join attributes from $R$
- $\hat{N}$ inlcudes the attributes in $N$ plus the join attributes from $S$

# Early Projection

**select** `E.LastName,D.DeptName`
**from** `Employee E, Department D, Project P`
**where** `P.DeptNo = D.DeptNo`
**and** `P.RespEmp = E.EmpNo` **and** `D.DeptNo = '00020'`

Pushing projections transforms this expression

$$\pi_{\mathbf{LastName,DeptName}}(\sigma_{\mathbf{DeptNo='00020'}}(E \bowtie_{RespEmp=EmpNo} (D \bowtie P)))$$

into this expression

$$\pi_{\mathbf{LastName,DeptName}}(\sigma_{\mathbf{DeptNo='00020'}}$$
$$(\pi_{\mathbf{LastName,EmpNo}}(E) \bowtie_{RespEmp=EmpNo}$$
$$(\pi_{\mathbf{DeptNo,DeptName}}(D) \bowtie \pi_{RespEmp,DeptNo}(P))))$$

# Some Algebraic Rules Involving Joins

## Commutativity

- $R \bowtie S = S \bowtie R$

## Associativity

- $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

## Distribution

- $R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$

## Join Ordering

The commutativity and associativity of joins leads to the problem of join ordering, one of the most important issues in query optimization.

# Join Order

The join order may have a significant impact on the cost of a plan. Consider the modified plan:

$$\pi_{\mathbf{LastName}}(E \bowtie_{RespEmp=EmpNo} ((\sigma_{\mathbf{DeptNo='00020'}}(D)) \bowtie P))$$

The joins can be computed, pair-wise, like this:

$$E \bowtie_{RespEmp=EmpNo} (D \bowtie P)$$

or like this:

$$(E \times D) \bowtie_{RespEmp=EmpNo} P$$

or like this:

$$(E \bowtie_{RespEmp=EmpNo} P) \bowtie D$$

# Join Order Example

- Assume that $|E| = 1000$, $|P| = 5000$, and $|D| = 500$. On average, each employee is responsible for five projects.
- If the plan is

$$\sigma_{\text{Lastname}='\text{Smith}'}(E) \bowtie (P \bowtie D))$$

then $P \bowtie D$ must be produced. It has one tuple for each project, i.e., 5000 tuples.

# Join Order Example (cont.)

- If the plan is

$$(\sigma_{\text{Lastname}='\text{Smith}'}(E) \bowtie P) \bowtie D$$

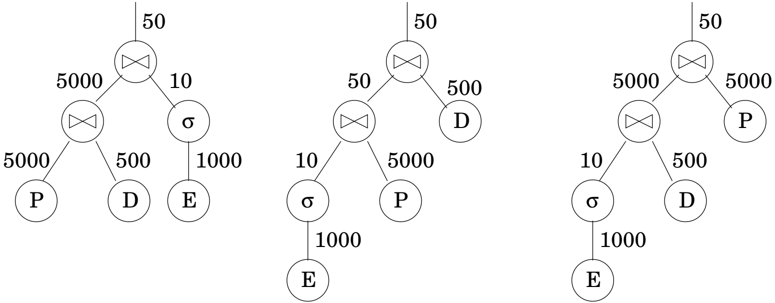
then the intermediate relation has one tuple for each project for which some Smith is responsible. If there are only a few Smith's among the 1,000 employees (say there are 10), this relation will contain about 50 tuples.

- If the plan is

$$(\sigma_{\text{Lastname}='\text{Smith}'}(E) \times D) \bowtie P$$

Since there is no join condition between $E$ and $D$, the intermediate relation will be the cross product $E \times D$. Assuming 10 Smiths, this will have 5000 tuples.

# Join Order Example: Join Sizes



# Join Structure

- The join graph $G_Q$ of a query $Q$ is an undirected graph with
  - one node for each relation in $Q$
  - an edge from $R_1$ to $R_2$ iff there is a join condition linking $R_1$ and $R_2$ in $Q$
- join queries can be classified according to the structure of their join graph, e.g., linear joins, star joins
- optimizers may use the join graph to prune or search the plan space
- another important special case is foreign key joins
- example: a linear query with foreign key joins

**select** `E.LastName`
**from** `Employee E, Department D, Project P`
**where** `P.DeptNo = D.DeptNo`
**and** `P.RespEmp = E.EmpNo` **and** `D.DeptNo = '00020'`

# Cost-Based Optimization: Objective

- goal: find a physical plan with low cost
  - ideally, find an optimal (minimum cost) plan
  - however:
    - cost estimation is imperfect
    - optimization takes time
  - a more modest goal: find a good plan, avoid really bad plans

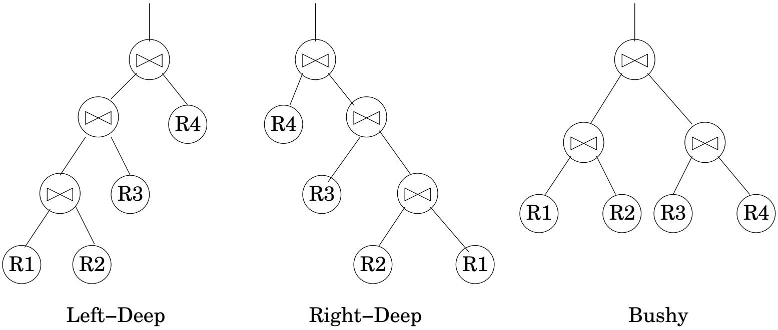
## Control of Optimization

Need to balance optimization effort and plan quality.

# Plan Spaces

- query optimizer must search a space of possible physical plans to identify a good one
- the size of the plan space grows quickly with the number of relations involved in the query block
  - for example: $n!$ left-deep join orders for an $n$-relation query, e.g., 6 join orders for 3-way join, $3x10^6$ join orders for 10-way join, $> 6x10^9$ join orders for 13-way join.
  - this only considers different ways of ordering joins, not, e.g., different realizations of the joins
- many possible ways for an optimizer to explore the plan space, e.g.,
  - bottom-up, dynamic programming
  - branch and bound

# Plan Structure



# A Cost-Based Optimizer Using Dynamic Programming

- we will describe an optimizer that considers only left-deep plans
- key properties of the optimizer
  - bottom-up determination of join order
  - prune high-cost alternatives
  - retain sub-optimal alternatives with "interesting" physical properties that may be useful later
- this optimizer may fail to find an optimal plan in some cases because it does not consider all ways of generating all "interesting" physical properties at all levels

# Dynamic Programming Optimization: Main Idea

```
Select LastName, EmpTime, Projname
From Employee E, Emp_Act A, Project P
Where E.Empno = A.Empno And A.ProjNo = P.ProjNo
And A.EmStDate like '82%' And A.EmpTime >= 0.5
```

- first determine the best way evaluate each single-relation subquery
- next, determine the best way to evaluate each possible two-relation subquery, by joining one more relation to a best single-relation plan
  - plans for $E \bowtie A$, $E \bowtie P$, and $A \bowtie P$
- finally, determine the best way to evaluate the full three-relation query by joining one relation to a best two-relation plan

# Optimization Example: Assumptions

- The query to be optimized consists of a single block with no subqueries or set operations. Otherwise, each block is optimized separately, and the resulting plans are then combined.
- The optimizer pushes selection and projection down as far as possible in each query block.
- Ordering, Grouping and aggregation, if required, are performed last. (No attempt, e.g., to push grouping below joins.)

# Interesting Orders

```
Select LastName, EmpTime, Projname
From Employee E, Emp_Act A, Project P
Where E.Empno = A.Empno And A.ProjNo = P.ProjNo
And A.EmStDate like '82%' And A.EmpTime >= 0.5
```

- suppose the best (cheapest) plan for $E \bowtie A$ produces unordered output, and that there was another plan for $E \bowtie A$ that was more expensive, but produced tuples in `A.ProjNo` output.
- though producing $E \bowtie A$ in `A.ProjNo` order is more expensive, it may be the best way to produce $E \bowtie A$ because it may allow $P$ to be joined in cheaply, using a merge join
- `A.ProjNo` is said to be an interesting order for $E \bowtie A$, since `A.ProjNo` may be useful in processing the rest of the full query (other than $E \bowtie A$)

# Interesting Orders and Pruning

- When determining the best plan for $E \bowtie A$, the DP optimizer behaves as follows:
  - generate and check the cost of each possible plan for $E \bowtie A$ using previously-computed single-relation plans
  - of these possible plans, keep:
    - the cheapest overall plan for $E \bowtie A$
    - for each interesting generated order, the cheapest $E \bowtie A$ plan that produces output in that order
  - all other $E \bowtie A$ plans are pruned (discarded) (why??)

# Dynamic Programming Optimization Example

```
Select LastName, EmpTime, Projname
From Employee E, Emp_Act A, Project P
Where E.Empno = A.Empno And A.ProjNo = P.ProjNo
And A.EmStDate like '82%' And A.EmpTime >= 0.5
```

Available Access Methods:

| | | |
|---|---|---|
| **EI1**: | clustered Btree on E.Empno | relevant |
| **EI2**: | table scan of E | relevant |
| **PI1**: | clustered Btree on P.Projno | relevant |
| **PI2**: | table scan of P | relevant |
| **AI1**: | clustered Btree on A.(Actno,Projno) | not relevant |
| **AI2**: | unclustered Btree on A.EmStDate | relevant |
| **AI3**: | unclustered Btree on A.Empno | relevant |
| **AI4**: | table scan of A | relevant |

# Dynamic Programming Optimization Example (cont'd)

- first iteration: choose the best plan(s) for generating the required tuples from each single relation
  - $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}(A)$
  - $E$
  - $P$
- to choose plans for generating tuples from a relation, consider the available access methods for that relation

# Dynamic Programming Optimization Example (cont'd)

Choose plan(s) for $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}(A)$

1. Some possible plans:

   **A1**: table scan (AI4), then $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}$

   **A2**: index scan (AI2) tuples with EmStDate like '82%', then $\sigma_{EmpTime \geq 0.5}$

   **A3**: index scan (AI3) all tuples, then $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}$

2. Estimate costs of possible plans:
   - suppose that $cost(A2) < cost(A1) < cost(A3)$.

3. Prune plans:

   **A1**: PRUNE!

   **A2**: keep (lowest cost)

   **A3**: keep (more costly than A2, but generates tuples in an *interesting* order - Empno order)

# Dynamic Programming Optimization Example (cont'd)

Choose plan(s) for $E$

1. Generate possible plans:
   - **E1**: table scan (EI2)
   - **E2**: index scan (EI1)
2. Estimate cost of possible plans:
   - suppose that $cost(E1) < cost(E2)$.
3. Prune plans:
   - **E1**: keep (lowest cost)
   - **E2**: keep (more costly than E1, but generates tuples in Empno order)

# Dynamic Programming Optimization Example (cont'd)

Choose plan(s) for $P$

1. Generate possible plans:

   **P1**: table scan (PI2)
   **P2**: index scan (PI1)

2. Estimate cost of possible plans:
   - suppose that $cost(P1) < cost(P2)$.

3. Prune plans:

   **P1**: keep (lowest cost)
   **P2**: keep (more costly than P1, but generates tuples in Projno order)

# Dynamic Programming Optimization Example (cont'd)

- second iteration: choose the best plan(s) for generating the required tuples from each pair of relations
  - $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie E$
  - $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie P$
  - $E \bowtie P$
- to build plans for generating tuples from $n$ relations:
  - choose a join type
  - choose an unpruned plan for $n - 1$ relations as the outer input to the join
  - choose an access method for the remaining relation as the inner input to the join

# Dynamic Programming Optimization Example (cont'd)

Choose plan(s) for $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie E$

1. Generate possible plans (not all shown)

   **AE1**: nested loop join A2 and EI2
   **AE2**: index nested loop join A2 and EI1
   **AE3**: merge join sort(A2) and EI1 (Empno order)
   **AE4**: nested loop join A3 and EI1 (Empno order)
   **AE5**: merge join A3 and EI1 (Empno order)
   **EA1**: nested loop join E1 and AI2
   **EA2**: merge join E2 and sort(AI4) (Empno order)
   **EA3**: index nested loop join E2 and AI3 (Empno order)

2. Estimate costs of possible plans and prune:
   - suppose that $cost(AE1)$ is the cheapest overall. Prune all but AE1.

# Dynamic Programming Optimization Example (cont'd)

Choose plan(s) for $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie P$

1. Generate possible plans (not all shown)
   - **AP1**: nested loop join A3 and sort(PI2) (Empno order)
   - **AP2**: merge join sort(A2) and PI1 (Projno order)
   - **AP3**: index nested loop join A3 and PI1 (Empno order)
   - **AP4**: index nested loop join A2 and PI1
   - **PA1**: index nested loop join P2 and AI3 (Projno order)
2. Estimate costs of possible plans and prune:
   - suppose that AP2 is cheapest overall, and that $cost(AP3) < cost(AP1)$
   - keep only AP2 (cheapest) and AP3 (more expensive, but Empno order is interesting)

# Dynamic Programming Optimization Example (cont'd)

Choose plan(s) for $P \bowtie E$

1. Generate possible plans (not all shown)

   **PE1**: nested loop join of P2 and EI2 (Projno order)

   **EP1**: nested loop join of E2 and PI2 (Empno order)

2. Estimate costs of possible plans and prune:
   - suppose that PE2 is cheapest overall, and EP1 is the cheapest plan producing Empno order
   - keep PE2 (cheapest) and PE1 (interesting order)

# Dynamic Programming Optimization Example (cont'd)

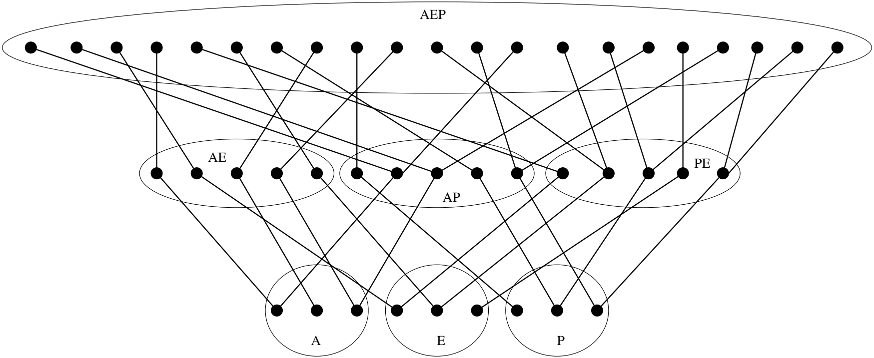
- third iteration: choose the best plan(s) for generating the required tuples from all three relations
  - $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie E \bowtie P$
- consider
  - the best plans for $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie E$ combined with an access method for $P$
  - the best plans for $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie P$ combined with an access method for $E$
  - the best plans for $E \bowtie P$, combined with an access method for $A$

# Dynamic Programming Optimization Example (cont'd)

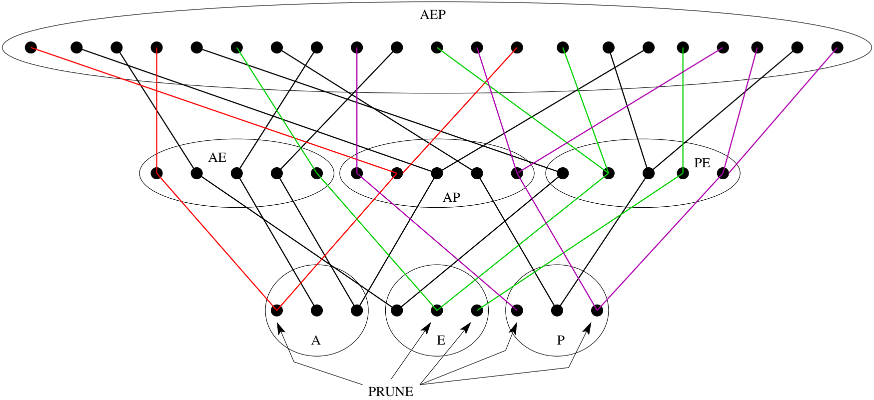
Choose plan(s) for $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}(A) \bowtie E \bowtie P$

1. Generate possible plans (not all shown)
   - **AEP1**: index nested loop join AE1 and PI1
   - **AEP2**: nested loop join AE1 and PI2
   - **APE1**: index nested loop join AP2 and EI1
   - **APE2**: merge join AP3 and EI1 (Empno order)
   - **PEA1**: index nested loop join PE1 and AI3 (Projno order)
   - **PEA2**: merge join PE2 and AI3 (Empno order)
2. Estimate costs of possible plans and prune:
   - suppose that AEP1 is cheapest
   - since there are no more relations to be joined and there are no GROUP BY or ORDER BY clauses in the query, there is no need to further preserve interesting orders.
   - prune all plans except the winner: AEP1

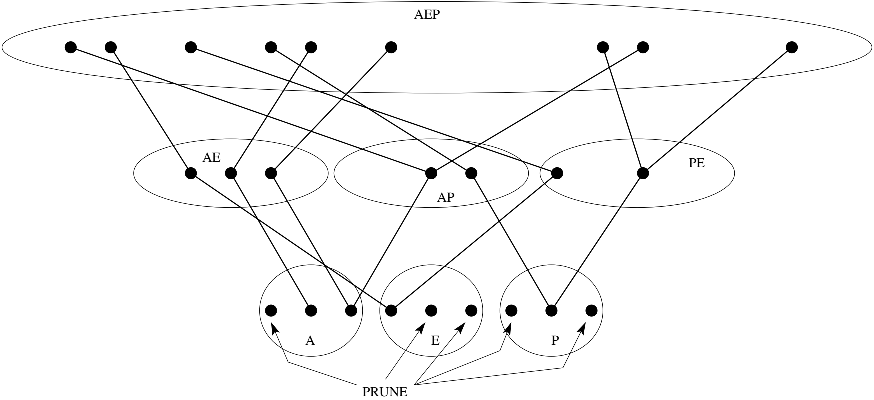
# Effects of Pruning



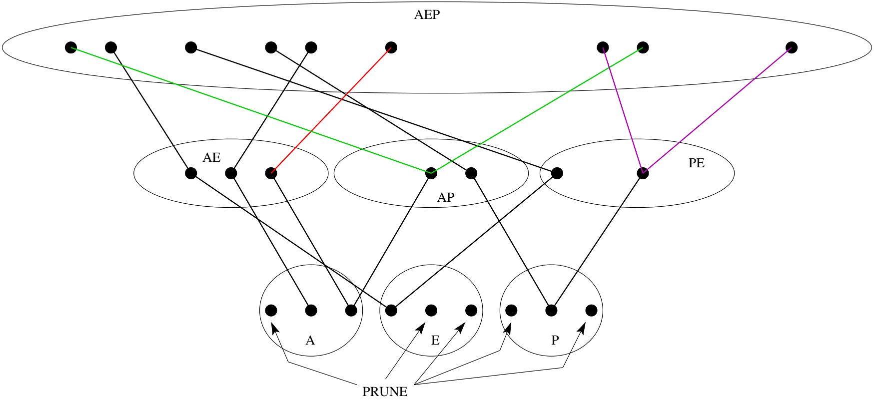
# Effects of Pruning (cont'd)



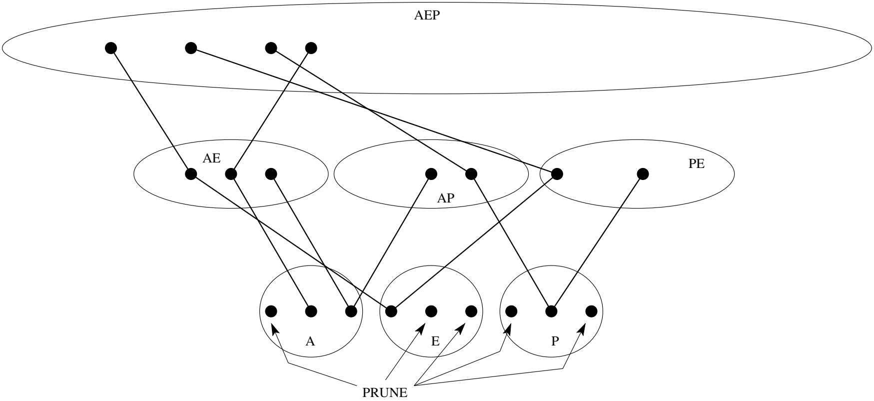
# Effects of Pruning (cont'd)



# Effects of Pruning (cont'd)



# Effects of Pruning (cont'd)



# Cost Models

- An optimizer estimates costs for plans so that it can choose the least expensive plan from a set of alternatives.
- Inputs to the cost model include:
  - the query
  - database statistics
  - description of computational resources, e.g.
    - CPU speed
    - costs of sequential and random disk operations
    - size of buffer pool, amount of memory available for query operators
  - concurrency environment, e.g., number of concurrent queries
  - system configuation parameters

# What is Cost??

- a cost model assigns a number (the cost) to each query, but what does that number represent?
- some possibilities:
  - query response time
  - total computing resource consumption for query execution
  - dollar cost of executing the query
- a common approach is to use total resource consumption:

  `cost(Q)=CPUCost(Q)+DiskCost(Q)+NetworkCost(Q)`

  where `CPUCost(Q)` is an estimate of the total CPU time required to execute the query, `DiskCost(Q)` is an estimate of the total time required for all disk I/O operations for the query, and so on.

# Costing a Plan

- first estimate the cost of leaf operators (access methods) in the plan, and the size of their output
  - estimates require database statistics and selectivity estimation for any predicate implemented by the access method
  - estimating the number of tuples in the result of an operator is called cardinality estimation
- do cost and cardinality estimation for non-leaf operators once estimates of their input cardinality are available

# Costing Access Methods

- Consider access to the relation $A$ in the optimization example: $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}(A)$
- these three access methods were possible:

**Method A1**: table scan, then $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}$

**Method A2**: index scan using unclustered index on A.EmStDate, then $\sigma_{EmpTime \geq 0.5}$

**Method A3**: index scan using unclustered index on A.EmpNo, then $\sigma_{EmStDate\ \mathrm{like}\ '82\%' \wedge EmpTime \geq 0.5}$

- To estimate the costs of these methods, the optimizer needs to answer some basic questions, e.g.:
  - how many tuples in $A$? How many blocks?
  - how large are the indexes? How many leaves? How deep?
  - how many of the tuples will satisfy the conditions?

# Database Statistics

- To support costing, the DBMS maintains basic statistics about the database in its catalog. For example:
  - number of rows and tuples in each table
  - number of key values, levels, leaf pages in each index
- In addition, to help answer questions such as "how many tuples have `EmpStDate like '82%'`, the DBMS maintains information about the values in some or all of the columns of the table. For example:
  - number of distinct values
  - minimum and maximum values
  - quantiles, histograms or similar structures describing the distribution of different values for that column

## Updates

The DBMS must have some means of maintaining these statistics as the underlying database is updated.

# Database Statistics in DB2

```
db2 "select colname,colcard,high2key,low2key
   from sysstat.columns where tabname = 'EMPLOYEE'"

BIRTHDATE  30 '1955-04-12' '1926-05-17'
BONUS       8 +0000900.00    +0000400.00
COMM       32 +0003720.00    +0001272.00
EDLEVEL     8 19             14
EMPNO      32 '000330'       '000020'
FIRSTNME   30 'WILLIAM'      'CHRISTINE'
HIREDATE   31 '1980-06-19'   '1949-08-17'
JOB         8 'PRES    '     'CLERK   '
LASTNAME   31 'WALKER'       'BROWN'
MIDINIT    20 'W'            'A'
PHONENO    32 '9001'         '0942'
SALARY     32 +0046500.00    +0015900.00
SEX         2 'M'            'F'
WORKDEPT    8 'E11'          'B01'
```

# Selectivity Estimation

- an important problem for the optimizer is estimating the selectivity of query predicates
- the filter factor of a predicate $C$ applied to relation $R$ is the fraction of $R$'s tuples that satisfy $C$

$$\frac{|\sigma_C(R)|}{|R|}$$

- filter factors (selectivity) can be estimated
  - using basic statistics about the columns in $C$
  - using histograms on the columns in $C$
  - using sampling

# Selectivity Estimation

If no other information is available, selectivity can be estimated using basic column statistics, e.g.:

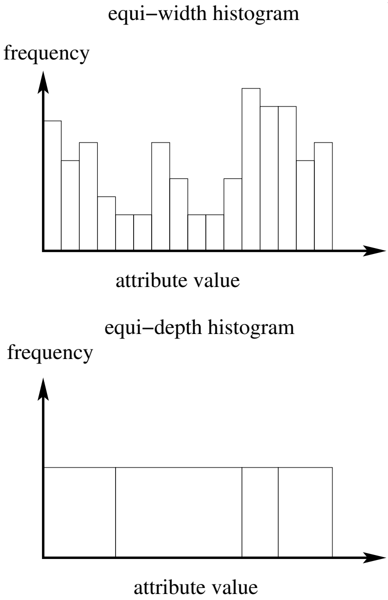
- $|\sigma_{R.a=c}(R)| \approx \frac{1}{distinct(R.a)}$
- $|\sigma_{R.a\leq c}(R)| \approx \frac{c-min(R.a)}{max(R.a)-min(R.a)}$
- $|\sigma_{C1\wedge C2}(R)| \approx |\sigma_{C1}(R)| \cdot |\sigma_{C2}(R)|$

These formulas are based on assumptions, e.g.,

- uniformity assumptions
- independence assumptions

To the extent that these are incorrect, such estimates may be inaccurate.

# Two Basic Types of Histograms




equi-width:

- all buckets ranges are the same width
- store: frequency for each bucket

equi-depth

- all buckets have the same freq
- store: bucket boundaries

# Compressed Histograms

- When there is data skew, it is particularly important to have accurate estimates of the number of occurrences of common values.
- In a compressed histogram, some space is devoted to keeping exact counts for the most frequently occuring values. A regular histogram (e.g., equi-depth) is then used to estimate the frequency of other, less frequent values.

# Histograms in DB2

```
db2 "select seqno,colvalue,valcount from
     sysstat.coldist where tabname = 'EMPLOYEE'
     and colname = 'SALARY'"

SEQNO  COLVALUE       VALCOUNT
1      +0015340.00     1
2      +0015900.00     2
3      +0017750.00     4
4      +0018270.00     5
5      +0019950.00     7
6      +0021340.00     9
7      +0022180.00    10
...
16     +0036170.00    26
17     +0038250.00    27
18     +0040175.00    29
19     +0046500.00    31
20     +0052750.00    32
```

# Complex Predicates

- simple estimation rules or histograms can be used to estimate the selectivity of predicates involving a single attribute
- what about about multi-attribute predicates? For example:

$$C_1 \wedge C_2 \wedge C_3$$

- some possibilities:
  - combine single attribute estimates, e.g., using independence assumption
  - multi-dimensional histograms
  - tuple sampling
- challenge is to obtain quick and accurate estimates using small synopses (histograms or samples)

# Selectivity Estimation via Sampling

- main idea: sample a small set of tuples from a large relation
  - can be done on-demand, when a query is being optimized
  - alternatively, sample can be drawn in advance, stored, and used to estimate selectivity for multiple queries
- to estimate the selectivity of an arbitrary predicate:
  1. measure the number of sample tuples that satisfy the predicate
  2. extrapolate the measurement to the whole relation

# Estimating the Cost of An Access Method

Access method **A2** for $\sigma_{EmStDate \text{ like } '82\%' \wedge EmpTime \geq 0.5}(A)$:

- index scan for tuples with EmStDate like '82%' using unclustered index on EmStDate, then apply predicate EmpTime $\geq$ 0.5

Estimate cost (total CPU + disk I/O time) of **A2**:

- estimate numbers of tuples scanned and selected (selectivity estimation)
- estimate number of disk blocks read, and whether read sequentially or randomly
- for disk I/O time, charge fixed amount per random disk read, and (smaller) fixed amount per sequential disk read
- for CPU time, charge fixed amount per block read, fixed amount per tuple read from index, and fixed amount per output tuple
- total cost is sum of disk I/O time and CPU time.

# Join Size Estimation

- Another important problem is estimating the size of joins.
- We know:

$$0 \leq |R \bowtie S| \leq |R| \cdot |S|$$

- The ratio

$$\frac{|R \bowtie S|}{|R| \cdot |S|}$$

  is sometimes called the join selectivity
- Several techniques may be used for join size estimation, sometimes in combination:
  - exploit schema information, e.g., for foreign key joins (a common case)
  - exploit histograms if available
  - estimate using simple statistics only

# Join Size Estimation (cont'd)

foreign key joins: Consider $P \bowtie_{(RespEmp=EmpNo)} E$. Since EmpNo is the key of $E$, the join size may be estimated as $|P|$.

using histograms: If histograms are available on the join keys, they can be used to upper-bound the join selectivity. For example, in $R \bowtie_{R.a=S.b} S$, each $t \in R$ can on join with tuples from the $S.b$ histogram bucket into which $t.a$ would fall.

using simple statistics: One way to estimate the join selectivity of $R \bowtie_{R.a=S.b} S$ is

$$min\left(\frac{1}{\mathrm{distinct}(R.a)}, \frac{1}{\mathrm{distinct}(S.b)}\right)$$

# Estimating Plan Cost (DB2 Example)

```
           0.639622
              SORT
            66.5245
            2.63962
               |
           0.639622 <-- estimated rows
            NLJOIN
            66.5166 <-- cumulative cost
            2.63962 <-- cumulative I/Os
       /-------+-------\
  3.2                    0.199882
TBSCAN                    FETCH
50.3429                   5.11925
   2                     0.199882
   |                     /----+---\
```

- estimate plan cost by estimating costs of plan operators
- need properties (e.g., size distribution) of intermediate results to estimate costs of non-leaf operators