Distributed Query Optimization

Definition

- Query optimization refers to the process of producing a query execution plan (QEP) which represents an execution strategy for the query. This QEP minimizes an objective cost function.
- A query optimizer, the software module that performs query optimization, is usually seen as consisting of three components: a search space, a cost model, and a search strategy

- Search space is the set of alternative execution plans that represent the input query. These plans are equivalent, in the sense that they yield the same result, but they differ in the execution order of operations and the way these operations are implemented, and therefore in their performance. The search space is obtained by applying transformation rules
- The cost model predicts the cost of a given execution plan. To be accurate, the cost model must have good knowledge about the distributed execution environment.
- **The search strategy explores the search space and selects the best plan, using the cost model.** It defines which plans are examined and in which order. The details of the environment (centralized versus distributed) are captured by the search space and the cost model.

Three classes of environmental factors

- The first class for frequently changing factors (every second to every minute) includes CPU load, I/O throughput, and available memory
- The second class for slowly changing factors (every hour to every day) includes DBMS configuration parameters, physical data organization on disks, and database schema
- The third class for almost stable factors (every month to every year) includes DBMS type, database location, and CPU speed.

We focus on solutions that deal with the first two classes.

Centralized Query Optimization

- A distributed query is translated into local queries, each of which is processed in a centralized way. Also, distributed query optimization techniques are often extensions of the techniques for centralized systems.
- Dynamic Query Optimization: Optimization done with execution. The *Query Execution Plan (QEP)* is dynamically constructed by the query optimizer which makes calls to the DBMS execution engine for executing the query's operations. Thus, there is no need for a cost model.
- With static query optimization, there is a clear separation between the generation of the QEP at compile-time and its execution by the DBMS execution engine. Thus, an accurate cost model is key to predict the costs of candidate QEPs.

Centralized Query Optimization

- Dynamic and static query optimization both have advantages and drawbacks. Dynamic query optimization mixes optimization and execution and thus can make accurate optimization choices at run-time. However, query optimization is repeated for each execution of the query. Therefore, this approach is best for ad-hoc queries.
- Static query optimization, done at compilation time, amortizes the cost of optimization over multiple query executions. The accuracy of the cost model is thus critical to predict the costs of candidate QEPs. This approach is best for queries embedded in stored procedures, and has been adopted by all commercial DBMSs

Centralized Query Optimization

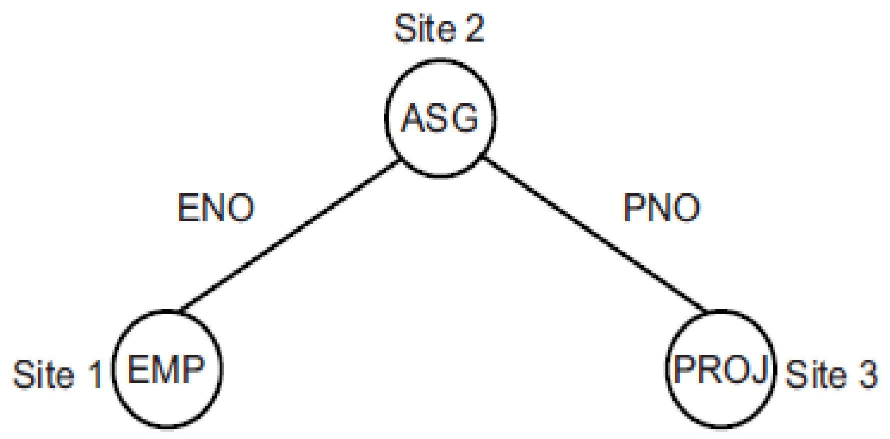
- However, even with a sophisticated cost model, there is an important problem that prevents accurate cost estimation and comparison of QEPs at compile-time: *the actual bindings of parameter values in embedded queries is not known until run-time.*
- Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates.
- The Hybrid approach is basically static, but further optimization decisions may take place at run time. This approach was pioneered in System R by adding a conditional runtime reoptimization phase for execution plans statically optimized

Join Ordering

- Some algorithms optimize the ordering of joins directly without using semijoins.
- In order to overcome the difficulty of join ordering, mostly use of semijoins is preferred to optimize join queries.
- A number of assumptions are necessary to concentrate on the main issues. Since the query is localized and expressed on fragments, we do not need to distinguish between fragments of the same relation and fragments of different relations.
- To simplify notation, we use the term relation to designate a fragment stored at a particular site. Also, to concentrate on join ordering, we ignore local processing time, assuming that reducers (selection, projection) are executed locally either before or during the join (remember that doing selection first is not always efficient). Therefore, we consider only join queries whose operand relations are stored at different sites. We assume that relation transfers are done in a set-at-a-time mode rather than in a tuple-at-a-time mode. Finally, we ignore the transfer time for producing the data at a result site.

Example of Join Ordering

$$\text{PROJ} \bowtie_{\text{PNO}} \text{ASG} \bowtie_{\text{ENO}} \text{EMP}$$



Join Graph of Distributed Query

This query can be executed in at least five different ways as below:

1. EMP $\rightarrow$ site 2; Site 2 computes EMP$'$ = EMP $\bowtie$ ASG; EMP$'$ $\rightarrow$ site 3; Site 3 computes EMP$'$ $\bowtie$ PROJ.
2. ASG $\rightarrow$ site 1; Site 1 computes EMP$'$ = EMP $\bowtie$ ASG; EMP$'$ $\rightarrow$ site 3; Site 3 computes EMP$'$ $\bowtie$ PROJ.
3. ASG $\rightarrow$ site 3; Site 3 computes ASG$'$ = ASG $\bowtie$ PROJ; ASG$'$ $\rightarrow$ site 1; Site 1 computes ASG$'$ $\bowtie$ EMP.
4. PROJ $\rightarrow$ site 2; Site 2 computes PROJ$'$ = PROJ $\bowtie$ ASG; PROJ$'$ $\rightarrow$ site 1; Site 1 computes PROJ$'$ $\bowtie$ EMP.
5. EMP $\rightarrow$ site 2; PROJ $\rightarrow$ site 2; Site 2 computes EMP $\bowtie$ PROJ $\bowtie$ ASG

To select one of these programs, the following sizes must be known or predicted: size(EMP), size(ASG), size(PROJ), size(EMP 1 ASG), and size(ASG 1 PROJ).

Semi-joins

- A semi-join between two tables returns rows that match an EXISTS subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery.
- Semi-joins are written using EXISTS or IN.
- A semi-join can be performed using the nested loops, hash join, or merge join algorithms

Semi join in query optimization

- The join of two relations R and S over attribute A, stored at sites 1 and 2, respectively, can be computed by replacing one or both operand relations by a semijoin with the other relation, using the following rules:

$$
\begin{aligned}
R \bowtie_A S &\Leftrightarrow (R \ltimes_A S) \bowtie_A S \\
&\Leftrightarrow R \bowtie_A (S \ltimes_A R) \\
&\Leftrightarrow (R \ltimes_A S) \bowtie_A (S \ltimes_A R)
\end{aligned}
$$

The use of the semi-join is beneficial if the cost to produce and send it to the other site is less than the cost of sending the whole operand relation and of doing the actual join. To illustrate the potential benefit of the semi-join.

More generally, the semijoin can be useful in reducing the size of the operand relations involved in multiple join queries.

Semi join in query optimization

1. $\Pi_A(S) \rightarrow$ site 1
2. Site 1 computes $R' = R \ltimes_A S$
3. $R' \rightarrow$ site 2
4. Site 2 computes $R' \bowtie_A S$

The cost of the join-based
algorithm is that of transferring relation R to site 2. The
cost of the semijoin-based
algorithm is the cost of steps 1 and 3 above. Therefore,
the semijoin approach is
better if

$$size(\Pi_A(S)) + size(R \ltimes_A S) < size(R)$$

Semi join in query optimization

- For a given relation, there exist several potential semi-join programs. The number of possibilities is in fact exponential in the number of relations. But there is one optimal semi-join program, called the *full reducer*, which for each relation R reduces R more than the others. A simple method is to evaluate the size reduction of all possible semi-join programs and to select the best one. The problems with the enumerative method are twofold:

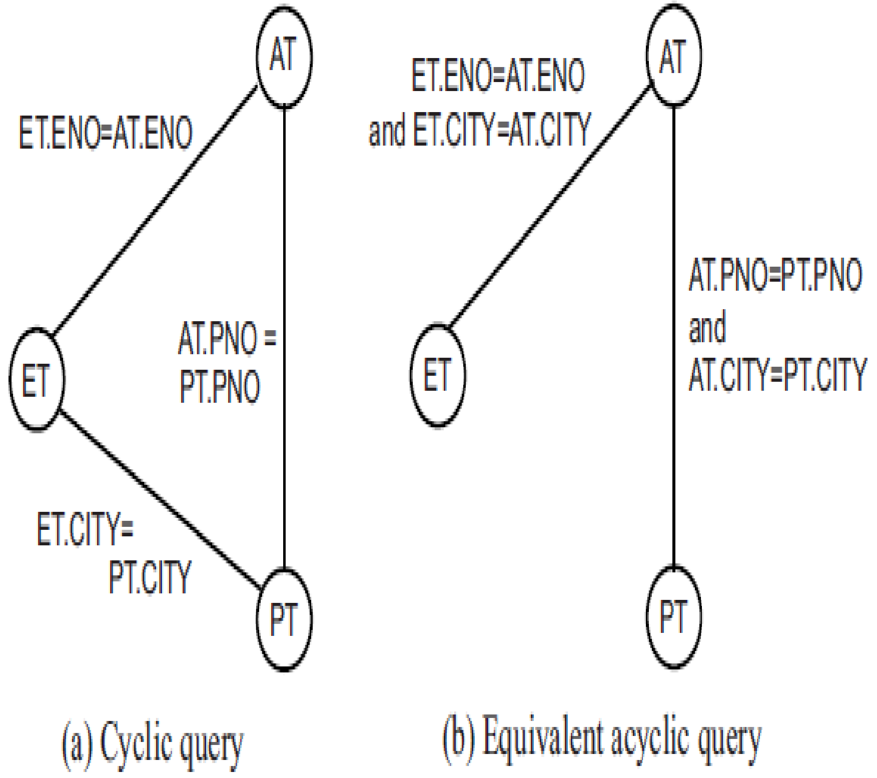
  1. There is a class of queries, called cyclic queries, that have cycles in their join graph and for which full reducers cannot be found.

  2. For other queries, called tree queries, full reducers exist, but the number of candidate semi-join programs is exponential in the number of relations, which makes the enumerative approach NP-hard.

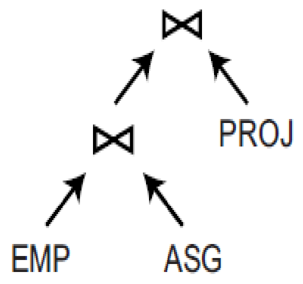
Transformation of Cyclic Query

Consider the following relations, where attribute CITY has been added
to relations EMP (renamed ET), PROJ (renamed PT) and ASG (renamed AT) of
the engineering database. Attribute CITY of AT corresponds to the city where the
employee identified by ENO lives.
ET(ENO, ENAME, TITLE, CITY)
AT(ENO, PNO, RESP, DUR)
PT(PNO, PNAME, BUDGET, CITY)
The following SQL query retrieves the names of all employees living in the city
in which their project is located together with the project name.

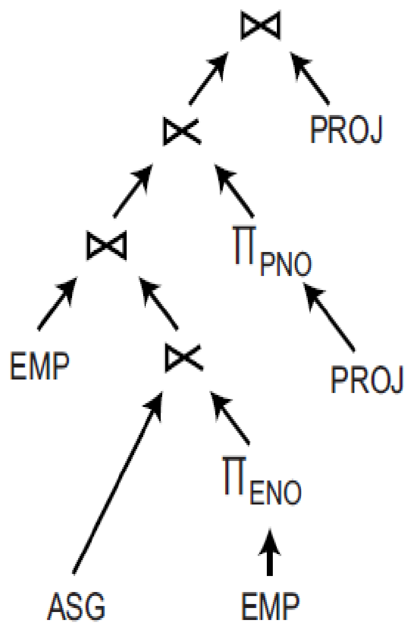
```
SELECT ENAME, PNAME
FROM ET, AT, PT
WHERE ET.ENO = AT.ENO
AND AT.ENO = PT.ENO
AND ET.CITY = PT.CITY
```


AT
ET.ENO=AT.ENO
ET
AT.PNO = PT.PNO
ET.CITY= PT.CITY
PT
(a) Cyclic query
ET.ENO=AT.ENO and ET.CITY=AT.CITY
AT
ET
AT.PNO=PT.PNO and AT.CITY=PT.CITY
PT
(b) Equivalent acyclic query

Join vs Semi-join Approaches



(a) Join approach



(b) Semijoin approach

1. Compared with the join, the semijoin induces more operations but possibly on smaller operands.

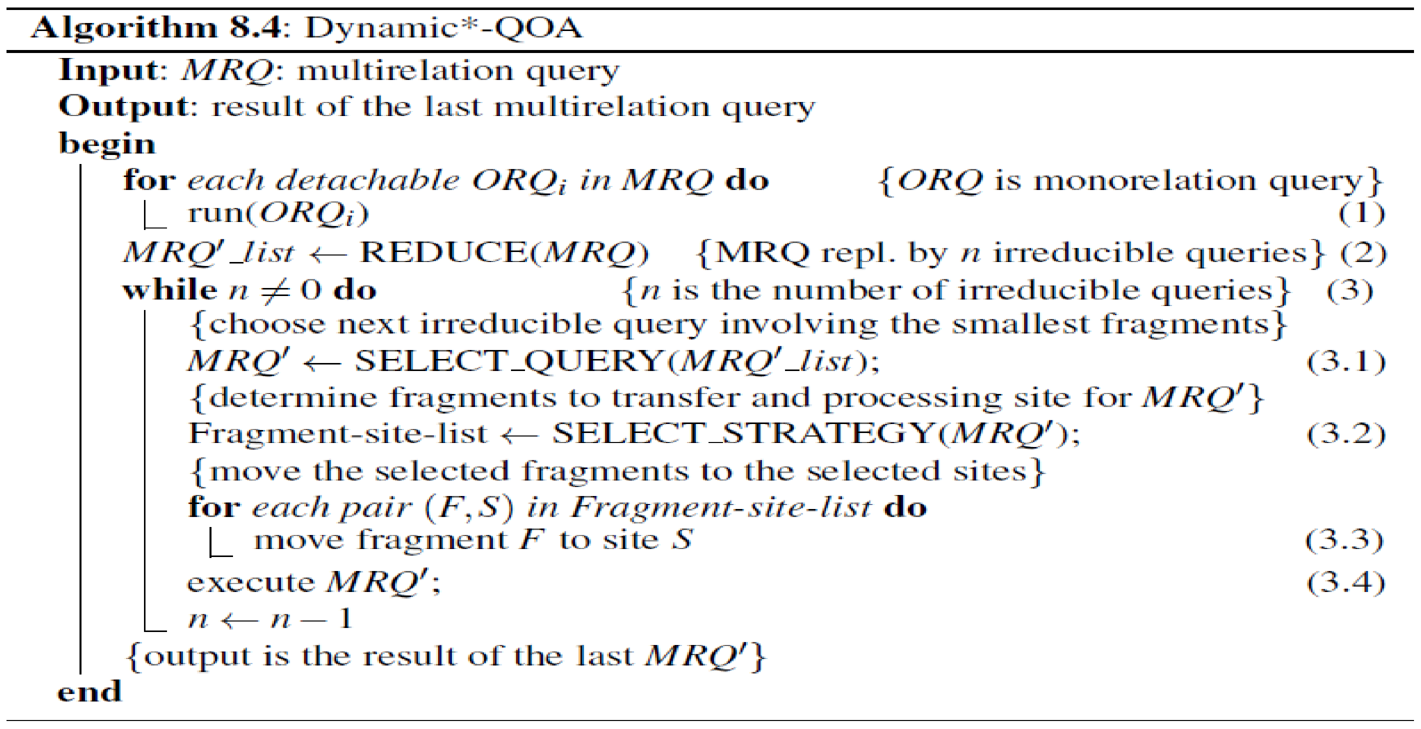
2. If the join attribute length is smaller than the length of an entire tuple and the semijoin has good selectivity, then the semijoin approach can result in significant savings in communication time.

3. Using semijoins may well increase the local processing time, since one of the two joined relations must be accessed twice.

4. Using semijoins might not be a good idea if the communication time is not the dominant factor, as is the case with local area networks

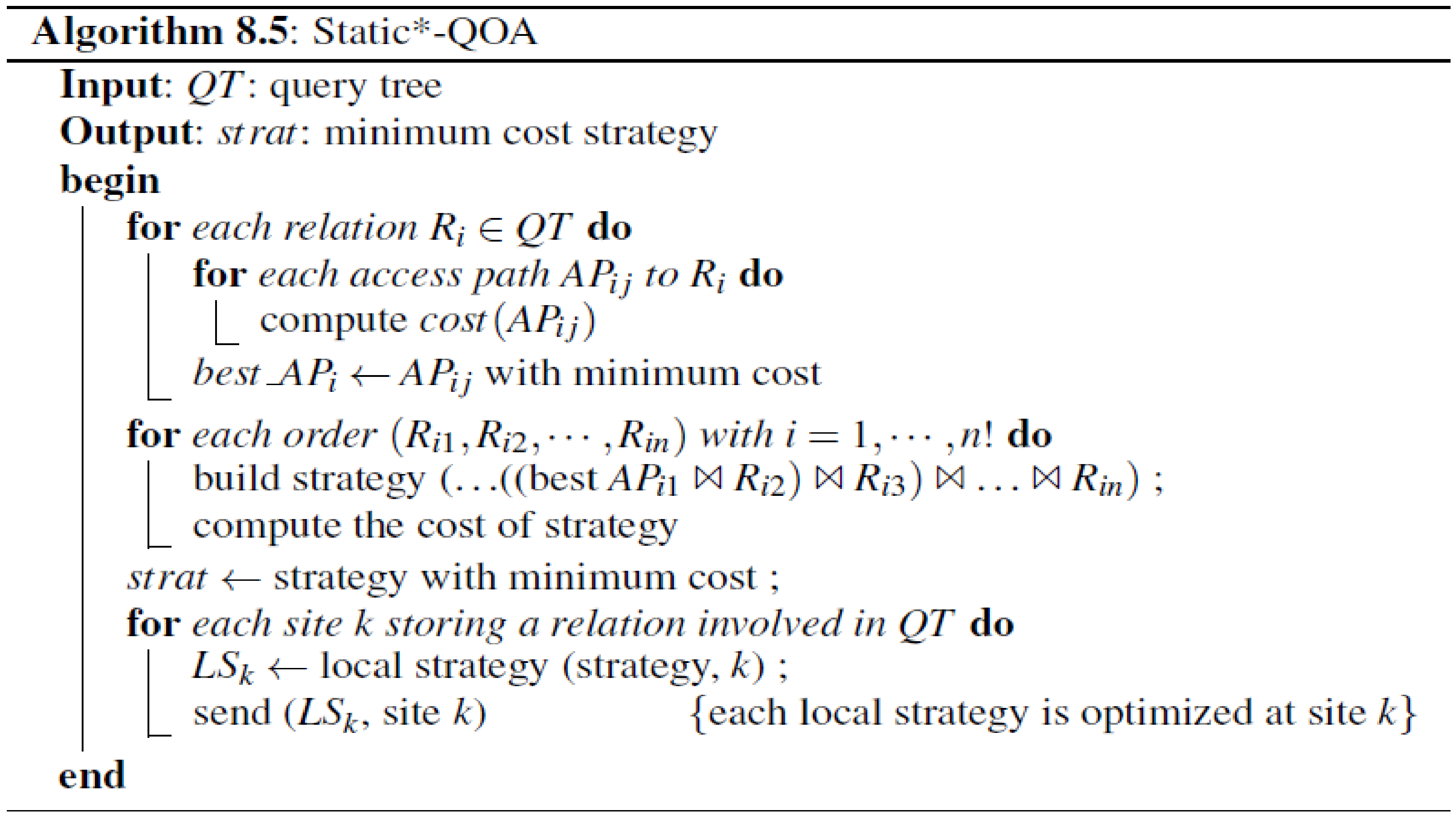
Distributed Query Optimization: Dynamic Approach

The input to the algorithm is a query expressed in tuple relational calculus (in conjunctive normal form) and schema information (the network type, as well as the location and size of each fragment). This algorithm is executed by the site, called the master site, where the query is initiated.



```
Algorithm 8.4: Dynamic*-QOA
Input: $MRQ$: multirelation query
Output: result of the last multirelation query
begin
    for each detachable $ORQ_i$ in $MRQ$ do          {$ORQ$ is monorelation query}
        run($ORQ_i$)                                                        (1)
    $MRQ'\_list \leftarrow$ REDUCE($MRQ$)   {MRQ repl. by $n$ irreducible queries} (2)
    while $n \neq 0$ do                   {$n$ is the number of irreducible queries}   (3)
        {choose next irreducible query involving the smallest fragments}
        $MRQ' \leftarrow$ SELECT_QUERY($MRQ'\_list$);                            (3.1)
        {determine fragments to transfer and processing site for $MRQ'$}
        Fragment-site-list $\leftarrow$ SELECT_STRATEGY($MRQ'$);                 (3.2)
        {move the selected fragments to the selected sites}
        for each pair $(F, S)$ in Fragment-site-list do
            move fragment $F$ to site $S$                                      (3.3)
        execute $MRQ'$;                                                      (3.4)
        $n \leftarrow n - 1$
    {output is the result of the last $MRQ'$}
end
```

Distributed Query Optimization: Static Approach



Algorithm 8.5: Static*-QOA

```
Input: $QT$: query tree
Output: $strat$: minimum cost strategy
begin
    for each relation $R_i \in QT$ do
        for each access path $AP_{ij}$ to $R_i$ do
            compute $cost(AP_{ij})$
        $best\_AP_i \leftarrow AP_{ij}$ with minimum cost
    for each order $(R_{i1}, R_{i2}, \cdots, R_{in})$ with $i = 1, \cdots, n!$ do
        build strategy $(\ldots((\text{best } AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie \ldots \bowtie R_{in})$ ;
        compute the cost of strategy
    $strat \leftarrow$ strategy with minimum cost ;
    for each site $k$ storing a relation involved in $QT$ do
        $LS_k \leftarrow$ local strategy (strategy, $k$) ;
        send ($LS_k$, site $k$)            {each local strategy is optimized at site $k$}
end
```

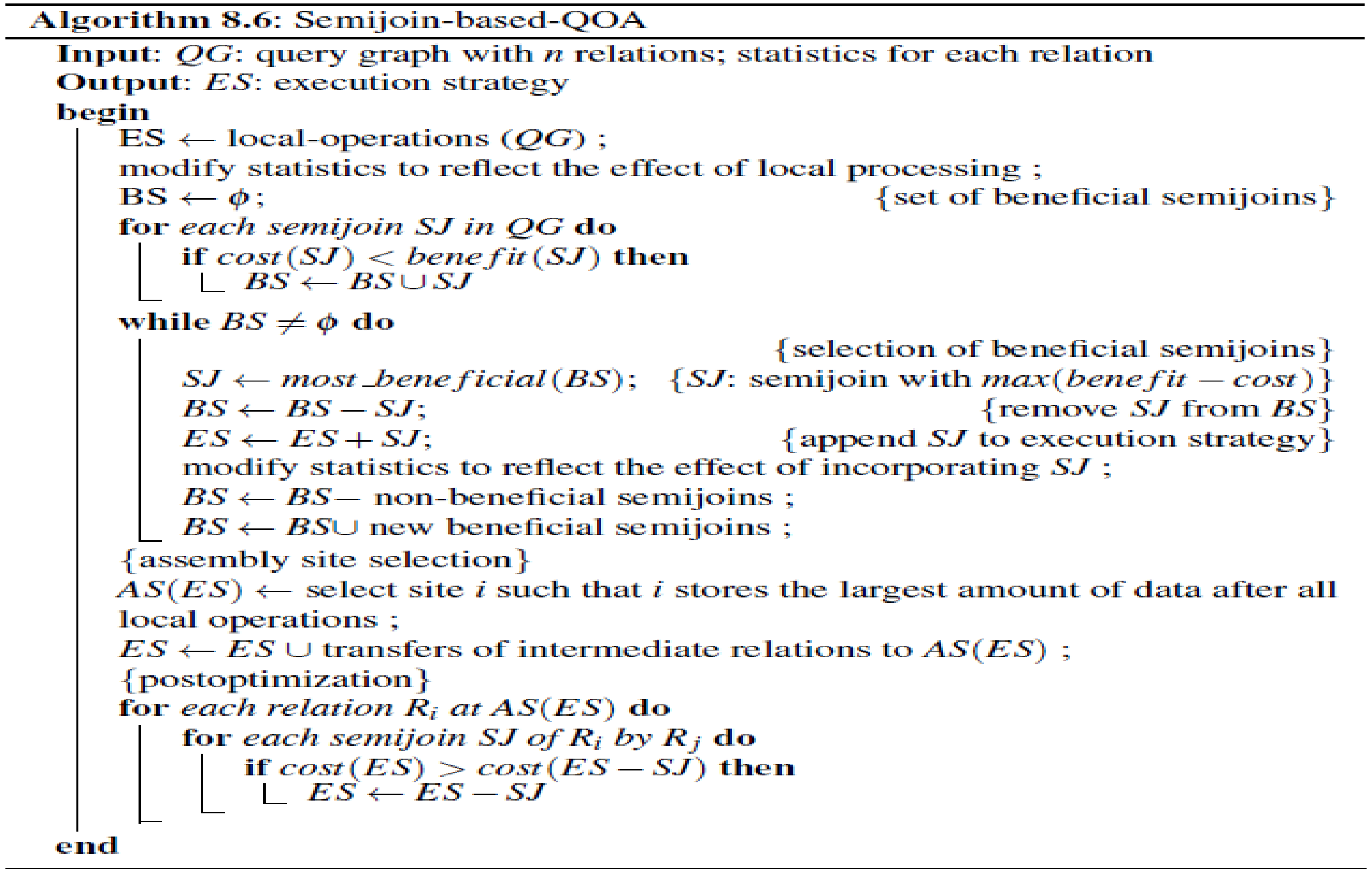
Distributed Query Optimization: Static Approach

- The optimizer must select the join ordering, the join algorithm (nested-loop or merge-join), and the access path for each fragment (e.g., clustered index, sequential scan, etc.). These decisions are based on statistics and formulas used to estimate the size of intermediate results and access path information.
- In addition, the optimizer must select the sites of join results and the method of transferring data between sites. To join two relations, there are three candidate sites: the site of the first relation, the site of the second relation, or a third site (e.g., the site of a third relation to be joined with). Two methods are supported for intersite data transfers.

Distributed Query Optimization: Static Approach

1. Ship-whole. The entire relation is shipped to the join site and stored in a temporary relation before being joined. If the join algorithm is merge join, the relation does not need to be stored, and the join site can process incoming tuples in a pipeline mode, as they arrive.
2. Fetch-as-needed. The external relation is sequentially scanned, and for each tuple the join value is sent to the site of the internal relation, which selects the internal tuples matching the value and sends the selected tuples to the site of the external relation. *This method is equivalent to the semijoin of the internal relation with each external tuple.*

Distributed Query Optimization: Semi-join based approach

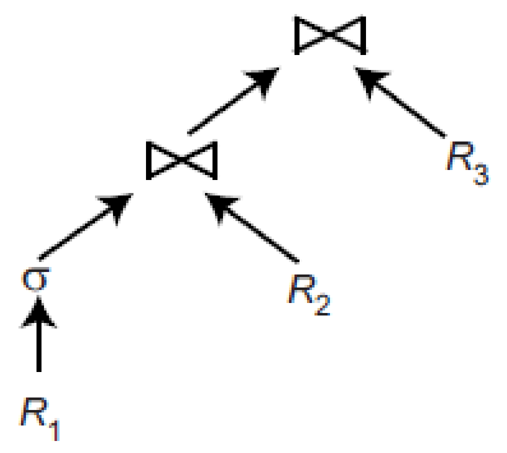


```
Algorithm 8.6: Semijoin-based-QOA
Input: $QG$: query graph with $n$ relations; statistics for each relation
Output: $ES$: execution strategy
begin
    ES $\leftarrow$ local-operations ($QG$) ;
    modify statistics to reflect the effect of local processing ;
    BS $\leftarrow \phi$ ;                                  {set of beneficial semijoins}
    for each semijoin SJ in QG do
        if $cost(SJ) < benefit(SJ)$ then
            $BS \leftarrow BS \cup SJ$
    while $BS \neq \phi$ do
                                                  {selection of beneficial semijoins}
        $SJ \leftarrow most\_beneficial(BS)$;   {$SJ$: semijoin with $max(benefit - cost)$}
        $BS \leftarrow BS - SJ$;                        {remove $SJ$ from $BS$}
        $ES \leftarrow ES + SJ$;               {append $SJ$ to execution strategy}
        modify statistics to reflect the effect of incorporating $SJ$ ;
        $BS \leftarrow BS -$ non-beneficial semijoins ;
        $BS \leftarrow BS \cup$ new beneficial semijoins ;
    {assembly site selection}
    $AS(ES) \leftarrow$ select site $i$ such that $i$ stores the largest amount of data after all
    local operations ;
    $ES \leftarrow ES \cup$ transfers of intermediate relations to $AS(ES)$ ;
    {postoptimization}
    for each relation $R_i$ at $AS(ES)$ do
        for each semijoin SJ of $R_i$ by $R_j$ do
            if $cost(ES) > cost(ES - SJ)$ then
                $ES \leftarrow ES - SJ$
end
```

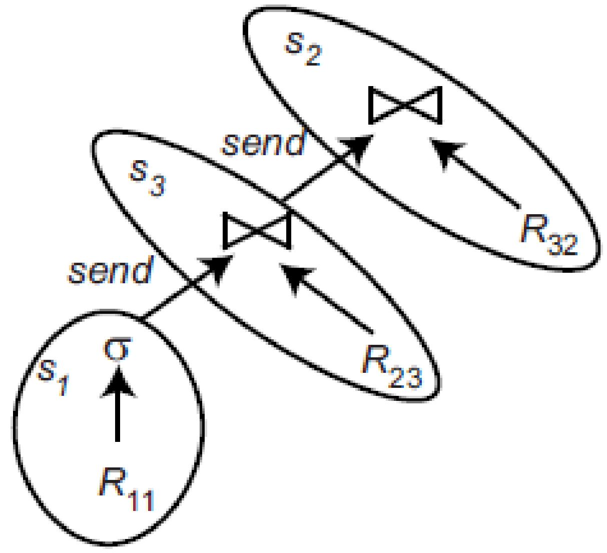
The semijoin-based algorithm proceeds in four phases: initialization, selection of beneficial semijoins, assembly site selection, and postoptimization. The output of the algorithm is a global strategy for executing the query

***benefit is the cost of transferring irrelevant tuples of R (which is avoided by the semijoin):

Distributed Query Optimization: Hybrid approach



(a) Static plan



(b) Run-time plan

The first step can be done by a centralized query optimizer. It may also include
choose-plan operators so that runtime bindings can be used at startup time to make
accurate cost estimations. The second step carries out site and copy selection, possibly
in addition to choose-plan operator execution.