Database Management Systems

Query Optimization

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
and
Centre for Artificial Intelligence and Machine Learning
Indian Statistical Institute, Kolkata

April, 2022

Basics of query optimization

Why optimizing a query?
So that it is processed efficiently.

Basics of query optimization

Why optimizing a query?
So that it is processed efficiently.

What is meant by efficiently?
Minimizing the cost of query evaluation.

Basics of query optimization

Why optimizing a query?
So that it is processed efficiently.

What is meant by efficiently?
Minimizing the cost of query evaluation.

Who will minimize?
The system, not the user.

Basics of query optimization

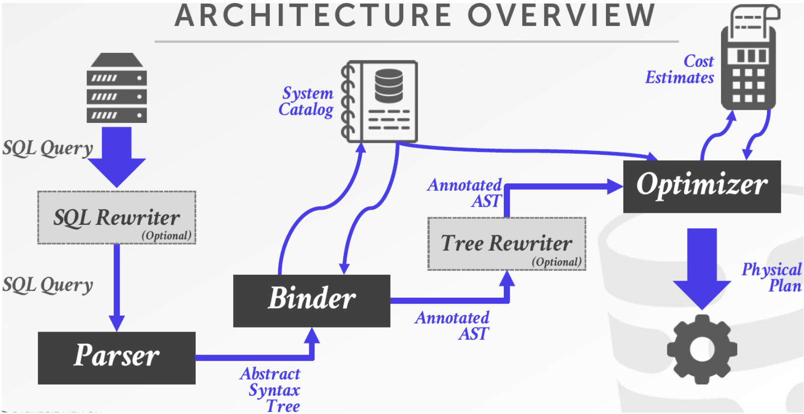
Why optimizing a query?
So that it is processed efficiently.

What is meant by efficiently?
Minimizing the cost of query evaluation.

Who will minimize?
The system, not the user.

Query optimization is facilitating a system to construct a query-evaluation plan for processing a query efficiently, without expecting users to write efficient queries

How does a query optimizer work?

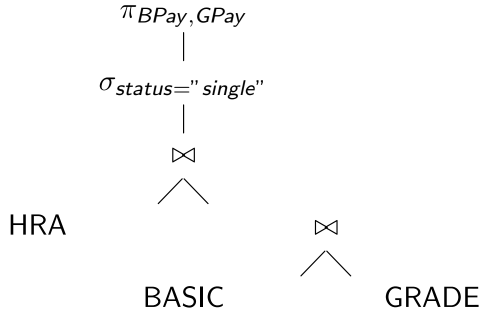


An example query

Find the basic pay and grade pay of ISI employees who are single.

An example query

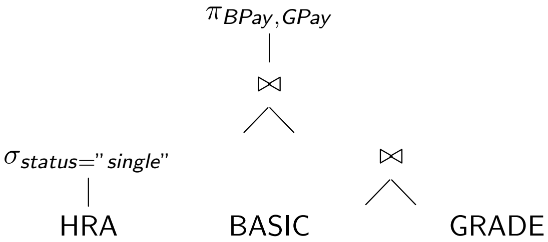
Find the basic pay and grade pay of ISI employees who are single.



Note: Marital status is a part of HRA table.

An example query – revised

Find the basic pay and grade pay of ISI employees who are single.



Note: Marital status is a part of HRA table.

Evaluating query cost

The basic parameters for estimating the query cost are

- The number of seek operations performed
- The number of blocks read
- The number of blocks written

Evaluating query cost

The basic parameters for estimating the query cost are

- The number of seek operations performed
- The number of blocks read
- The number of blocks written

Intuitive (Naive) approach for least-cost query finding:

1. Generate expressions that are logically equivalent to the original expression
2. Annotate the resultant expressions in alternative ways to generate alternative query evaluation plans.
3. Go to 1 until you get some new expression.

Catalog information

For a given relation R, we can store the following relevant information in the catalog:

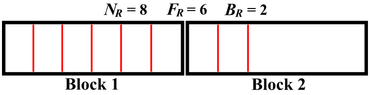
- N_R – the number of tuples
- B_R – the number of blocks containing tuples of relation R
- L_R – the size of a tuple in bytes
- F_R – the number of tuples that fit into one block (blocking factor)
- $V(X, R)$ – the number of distinct values for attribute X
- H_R – the height of B$^+$-tree indices for R
- P_R – the number of leaf pages in the B$^+$-tree indices for R

Note: $V(X, R)$ equals to the size of $\pi_X(R)$, in general, and if X is a key then it is N_R.

Other statistical information

Suppose the tuples of a relation R are physically stored in a file then we have the following relation

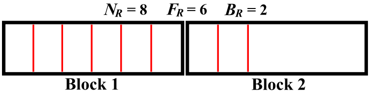
$$B_R = \left\lceil \frac{N_R}{F_R} \right\rceil$$



Other statistical information

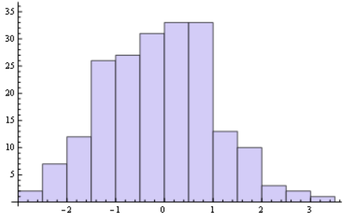
Suppose the tuples of a relation R are physically stored in a file then we have the following relation

$$B_R = \left\lceil \frac{N_R}{F_R} \right\rceil$$



Special statistical information:

Histogram –



Final comments

Some facts about the relation statistics:

– Recompute relation statistics on every update (but this might be a huge overhead), at least during the periods of light system load.

– In real-world cases, optimizers often maintain further statistical information to improve the accuracy of their cost estimates of evaluation plans.

Size estimation for selection operation

Assumption: Attribute values are uniformly distributed

- $\mathcal{S}(\sigma_{X=x}(R))$: $\frac{N_R}{V(X,R)}$
- $\mathcal{S}(\sigma_{X\leq x}(R))$: $\frac{N_R*(x-\min(X,R)+1)}{\max(X,R)-\min(X,R)+1}$, where $\min(X,R)$ and $\max(X,R)$ denote the minimum and maximum values of the attribute X in R, respectively
- $\mathcal{S}(\sigma_{\theta_1\wedge\theta_2\wedge\ldots\wedge\theta_n}(R))$: $\frac{\mathcal{S}_1*\mathcal{S}_2*\ldots*\mathcal{S}_n}{N_R^{n-1}}$, where $\mathcal{S}_i$ denotes the estimated size of the selection operation $\sigma_{\theta_i}(R)$
- $\mathcal{S}(\sigma_{\theta_1\vee\theta_2\vee\ldots\vee\theta_n}(R))$: $\frac{N_R^n-(N_R-\mathcal{S}_1)*(N_R-\mathcal{S}_2)*\ldots*(N_R-\mathcal{S}_n)}{N_R^{n-1}}$, where $\mathcal{S}_i$ denotes the estimated size of the selection operation $\sigma_{\theta_i}(R)$
- $\mathcal{S}(\sigma_{\neg\theta}(R))$: $N_R - \mathcal{S}(\sigma_\theta(R))$

Note: $\mathcal{S}_i$ denotes $\mathcal{S}(\sigma_{\theta_i})$, where θ_i represents a predicate.

Size estimation for selection operation

The estimation of size does not work well in the following cases:

- Values of the attribute on which the selection is applied are not uniformly distributed.
- The number of tuples is pretty low.
- The number of distinct values for an attribute is pretty low.
- The predicates connected together (by logical AND or OR) under the selection are dependent on each other.

Size estimation for selection – A conceptual example

Consider the following table and its system catalog information:

Table: TOY

ID	COLOR	COST
T1	Blue	10
T2	Blue	10
T3	Red	20
T4	Blue	20
T5	Red	30
T6	Red	30

$N_{TOY} = 6$
$V(COLOR, TOY) = 2$
$V(COST, TOY) = 3$
$\min(COST, TOY) = 10$
$\max(COST, TOY) = 30$
$H_{TOY} = 3$

- $\mathcal{S}(\sigma_{COLOR=Red}(TOY))$: $\frac{N_{TOY}}{V(COLOR,TOY)} = 3.$
- $\mathcal{S}(\sigma_{COST \leq 20}(TOY))$: $\frac{N_{TOY} * (20 - \min(COST,TOY) + 1)}{\max(COST,TOY) - \min(COST,TOY) + 1} \sim 3.14.$
- $\mathcal{S}(\sigma_{(COLOR=Red) \wedge (COST \leq 20)}(TOY))$: $\frac{\mathcal{S}(\sigma_{COLOR=Red}(TOY)) * \mathcal{S}(\sigma_{COST \leq 20}(TOY))}{N_{TOY}} \sim 1.57$

Size estimation for Cartesian product and natural join

Cartesian product:

$\mathcal{S}(R_1 \times R_2)$ is equal to $N_{R_1} * N_{R_2}$ (each tuple occupies $L_{R_1} + L_{R_2}$ bytes).

Natural join:

- If $A(R_1) \cap A(R_2) = \phi$: $\mathcal{S}(R_1 \bowtie R_2)$ equals to $N_{R_1} * N_{R_2}$
- If $A(R_1) \cap A(R_2)$ is a key for R_1: $\mathcal{S}(R_1 \bowtie R_2)$ is no greater than N_{R_2}
- If $A(R_1) \cap A(R_2)$ is a key for R_2: $\mathcal{S}(R_1 \bowtie R_2)$ is no greater than N_{R_1}
- If $A(R_1) \cap A(R_2)$ is a key for neither R_1 nor R_2: $\mathcal{S}(R_1 \bowtie R_2)$ is the maximum of $\frac{N_{R_1} * N_{R_2}}{V(X, R_1)}$ and $\frac{N_{R_1} * N_{R_2}}{V(X, R_2)}$

Size estimation for other operations

- Projection: $\mathcal{S}(\pi_X(R))$ equals to $V(X, R)$
- Aggregation: Involves a size of $V(X, R)$
- Set union operation: $\mathcal{S}(R_1 \cup R_2)$ is no greater than $N_{R_1} + N_{R_2}$
- Set intersection operation: $\mathcal{S}(R_1 \cap R_2)$ is no greater than $\min(N_{R_1}, N_{R_2})$
- Set difference operation: $\mathcal{S}(R_1 - R_2)$ is no greater than N_{R_1}

Query size estimation – Example I

Given a relation R with 60 tuples. If R has an attribute *Age* within the range [20, 30] and there are 15 distinct values for the attribute *Height* minimum of which is 170, estimate the size of the query $\sigma_{(Age \leq 23) \vee Height = 170}(R)$.

Query size estimation – Example I

Given a relation R with 60 tuples. If R has an attribute Age within the range [20, 30] and there are 15 distinct values for the attribute $Height$ minimum of which is 170, estimate the size of the query $\sigma_{(Age \leq 23) \vee Height=170}(R)$.

Solution: It is given that $N_R = 60$, $\min(Age, R) = 20$, $\max(Age, R) = 30$, $\min(Height, R) = 170$ and $V(Height, R) = 15$. Therefore, with uniform distribution assumption, the size of the query can be estimated as $\mathcal{S}(\sigma_{(Age \leq 23) \vee Height=170}(R))$

$$= \frac{N_R^2 - (N_R - \mathcal{S}(\sigma_{Age \leq 23}(R))) * (N_R - \mathcal{S}(\sigma_{Height=170}(R)))}{N_R}$$

$$= \frac{N_R^2 - (N_R - \frac{N_R*(23-\min(Age,R))}{\max(Age,R)-\min(Age,R)}) * (N_R - \frac{N_R}{V(Height,R)})}{N_R}$$

$$= 20.8.$$

Query size estimation – Example II

Let $R1(ID, Name)$ **and** $R2(Roll, CGPA)$ **be a pair of relations. Now if** ID **be the primary key for** $R1$ **and the attribute** $Roll$ **has a minimum value of 118002001, then estimate the size of the query** $\sigma_{ID=11}(R1) \bowtie \sigma_{Roll \leq 118002000}(R2)$**.**

Query size estimation – Example II

Let $R1(ID, Name)$ and $R2(Roll, CGPA)$ be a pair of relations. Now if ID be the primary key for $R1$ and the attribute $Roll$ has a minimum value of 118002001, then estimate the size of the query $\sigma_{ID=11}(R1) \bowtie \sigma_{Roll \leq 118002000}(R2)$.

Solution: If ID be the primary key of $R1$, then it should have distinct values satisfying $V(ID, R1) = N_{R1}$. Again it is given that $\min(Roll, R2) = 118002001$. Therefore, with the assumption of uniform distribution over the attribute domains, the given query size can be estimated as $\mathcal{S}(\sigma_{ID=11}(R1) \bowtie \sigma_{Roll \leq 118002000}(R2))$

$$
\begin{aligned}
&= \mathcal{S}(\sigma_{ID=11}(R1)) \times \mathcal{S}(\sigma_{Roll \leq 118002001}(R2)) \\
&= \frac{N_{R1}}{V(ID, R1)} \times \frac{N_{R2} * (118002000 - \min(Roll, R2) + 1)}{\max(Roll, R2) - \min(Roll, R2) + 1} \\
&= \frac{N_{R1}}{N_{R1}} \times \frac{N_{R1} * (118002000 - 118002001 + 1)}{\max(Roll, R2) - 118002001} = 0.
\end{aligned}
$$

Query size estimation – Example III

Given a pair of relations $R1$ and $R2$, wherein the primary key of $R1$ (say K) is the only foreign key of $R2$, estimate the size of the following queries. Assume that the number of tuples in $R1$ and $R2$ are $t1$ and $t2$, respectively.

i) $R1 \bowtie R2$.

ii) $R1 \times R2$.

iii) $\sigma_{K=\text{'}19BM6JP01'}(R1) \times R2$.

Query size estimation – Example III

Given a pair of relations $R1$ and $R2$, wherein the primary key of $R1$ (say K) is the only foreign key of $R2$, estimate the size of the following queries. Assume that the number of tuples in $R1$ and $R2$ are $t1$ and $t2$, respectively.

i) $R1 \bowtie R2$.

ii) $R1 \times R2$.

iii) $\sigma_{K=\text{'}19BM6JP01\text{'}}(R1) \times R2$.

Solution:

i) No greater than t_2.

ii) $t_1 t_2$.

iii) t_2.

Query size estimation – Example IV

Consider a pair of relations $R1(Year, Venue, Host)$ **and** $R2(RegistrationID, Host)$**, where** `Year` **and** `RegistrationID` **are the primary keys for** $R1$ **and** $R2$**, respectively. Estimate the size of the following query:**

$$\sigma_{Year=2018}(R1) \bowtie \sigma_{RegistrationID=OLYMPIC1234}(R2).$$

Query size estimation – Example IV

Consider a pair of relations $R1(Year, Venue, Host)$ **and** $R2(RegistrationID, Host)$**, where** `Year` **and** `RegistrationID` **are the primary keys for** $R1$ **and** $R2$**, respectively. Estimate the size of the following query:**

$$\sigma_{Year=2018}(R1) \bowtie \sigma_{RegistrationID=OLYMPIC1234}(R2).$$

Solution: The size will be $1 \times 1 = 1$.

What is query equivalence?

Two relational algebra expressions are said to be *equivalent* if on every legal database instance (i.e., a relation) the two expressions generate the same relation (i.e., the same set of tuples).

Query equivalence relations are used for tuning a query into an optimized form.

Query equivalence – On projection and selection

- Cascade property of projection: $\pi_{X_1}(\pi_{X_2}(\ldots(\pi_{X_n}(R))\ldots)) \equiv \pi_{X_1}(R)$. Note that, the attribute set $X_1 \subseteq X_2 \subseteq \ldots \subseteq X_n$.
- Cascade property of selection: $\sigma_{\theta_1 \wedge \theta_2}(R) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(R))$.
- Commutative property of selection: $\sigma_{\theta_1}(\sigma_{\theta_2}(R)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(R))$.
- Selection can be combined with Cartesian product and theta join in the following way:
    1. $\sigma_{\theta}(R_1 \times R_2) \equiv R_1 \bowtie_{\theta} R_2$ (this is simply the definition of theta join).
    2. $\sigma_{\theta_1}(R_1 \bowtie_{\theta_2} R_2) \equiv R_1 \bowtie_{\theta_1 \wedge \theta_2} R_2$.

Query equivalence – An observation

Suppose the following queries are applied on a relation $R(X, Y, Z)$.

$$Q1 : \sigma_{\theta_1}(\sigma_{\theta_2}(R))$$

$$Q2 : \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

Let the predicates θ_1 and θ_2 perform equality check (let for a single value) on the attributes X and Z, respectively. Compare the efficiencies of $Q1$ and $Q2$ if it is mentioned that $V(X, R) > V(Y, R) > V(Z, R)$.

Query equivalence – An observation

Suppose the following queries are applied on a relation $R(X, Y, Z)$.

$$Q1 : \sigma_{\theta_1}(\sigma_{\theta_2}(R))$$

$$Q2 : \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

Let the predicates θ_1 and θ_2 perform equality check (let for a single value) on the attributes X and Z, respectively. Compare the efficiencies of $Q1$ and $Q2$ if it is mentioned that $V(X, R) > V(Y, R) > V(Z, R)$.

Solution: $Q2$ is better because it first selects on the attribute X, having more distinct entries, returning lesser number of tuples. Note that, all the attributes in R have same number of values (corresponding to the tuples).

Query equivalence – On theta-join

- Commutative property of theta-join: $R_1 \bowtie_\theta R_2 \equiv R_2 \bowtie_\theta R_1$.
- Theta joins are associative in the following way: $(R_1 \bowtie_{\theta_1} R_2) \bowtie_{\theta_2 \wedge \theta_3} R_3 \equiv R_1 \bowtie_{\theta_1 \wedge \theta_3} (R_2 \bowtie_{\theta_2} R_3)$, where θ_2 involves attributes only from R_2 and R_3. Any of these conditions may be empty, and hence it follows that the Cartesian product operation is also associative.
- The selection operation distributes over the theta-join operation under the following two conditions:
  1. It distributes when all the attributes in selection condition θ_0 involve only the attributes of one of the relations (say R_1) being joined. i.e. $\sigma_{\theta_0}(R_1 \bowtie_\theta R_2) \equiv (\sigma_{\theta_0}(R_1) \bowtie_\theta R_2)$.
  2. It distributes when selection condition θ_1 involves only the attributes of R_1 and θ_2 involves only the attributes of R_2. i.e. $\sigma_{\theta_1 \wedge \theta_2}(R_1 \bowtie_\theta R_2) \equiv (\sigma_{\theta_1}(R_1) \bowtie_\theta \sigma_{\theta_2}(R_2))$.

Query equivalence – On theta-join

- The projection operation distributes over the theta-join operation under the following two conditions:
  1. Let L_1 and L_2 be the attributes of R_1 and R_2, respectively, and the join condition θ involves the attributes only in $L_1 \cup L_2$. Then we have

$$\pi_{L_1 \cup L_2}(R_1 \bowtie_\theta R_2) \equiv (\pi_{L_1}(R_1)) \bowtie_\theta (\pi_{L_2}(R_2)).$$

  2. Consider a join operation $R_1 \bowtie_\theta R_2$ and suppose L_1 and L_2 be the sets of attributes from R_1 and R_2, respectively. Further assume that L_3 and L_4 denote the attributes of R_1 and R_2, respectively, that are involved in join condition θ, but are not in $L_1 \cup L_2$. Then we have

$$\pi_{L_1 \cup L_2}(R_1 \bowtie_\theta R_2) \equiv \pi_{L_1 \cup L_2}((\pi_{L_1 \cup L_3}(R_1)) \bowtie_\theta (\pi_{L_2 \cup L_4}(R_2))).$$

Query equivalence – On natural join

- Commutative property of natural join: $R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$.
- Associative property of natural join: $(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$.

Note: The commutativity and associativity of join operations are important for join reordering in query optimization.

Query equivalence – On set operations

- Commutative property of set union: $R_1 \cup R_2 \equiv R_2 \cup R_1$.
- Associative property of set union: $(R_1 \cup R_2) \cup R_3 \equiv R_1 \cup (R_2 \cup R_3)$.
- Commutative property of set intersection: $R_1 \cap R_2 \equiv R_2 \cap R_1$.
- Associative property of set intersection: $(R_1 \cap R_2) \cap R_3 \equiv R_1 \cap (R_2 \cap R_3)$.
- The selection operation distributes over the union, intersection and set difference operations: $\sigma_\theta(R_1 - R_2) \equiv \sigma_\theta(R_1) - \sigma_\theta(R_2)$ (replacing '$-$' with either $\cup$ or $\cap$ also holds).
- Again we have the equivalence relation: $\sigma_\theta(R_1 - R_2) \equiv \sigma_\theta(R_1) - R_2$ (replacing '$-$' with $\cap$ also holds, but not for $\cup$).
- Distributive property of projection over union: $\pi_X(R_1 \cup R_2) \equiv (\pi_X(R_1)) \cup (\pi_X(R_2))$. Note that, the attribute set X belongs to both R_1 and R_2.

Motivation behind query tuning

For a given query, find a *correct* execution plan that has the lowest *cost* (e.g., time, size, etc.).

Note that, no optimizer can truly produce the *optimal* plan, hence do the following:

- Use estimation techniques to guess real plan cost.
- Use heuristics to limit the search space.

Note: Query tuning is a part of the DBMS and it is proven to be NP-Complete.

The costs to be optimized

For a given query, an estimatation of the cost of executing a plan for the current state of the database is carried out. These include the following:

- Interactions with other work in DBMS
- Size of intermediate results
- Choices of algorithms, access methods
- Resource utilization (CPU, I/O, network)
- Data properties (skew, order, placement)

Nested query processing

Query optimizers often use nested loops while joining tables containing small number of rows with an efficient driving condition. It is important to have an index on column of inner join table as this table is probed every time for a new value from outer table.

However, nested queries are quite complicated to process.

Note: We cannot always un-nest sub-queries (its tricky!!!).

Query tuning – Example I

If Cartesian product and natural join on the pair of relations $R1$ and $R2$, with number of tuples t_1 and t_2 respectively, produce the same result, then (if possible) tune the following query:

$$\sigma_{R1.X=10}(R1 \times (R1 \times R2)).$$

Estimate the size of your query.

Query tuning – Example I

If Cartesian product and natural join on the pair of relations $R1$ and $R2$, with number of tuples t_1 and t_2 respectively, produce the same result, then (if possible) tune the following query:

$$\sigma_{R1.X=10}(R1 \times (R1 \times R2)).$$

Estimate the size of your query.

Solution: ???

Logical and physical plan

The optimizer generates a mapping of a logical algebra expression to the optimal equivalent physical algebra expression.

Physical operators define a specific execution strategy using a particular access path.
– They can depend on the physical format of the data that they process (i.e., sorting, compression).
– Not always a 1:1 mapping from logical to physical.

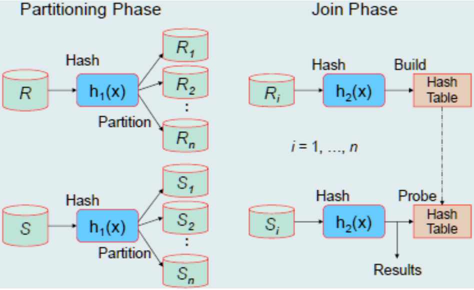
Physical plans – Hash join

Hash joins are used while joining large tables. The optimizer uses smaller of the two tables to build a hash table in memory and then scans the larger table and compares its hash values (of tuples from larger table) with the hash table to find the joined rows.

The algorithm of hash join is divided in two parts as follows:

1. Build an in-memory hash table on smaller of the two tables.
2. Probe this hash table with hash value for each row in the other table

Physical plans – Hash join



Physical plans – Sort Merge join

Sort merge join is used to join two independent data sources. They perform better than the nested loop when the volume of data is big in tables.

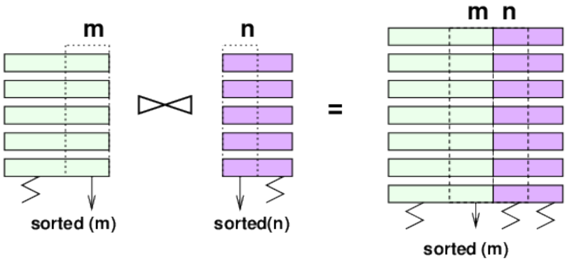
They perform better than hash join when the join condition is either an inequality condition or if sorting is anyways required due to some other attribute (other than join) like *order by*.

The full operation is done in two parts as follows:

1. Sort join operation
2. Merge join operation

Note: If the data is already sorted, first step is avoided.

Physical plans – Sort Merge join



Optimizing search strategies

There are several ways to optimize the searching mechanism as listed below:

- Heuristics
- Heuristics + Cost-based join order search
- Randomized algorithms
- Stratified search
- Unified search

Optimization based on heuristics

Define static rules that transform logical operators to a physical plan. Some of these as follows:

- Perform most restrictive selection early
- Perform all selections before joins
- Predicate/Limit/Projection pushdowns
- Join ordering based on cardinality

Note: Original versions of INGRES and Oracle (until mid 1990s) used this.

Optimization based on heuristics – An example

Consider the following three relations ACTOR, MOVIE and ACTS with primary keys {AID}, {MID} and {AID, MID}, respectively.

ACTOR

AID	Name
1	Amitabh Bachchan
2	Taapsee Pannu
3	Aksay Kumar
4	Prabhas
5	Shraddha Kapoor
6	Kangana Ranaut

MOVIE

MID	Name
1	Badla
2	Kesari
3	Saaho
4	Mental Hai Kya

ACTS

AID	MID
1	1
2	1
3	2
5	3
4	3
6	4

Optimization based on heuristics – An example

Suppose we want to retrieve the names of actors who acted in the movie *Saaho*.

The following query serves the purpose:
select ACTOR.Name from ACTOR, ACTS, MOVIE where ACTOR.AID = ACTS.AID and ACTS.MID = MOVIE.MID and MOVIE.Name = "Saaho";

Let us see how a heuristics based optimizer works!!!

Optimization based on heuristics – An example

Step 1: Decompose a complex query into single-variable queries

Q

select ACTOR.Name from ACTOR, ACTS, MOVIE where ACTOR.AID = ACTS.AID and ACTS.MID = MOVIE.MID and MOVIE.Name = "Saaho";

Q1

select MOVIE.MID into TEMP1 from MOVIE where MOVIE.Name = "Saaho";

Q2

select ACTOR.Name from ACTOR, ACTS, TEMP1 where ACTOR.AID = ACTS.AID and ACTS.MID = TEMP1.MID;

Optimization based on heuristics – An example

Step 1: Decompose a complex query into single-variable queries (Continued ...)

Q2

select ACTOR.Name from ACTOR, ACTS, TEMP1 where ACTOR.AID = ACTS.AID and ACTS.MID = TEMP1.MID;

Q3

select ACTS.AID into TEMP2 from ACTS, TEMP1 where ACTS.MID = TEMP1.MID;

Q4

select ACTOR.Name from ACTOR, TEMP2 where ACTOR.AID = TEMP2.AID;

Optimization based on heuristics – An example

Step 2: Substitute the values in the order Q1 → Q3 → Q4

Q1

select MOVIE.MID into TEMP1 from MOVIE where MOVIE.Name = "Saaho";

MID
3

Optimization based on heuristics – An example

Step 2: Substitute the values in the order Q1 $\rightarrow$ Q3 $\rightarrow$ Q4 (Continued ...)

Q3

select ACTS.AID into TEMP2 from ACTS, TEMP1 where ACTS.MID $=$ 3;

AID
5
4

Optimization based on heuristics – An example

Step 2: Substitute the values in the order Q1 $\rightarrow$ Q3 $\rightarrow$ Q4 (Continued ...)

Q4

select ACTOR.Name from ACTOR, TEMP2 where ACTOR.AID = TEMP2.AID;

Name
Shraddha Kapoor
Prabhas

Optimization based on heuristics – Pros and cons

Advantages:

1. Easy to implement and debug.
2. Works reasonably well and is fast for simple queries.

Disadvantages:

1. Relies on magic constants that predict the efficacy of a planning decision.
2. Nearly impossible to generate good plans when operators have complex inter-dependencies.

Optimization based on heuristics + cost-based join order search

Use static rules to perform initial optimization. Then use dynamic programming to determine the best join order for tables.

- First cost-based query optimizer
- Bottom-up planning (forward chaining) using a divide-and-conquer search method

Note: System R, early IBM DB2, most open-source DBMSs used this.

System R style optimization

1. Break query up into blocks and generate the logical operators for each block.
2. For each logical operator, generate a set of physical operators that implement it.
   – All combinations of join algorithms and access paths
3. Then iteratively construct a *left-deep* tree that minimizes the estimated amount of work to execute the plan.

System R style optimization – An example

Suppose we want to retrieve the names of actors who acted in the movie *Saaho* ordered by their actor ID.

The following query serves the purpose:
select ACTOR.Name from ACTOR, ACTS, MOVIE where ACTOR.AID = ACTS.AID and ACTS.MID = MOVIE.MID and MOVIE.Name = "Saaho" order by ACTOR.AID;

Let us see how the System R optimizer works!!!

System R style optimization – An example

Step 1: Choose the best access paths to each table

- ACTOR: Sequential Scan
- ACTS: Sequential Scan
- MOVIE: Index Look-up on Name

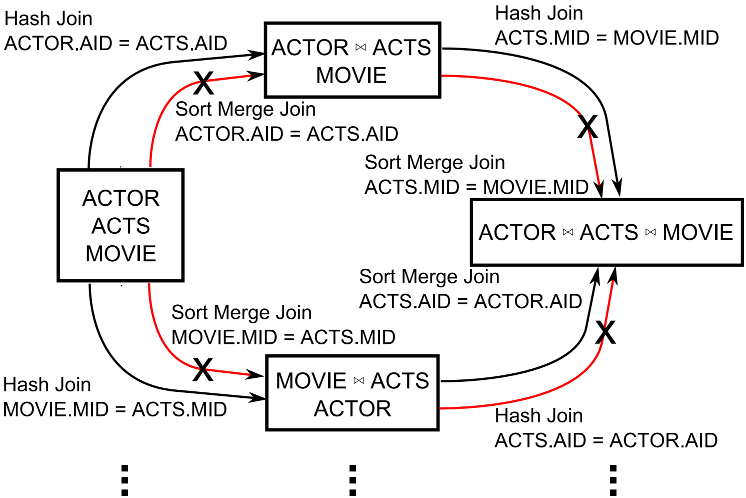
System R style optimization – An example

Step 2: Enumerate all possible join orderings for tables

- ACTOR $\bowtie$ ACTS $\bowtie$ MOVIE
- ACTOR $\bowtie$ MOVIE $\bowtie$ ACTS
- ACTS $\bowtie$ ACTOR $\bowtie$ MOVIE
- ACTS $\bowtie$ MOVIE $\bowtie$ ACTOR
- MOVIE $\bowtie$ ACTOR $\bowtie$ ACTS
- MOVIE $\bowtie$ ACTS $\bowtie$ ACTOR

System R style optimization – An example

Step 3: Determine the join ordering with the lowest cost



Optimization based on heuristics + cost-based join order search – Pros and cons

Advantages:

1. Usually finds a reasonable plan without having to perform an exhaustive search.

Disadvantages:

1. All the same problems as the heuristic-only approach.
2. Left-deep join trees are not always optimal.
3. Have to take in consideration the physical properties of data in the cost model (e.g., sort order).

Optimization based on randomized algorithms

Perform a random walk over a solution space of all possible (valid) plans for a query.

Continue searching until a cost threshold is reached or the optimizer runs for a particular length of time.

Note: Postgres' genetic algorithm used this.

Optimization based on randomized algorithms – Pros and cons

Advantages:

1. Jumping around the search space randomly allows the optimizer to get out of local minimums.
2. Low memory overhead (if no history is kept).

Disadvantages:

1. Difficult to determine why the DBMS may have chosen a particular plan.
2. Have to do extra work to ensure that query plans are deterministic.
3. Still have to implement correctness rules.

Optimization based on stratified search

First rewrite the logical query plan using transformation rules.
– The engine checks whether the transformation is allowed before it can be applied.
– Cost is never considered in this step.

Finally, perform a cost-based search to map the logical plan to a physical plan.

Optimization based on unified search

Unify the notion of both logical $\rightarrow$ logical and logical $\rightarrow$ physical transformations.
– No need for separate stages because everything is transformations.

This approach generates a lot more transformations so it makes heavy use of memorization to reduce redundant work.