Multidatabase Query Optimization: Tree Balance and Node Allocation

Yangjun Chen and Wolfgang Benn

Department of Computer Science, Technical University of Chemnitz-Zwickau,
09107 Chemnitz, Germany

Abstract In this paper, we consider the query evaluation problem in multidatabases and develop an algorithm for generating optimal plans for queries submitted to an integrated schema. we try to extend the basic transformation step used in the method proposed by Du *et al* [DSD95] and construct a dynamic time table to support a query optimization process. In this way, not only the join tree balance, but also the node allocation can be achieved according to both communication costs and load measurements. In addition, the queuing analysis has been utilized to estimate the response times of local database systems.

1. Introduction

Multidatabase management systems enable data sharing among heterogeneous local databases and thus provide interoperability required by diverse applications [BOT86, CW93, Du94, HLM94, KC93]. In such systems, as with distributed databases, the query optimization is very important but much different from the traditional query strategies. First, due to the heterogeneity of component databases, a query submitted to a multidatabase has to decomposed and translated so that it can be evaluated against different, possibly heterogenous, local databases (see [CB96, KLK91, LBT93]). On the other hand, due to the autonomy requirement, not only the communication costs but also the load measurements of local databases must be considered to generate an optimal execution plan for a given query.

In this paper, we confine ourselves only on how to generate the optimal execution plans for queries and try to develop an algorithm for this issue, by which both the join tree balance and the node allocation can be achieved. (As to the query translation, the interested reader is referred to [CB96] for a detailed description.) Many theoretical solutions have been proposed and different implementations provide varying approaches for such problems [DSD95, ETB96, LOG93]. In [DSD95], a hybrid algorithm has been developed to translate a left deep join tree into a balanced bushy join tree based on a simple cost analysis, so that the response time can be reduced. However, how to allocate a (join) operation to a local database system is untouched. In [ETB96], another approach has been proposed to find both the allocation and the sequence of join operations using linear programming. But the balance problem has not been taken into account there. Furthermore, in this method, only the communication costs are considered (see Appendix B of [ETB96]) and the load measurements has not been addressed at all. In contrast, our method is a combination of the node allocation and the bushy join tree balance, based on which an optimal execution plan for a given query can be produced. The key idea of it is the analysis of load measurement of local databases using the queuing analysis and the recursive generation of balanced bushy join trees according to both the communication costs and the actual load states. We argue that the load measurement is important for the query evaluation in a multidatabase due to the autonomy of local databases, which affects the response time of queries significantly.

For this purpose, we introduce the concept of the dynamic time table and modify the basic transformation step suggested in [DSD95], so that the generation of balanced bushy join trees can be made based

The work reported here is supported by DFG (Deutsche Forschungsgemeinschaft) under grant No. Be1786/1-1.

on an overall cost estimation. In addition, we develop a recursive algorithm which works more efficiently than that proposed by Du *et al* [DSD95].

The remainder of the paper is organized as follows. In Section 2, we show an example, thereby demonstrating the insufficiency of the existing approachs. Then, in Section 3, we discuss the communication costs and the load measurements as well as how to maintain the load informations. In Section 4, we show our algorithms for both the simple and general cases. Section 5 is a short conclusion.

2. An sample example

In order to specify the problem, let us consider a multidatabase for patient records, which consists of the following relations.

patient:	[*pid, name, clinic, insurance*]
physician:	[*phid, name, clinic*]
notes:	[*nid, pid**, phid*, note_text*]
prescription:	[*phid, drugname, prescription_text*]

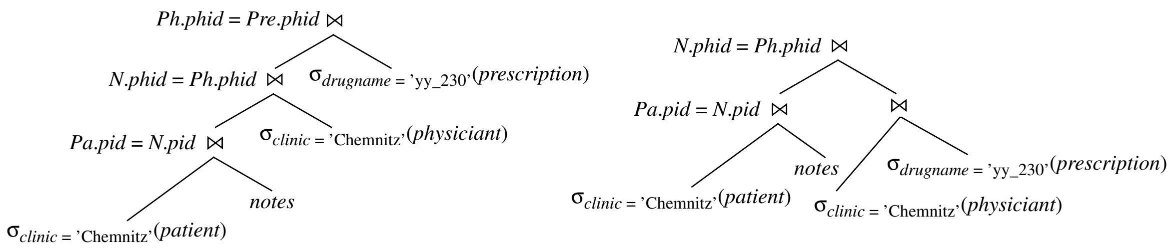
(In the relations, all the primary keys are underlined and the foreign keys are marked with asterisks.)

Fig. 1. A sample integrated relational schema

We assume that these relations are imported from four different component databases. The following query finds the names of patients at the Chemnitz Clinic, for whom an experimental drug yy_230 is prescribed, and the corresponding prescription text as well as the names of physicians who are responsible for them.

select *Pa.name*, *Pre.prescription_text, Ph.name*
from *patient Pa*, *physician Ph*, *notes N*, *prescription Pre*,
where *Pa.clinic* = ‘Chemnitz’ **and** *Ph.clinic* = ‘Chemnitz’ **and**
Pre.drugname = ‘yy_230’ **and** *Pa.pid* = *N.pid* **and**
Ph.phid = *Pre.phid* **and** *N.phid* = *Ph.phid*

An optimal left deep join tree for evaluating this query may be of the form as shown in Fig. 2(a).



(a) (b)

Fig. 2. Left deep join tree and balanced bushy join tree

Then, based on the cardinalities of local relations and the cost of each join operation, the algorithm proposed in [DSD95] may translate it into a balanced bushy join tree as shown in Fig. 2(b). Obviously, the plan has not yet been completed, since we do not know in which local database system a join operation will be performed. That is, the node allocation has not be suggested by this plan. But to do this, we should not only consider the communication costs, but also the load states of local databases. In other words, the information on how frequently queries arrives at a local database, namely, the information on how long time a local database system takes to react to a query averagely should be maintained. Then, based on the communication costs and it, the combination of the join tree balance and the node allocation can be achieved. But the cost calculation must be elaborated. Especially, in the case that more than one relations residing in the same local database participates in a join tree, the task of allocating nodes becomes non-trivial and a more sophisticated method is necessary. In the following sections, we discuss this method in detail.

3. Communication costs and load measurement

As demonstrated above, we have to know the load states of local databases as well as the communication costs to develop an optimal execution plan for a given query. To this end, we maintain a static matrix for transmission rate between each pair of local databases and a dynamic table for load measurements, which will be changed periodically to indicate the actual load states of local databases (see Fig. 3(a) and (b)).

transmission cost: c_{ij} = per bit $\times 10^{-4}$

	j			
i	1	2	3	4
1	0	2	3	3
2	2	0	1	2
3	3	1	0	1
4	3	2	1	0

(a)

	average service time s (sec.)	current query rate λ (per minute)
DB_1	6.0	2
DB_2	4.0	5
DB_3	7.0	3
DB_4	8.0	1.5

(b)

Fig. 3. Transmission cost matrix and load measurement

Fig. 3(a) is a sample transmission cost matrix, in which each c_{ij} corresponds to the transmission rate from DB_i to DB_j. In the table shown in Fig. 3(b), each entry is used to store the actual frequency of queries arriving at some local database and its average serving time, in terms of which the response time can be calculated based on the queuing analysis. Concretely, we have the following formulas for computing response times, by assuming that the query arrival rate obeys the Poisson distribution [Co81]:

$$\text{system load: } \rho = \lambda \cdot s \qquad (1)$$

$$\text{average response time: } = \begin{cases} t_q = s/(1 - \rho), & \rho < 1 \\ \infty, & \rho \geq 1 \end{cases} \qquad (2)$$

where λ represents the current query arrival rate and s represents the average service time.

For example, from the first entry of the table shown in Fig. 3(b), we can know that the response time of DB_1 is $t_q = s/(1 - \rho) = 6.0/(1 - 0.2) = 7.5$.

4. Plan generation

We consider queries that are expressible as conjunctive queries or as projection-selection-join queries where the selection and join conditions are restricted to equality. In terms of above estimation information, we can develop an efficient algorithm for constructing execution plans for any such query submitted to a multidatabase. The algorithm can be simply described as shown in Fig. 4.

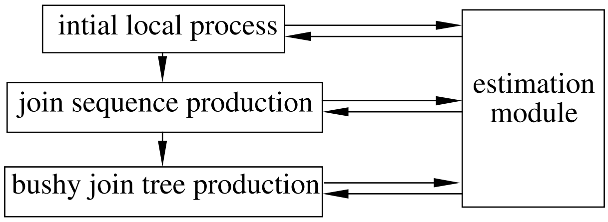


Fig. 4. Process for generating an execution plan for a given query

The algorithm mainly consists of three phases:

- initial local process,
- join sequence production and
- bushy join tree production.

Additionally, the estimation module is used to estimate the sizes of temporary relations in terms of the approach given in Appendix A.

(1) *Initial local process.* The initial local process specifies all the local selection operations in the query. It identifies all the attributes of each relation that appears in the original query. Furthermore, the attributes that are needed to perform the required normal joins that do not explicitly appear in the query are also identified. The initial local process performs the local projection operations for each relation over the previously identified attributes in order to reduce the sizes of the relations. In addition, the size estimation procedure shown in Appendix A for the selection and projection operations can be used to derive the sizes of the resultant temporary relations.

(2) *Join sequence production.* After the initial local process, we can determine an optimal order for the temporary relations resulting from the initial local process. This can be done in the same way as discussed in [HS93] to avoid cartesian products and to minimize the size of intermediate relations.

(3) *Bushy join tree production.* Principally, the method proposed by Du *et al.* [DSD95] can be extended to construct balanced bushy join trees (from optimal join sequences) in the simple case that each participating relation resides in a different database. The only difference consists in the load measurement and a more exactly calculation of communication costs as well as a labeling strategy for the node allocation. However, in the case that more than one relations reside in the same database, a more sophisticated method is needed to achieve an optimal plan. Besides, we develop a new recursive control strategy which improves the algorithm of Du *et al.* non-trivially.

In the following, we first discuss two methods for the simple and general cases in 4.1 and 4.2, respectively. Then, in 4.3, we propose another recursive algorithm, which further improves the algorithms discussed in 4.1 and 4.2 significantly.

4.1. Top-down algorithm for constructing balanced bushy join trees

The main idea of the method proposed by Du *et al.* is a repeated application of basic transformations to the join sequence (called the left deep join tree in [DSD95]). According to [DSD95], a basic transformation is a transformation step which takes a segment from a left deep join tree as the input, and then translate this segment as follows:

(1) The top node of the segment is called the *upper anchor node* (UAN) and the bottom node of the segment is called the *lower anchor node* (LAN) (see Fig. 5(a));

(2) The (direct) left child node (n_c) of the UAN (n_a) becomes the new UAN of the transformed segment (see Fig. 5(b)); and

(3) The LAN (n_b) remains unchanged but its (direct) right child node is replaced with a subtree corresponding to a join operation between the respective right child subtrees of n_a and n_b (see Fig. 5(c)).

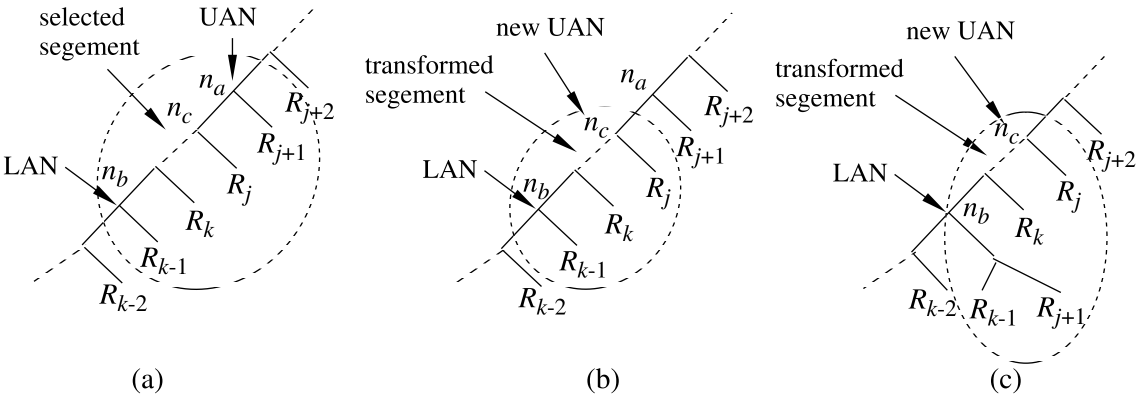


Fig. 5. Basic transformation

The primary use of a basic transformation is to balance left deep join trees. It does so by converting n sequentially executed joins (between and including the UAN and the LAN) into n-1 sequential joins (between and including the LAN and the new UAN) and a new join node (a node corresponding to a join operation) concurrent to the corresponding left child subtree of the LAN. In this way, since the number of sequentially executed joins between the LAN and the new UAN is one less than that of the original one, the query response time can be improved if the new join can be executed parallel to it and at the same time causes no extra costs.

In fact, with the transformation of a left deep join tree into a bushy join tree, we should not restrict UANs and LANs to be on the same segment. Instead, a LAN can be any node in the subtree rooted at a UAN as long as the corresponding transformation reduces the response time of the subtree. Then, we can employ a depth-first search to find any LAN in the subtree and do the corresponding transformation if it is possible. To this end, we introduce the concept of the extended basic transformation.

Definition 1 *An extended basic transformation* (*EB*-transformation) is a transformation step which takes a subtree T as the input, and translate the subtree as follows:

(1) The root of the subtree is called the upper anchor node (UAN) and any node in the subtree may be taken as a lower anchor node (LAN);

(2) The (direct) left child node of the UAN becomes the new UAN of the transformed subtree;

(3) Let n_b be the chosen LAN. Then, n_b remains unchanged but its right child node is replaced with a new join node between n_b's right subtree and the right subtree of the original UAN. Such a transformation is denoted *EB*-transformation(T, n_b).

The advantage of the *EB*-transformation over the basic transformation [DSD95] can be shown in the following example.

Example 2. Consider the bushy join tree shown in Fig. 6(a). The cost of each relation and the response time of each join node are shown in the figure. For ease of exposition, we do not consider the transmission costs here and assume that the response time of each local database system is 5 units of time. Using the hybrid algorithm of [DSD95], this bushy join tree can be translated into the tree shown in Fig. 6(b). But using the basic transformation of [DSD95], the further balance is not possible due to the restriction that a UAN and the corresponding LAN must be on the same segment, no matter what strategies, top-down or bottom-up, are utilized (see Appendix B). However, if we use the *EB*-transformation as the basic transformation step, we can get the tree shown in Fig. 6(c) from the tree shown in Fig. 6(b) by taking n_a as the UAN and n_b as the LAN. From Fig. 6(b), we see that n_a and n_b are not on the same segment.

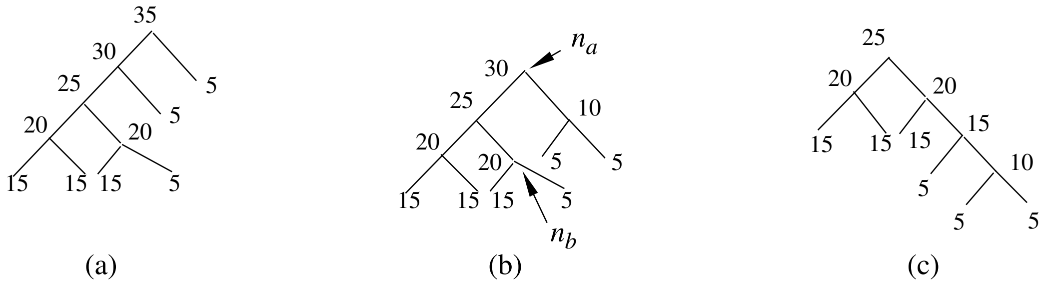


Fig. 6. Basic transformation and extended basic transformation

Intuitively, for any join subtree, what we want is those *EB*-transformations, by which the response time can be improved. It motivates another important concept.

Definition 2 Let T be a join subtree with the root n_a. Let n_c be the left child node of n_a. We say that *EB*-transformation(T, n_b) (n_b stands for a node in T) is time improving if response-time(n_c, T') < response-time(n_a, T), where T' represents the tree obtained by applying *EB*-transformation(T, n_b) to T and response-time(n_c, T') represents the response time of n_c with respect to T', while response-time(n_a, T) represents the response time of n_a with respect to T. The response time of a subtree can be evaluated using formulas (5) and (6) shown below.

As with the basic transformation step of [DSD95], a time improving *EB*-transformation (or simply *TIEB*-transformation) is uniquely identified by a UAN and a LAN. Then, the process of balancing a left deep join tree can be illustrated as a process of repeatedly selecting UANs and LANs and applying the *TIEB*-transformations in terms of the communication costs and the load measurements.

First, for a left deep join tree, we need to make an initial node allocation. This can be done by selecting a site such that if all the join operations are executed at it, the response time is minimal, compared to any site else. Assume that $(...(... (R_1 \bowtie R_2)... R_i) ... \bowtie R_n)$ is a left deep join tree being considered with each R_i from a different local database and the selected site is DB_i. Then, if n (the number of the relations involved in the join tree) is not large, the response time of DB_i for doing all the join operations can be calculated as follows:

$$\mathrm{GRT}(i) = \sum_{j \neq i} c_{ji}|R_j| + (n-1)\,t_i \qquad (3)$$

where t_i represents the current average response time of DB_i. We have this formula in terms of the following computation mode, in which a left deep join tree will be executed if we require that all the join operations are performed at one site. We proceed bottom-up. First, we ship R_1 and R_2 to DB_i from the respective sites and do the join over R_1 and R_2 using the sort merge join method. Then, we get R_3 from another site to perform the join over the result of the previous join and it, and so on. Therefore, we will have (n - 1) times of data transmission and (n - 1) join operations. If n is small, t_i can be thought of as unchanged during the evaluation of the query.

Of course, we can arrange the computation in another way. For example, we can first transfer in parallel all the relations from the respective databases to one site and then perform the join operations successively. Accordingly, the response time can be computed as follows:

$$\mathrm{GRT}(i) = \max_{j \neq i}\{c_{ji}|R_j|\} + (n-1)t_i \qquad (4)$$

For ease of exposition, we will use (3) for the following discussion. However, the analysis can be made in a similar way if (4) is employed.

Then, we traverses the left deep join tree from the root to the leaf and take the root as the initial UAN. For each UAN (the initial UAN, or any newly generated UAN during the tree transformation process), the (recursive) algorithm differentiate among three cases:

(1) If its left child node is a leaf node, the algorithm simply terminates (or returns if the process is recursively called) as the load of the UAN can not be redistributed and all other join nodes have already been processed.

(2) If the response time of the right subtree is about the same or greater than that of the left subtree, no transformation will be applied to the UAN. But the recursive invocations of the algorithm on both right and left subtrees will be performed.

(3) If the response time of the left subtree is greater than that of the right subtree, traverse the subtree rooted at the left child node of the UAN to find a LAN, by which a *TIEB*-transformation can be performed. If no such a node exists, the algorithm will first balance the left child subtree. Afterwards, the right child subtree will also be balanced. Otherwise, an extended basic transformation will be performed as follows.

   (i) Assume that each node in the join tree is associated with a triple of the form: (l, rt, S), where l is a label to indicate that the relation or the join operation represented by the corresponding node resides or will be executed at l (or in DB_l), rt represents the response time of the subtree rooted at this node and S stands for the estimated size of the result of the operation represented by this node. Then, a relation R residing at site l will be associated with (l, 0, $|R|$) if neither projection nor selection operation is performed over it at the beginning of the transformation process.

   (ii) Remove the original UAN and its right subtree from the current tree and mark its left child node to be the new UAN.

   (iii) Replace the LAN's right subtree with a new join node (corresponding to a join operation between the original UAN's right subtree and it). Let N_k and N_j be the roots of the left subtree of the LAN and the right subtree of the original UAN, respectively. Assume (l, rt, S) to be the triple being associated with the new join node N. Then, S can be obtained by using the method shown in Appendix A, while l and rt can be calculated as follows:

$$rt = \min\{\max\{rt_k, t_j + c_{l_j l_k} \cdot S_j\} + t_k, \max\{rt_j, t_k + c_{l_k l_j} \cdot S_k\} + t_j\} \qquad (5)$$

$$l = \begin{cases} l_k & \text{if } \max\{rt_k, t_j + c_{l_j l_k} \cdot S_j\} + t_k < \max\{rt_j, t_k + c_{l_k l_j} \cdot S_k\} + t_j \\ l_j & \text{otherwise.} \end{cases} \qquad (6)$$

   Further, the triples associated with the ancestor nodes of the new join node will be changed recursively in the same way as indicated by (5) and (6). That is, the triple associated with the LAN has to be changed in terms of the newly generated triple associated with N, which gives rise to the modification of the triple associated with the parent node of the LAN, and so on (see Fig. 7 for illustration).

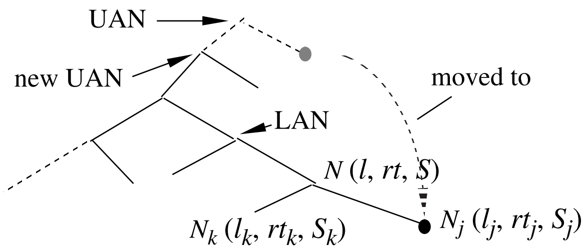


Fig. 7. Illustration of recursive modification of triples associated with ancestor nodes of N

Note that during a depth-first traversal of some subtree for a LAN, we may find several LANs, by which an *TIEB*-transformation can be performed. In this case, we choose one of them in terms of the following heuristics:

(1) If the right child node of a LAN has the join predicates only with the right child node of the original UAN, this LAN will be selected.

(2) If (1) is not the case, i.e., the right child node of each such LAN has join predicates with other relations, we select a LAN with the best time improvement.

The above algorithm works in a top-down fashion and can be used in the case that each participating relation resides in a different database. In the example below, we show how the triples associated with nodes can be calculated during a tree transformation process.

Example 3. We consider a multidatabase consisting of four local databases DB_i (i = 1, 2, 3, 4). Assume that the transmission cost matrix and the current load measurements are the same as shown in Fig. 3(a) and (b) and $((((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4)$ is the left deep join tree to be balanced, where R_i (i = 1, 2, 3, 4) represents a relation residing in DB_i. Then, the top-down transformation of this left deep join tree can be made as shown in Fig. 8.

First, we calculate the response time of each local database in terms of its actual load measurement:

$t_1 = s_1/(1 - \lambda_1 \cdot s_1) = 6.0/(1 - 0.2) = 7.5$ sec, $\quad t_2 = s_2/(1 - \lambda_2 \cdot s_2) = 4.0/(1 - 1/3) = 6.0$ sec,

$t_3 = s_3/(1 - \lambda_3 \cdot s_3) = 7.0/(1 - 0.35) = 9.33$ sec, $\quad t_4 = s_4/(1 - \lambda_4 \cdot s_4) = 8.0/(1 - 0.2) = 10.0$ sec.

Then, assuming that the sizes of R_i are $|R_1|$ = 10000, $|R_2|$ = 20000, $|R_3|$ = 5000 and $|R_4|$ = 8000, each GRT(i) can be computed according to formula (3):

$$\mathrm{GRT}(1) = c_{21} \cdot |R_2| + c_{31} \cdot |R_3| + c_{41} \cdot |R_4| + (4 - 1) \cdot t_1 = 4.0 + 1.5 + 2.4 + 3 \times 7.5 = 30.4,$$
$$\mathrm{GRT}(2) = c_{12} \cdot |R_1| + c_{32} \cdot |R_3| + c_{42} \cdot |R_4| + (4 - 1) \cdot t_2 = 2.0 + 0.5 + 1.6 + 3 \times 6.0 = 22.1,$$
$$\mathrm{GRT}(3) = c_{13} \cdot |R_1| + c_{23} \cdot |R_2| + c_{43} \cdot |R_4| + (4 - 1) \cdot t_3 = 3.0 + 2.0 + 0.8 + 3 \times 9.33 = 33.79,$$
$$\mathrm{GRT}(4) = c_{14} \cdot |R_1| + c_{24} \cdot |R_2| + c_{34} \cdot |R_3| + (4 - 1) \cdot t_4 = 3.0 + 4.0 + 0.5 + 3 \times 10.0 = 42.0.$$

Since GRT(2) is minimal, DB_2 will be chosen as the initial allocation (see Fig. 8(a)). If there is a join predicate between R_3 and R_4, the left deep join tree will be transformed into the bushy join tree as shown in Fig. 8(b).(In the figure, the estimated sizes of temporary relations are assumed for simplicity. But they can be calculated techniques shown in Appendix A.)

From the bushy join tree shown in Fig. 8(b), we see that due to the newly generated join node, the triple associated with the LAN will be changed from (DB_2, 14.5, 7000) to (DB_2, 17.03, 10000). (For this example, the LAN and the new UAN are accidentally identical.)

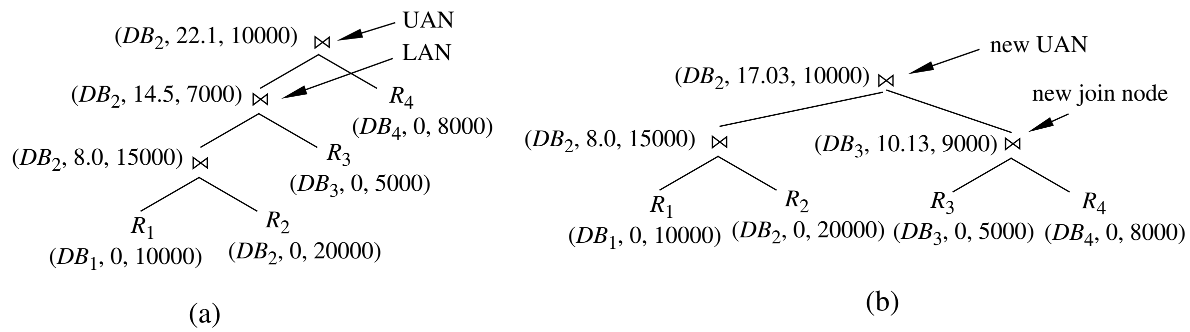


Fig. 8. Example of top-down transformation

Consequently, the optimal execution plan should be:

first step: transfer R_1 to DB_2 and at the same time transfer R_4 to DB_3; perform the corresponding join operations in DB_2 and DB_3, respectively.

second step: transfer the result of $R_3 \bowtie R_4$ from DB_3 to DB_2; then, perform the corresponding join operations in DB_2.

Although the top-down algorithm works in some cases, it suffers from several problems:

(1) In the case that more than one relations participating in the join tree reside in the same database, the response time of the newly generated join node can not be calculated in terms of formula (5);

(2) For the choice of a site, at which a new join operation will be performed, only those sites are considered, at which the results of its left or right subtree resides (see formula (6)); and

(3) The algorithm balances a node before its child nodes are balanced, which may hinder us from finding a more balanced join tree and gives rise to many time consuming modifications of triples associated with the ancestor nodes of a new join node.

In the following subsection, we discuss these problems and develop a new data structure to overcome the relevant drawbacks.

4.2. General top-down algorithm for constructing balanced bushy join trees

In this subsection, we address the first two problems listed above. First, we discuss an example, thereby showing why the above top-down algorithm is problematic when more than one relations residing in the same local database participates in a join tree.

Example 4. Consider Example 3 again. But this time we assume that R_2 and R_3 reside at the same site, namely, in DB_2. In this case, the initial left deep join tree is of the form as shown in Fig. 9(a). Using the top-down algorithm, the bushy join tree shown in Fig. 9(b) will be found. However, the nodes marked with (DB_2, 8.0, 15000) and with (DB_2, 6.8, 9000) can not be executed in parallel at all, since they are allocated to the same site. Therefore, the real response time should be 21.6 if we perform the join operations according to this plan. In fact, the bushy join tree shown in Fig. 9(c) is able to achieve a lower cost.

The reason the above algorithm fails to find the optimal bushy join tree in Example 4 is that the repeated labels make it difficult to make local choices of labels. To overcome this difficulty, we introduce the concept of the dynamic time table to record the overall response time of each local database system according to the currently generated bushy join tree during a transformation process. After a *TIEB*-transformation, it will be modified to reflect the new response times with respect to the new bushy join tree.

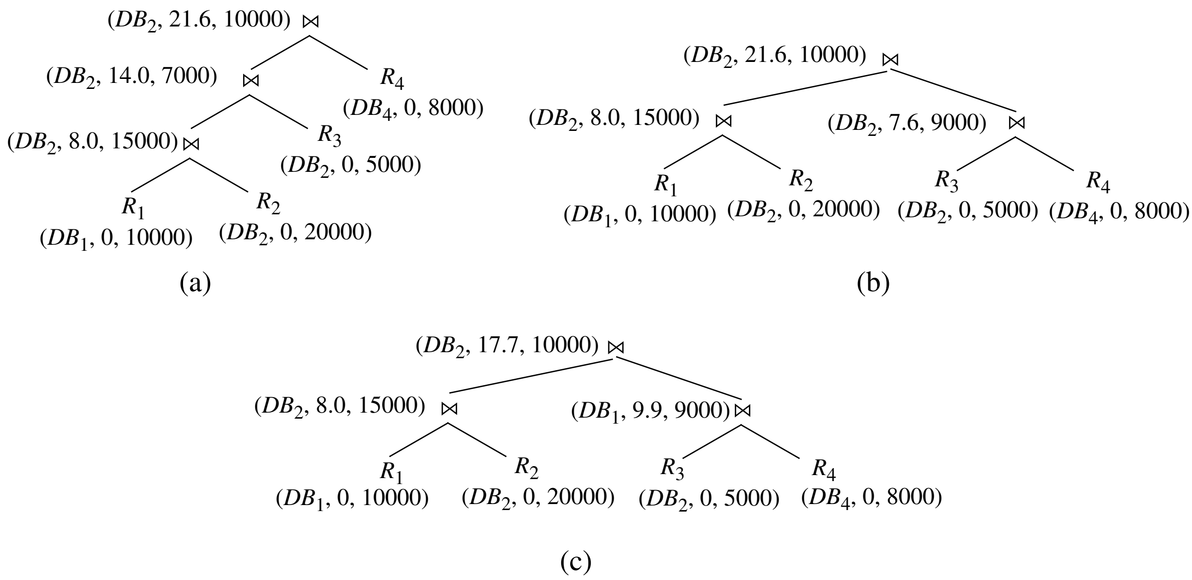


Fig. 9. Example of top-down transformation

Definition 3 A dynamic time table associated with a join tree T (denoted *dt-table*$_T$) is a set of pairs of the form: (DB, v), where DB is label representing a site and v is a list of the form: (v_1, ..., v_i, ...) with each v_i representing the response time of the i-th operation performed in DB w.r.t. T. Such a list is called a *temporary allocation* of DB and denoted *dt-table*$_T$(DB).

For example, associated with the left deep join tree shown in Fig. 9(a), we have a dynamic time table as shown in Fig. 10(a).

DB_i	response times
DB_1	( )
DB_2	(8.0, 14.0, 21.6)
DB_4	( )

(a)

DB_i	response times
DB_1	(9.9)
DB_2	(8.0, 17.7)
DB_4	( )

(b)

Fig. 10. Dynamic time table

If after some transformations the left deep join tree is changed to the form shown in Fig. 9(c), its dynamic time table will be of the form as shown in Fig. 10(b).

With the concept of the dynamic time table, we can develop a general top-down algorithm for balancing join trees in more general cases, thereby removing not only the first but also the second problem mentioned in the previous subsection. In contrast to the top-down algorithm discussed in that subsection, the general algorithm associates a quadruple of the form: (l, rt, S, i) with each node in the tree, where l, rt and S have the same meaning as for the above top-down algorithm (but calculated in a different way), and i is used to indicate that the node corresponds to the i-th operation performed at site l. In the following description of the general top-down algorithm, we will show the relevant computation methods.

General top-down algorithm description:

Let T be a left deep join tree or a temporary bushy join tree generated by the algorithm. For each UAN, we distinguish among three cases:

(1) If its left child node is a leaf node, the algorithm simply terminates (or returns) as the load of the UAN can not be redistributed and all other join nodes have already been processed.

(2) If the response time of the right subtree is about the same or greater than that of the left subtree, no *TIEB*-transformation will be applied to the UAN. But the recursive invocations of the algorithm on both the right and the left subtrees will be performed.

(3) If the response time of the left subtree is greater than that of the right subtree, traverse the subtree rooted at the left child node of the UAN to find a LAN, by which a *TIEB*-transformation can be performed. If no such a node exists, the algorithm will first balance the left child subtree. Afterwards, the right child subtree will also be balanced. Otherwise, a transformation will be performed as follows.

    (i) Change the dynamic time table as follows:

    for the LAN and all its ancestor nodes N **do**
    {let (l, rt, S, i) be the quadruple associated with N;
    Suppose that $dt\text{-}table_T(l)$ is of the form: $(v_1, ..., v_{n-1}, v_n)$;
    remove the i-th element from $(v_1, ..., v_{n-1}, v_n)$;
    }

    (ii) Remove the original UAN and its right subtree from the current tree and mark its left child node to be the new UAN.

    (iii) Replace the LAN's right subtree with the join node (representing the join operation over the original UAN's right subtree and it). Let N_k and N_j be the roots of the LAN's left subtree and the UAN's right subtree, respectively. Let (l_k, rt_k, S_k, p) be the quadruple associated with N_k and (l_j, rt_j, S_j, q) with N_j. Assume (l, rt, S, r) to be the quadruple being associated with the new join node N. Then, S can be obtained by using the techniques shown in Appendix A, while l, rt and i can be calculated as follows:

$$rt = \min_i \{\max\{dt\text{-}table_T(i)[last], rt_k + c_{l_k i} \cdot S_k, rt_j + c_{l_j i} \cdot S_j\} + t_i\} \quad (7)$$

$$l = p, \text{ if } rt = \max\{dt\text{-}table_T(p)[last], rt_k + c_{l_k p} \cdot S_k, rt_j + c_{l_j p} \cdot S_j\} + t_p\} \quad (8)$$

$$r = |dt\text{-}table_T(l)| + 1 \quad (9)$$

where $dt\text{-}table_T(i)[last]$ represents the last element of the list referenced by $dt\text{-}table_T(i)$, i.e., the response time of the last operation performed at site i and $|dt\text{-}table_T(l)|$ represents the number of elements in $dt\text{-}table_T(l)$. If $dt\text{-}table_T(i)$ is empty, $dt\text{-}table_T(i)[last]$ returns 0. We have formula (7) from an intuition that an operation can not be performed at a site until both operands have arrived at it and at the same it is idle.

As with the top-down algorithm discussed in the previous subsection, the heuristics may be used to find an optimal UAN. Additionally, the quadruple associated with the ancestor nodes of the newly generated join node have to be modified recursively. Simultaneously, the corresponding entries in $dt\text{-}table_T$ have to be modified, too. That is, $dt\text{-}table_T$ has to be changed to $dt\text{-}table_{T'}$ to reflect the new node allocation stand, where T' stands for the result of a *TIEB*-transformation applied to T.

Example 5. Consider Example 4 again. By using the general top-down algorithm, the left deep join tree shown in Fig. 11(a) can be transformed into the bushy join tree shown in Fig. 11(b).

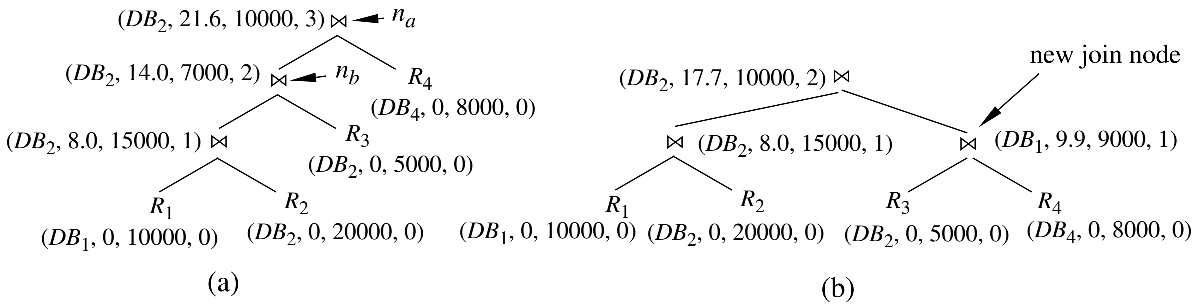


Fig. 11. Example of general top-down transformation

First, associated with the left deep join tree shown in Fig. 11(a), a dynamic time table as shown in Fig. 10(a) will be constructed. Then, by taking n_a and n_b as the UAN and the LAN, respectively, a new join node will be generated (see Fig. 11(b)), and the quadruple (l, rt, S, i) associated with it can be calculated in terms of formulas (7), (8) and (9) as follows:

site 1: $\max\{dt\text{-}table_T(1)[last], rt_{R3} + c_{21}\cdot|R_3|, rt_{R4} + c_{41}\cdot|R_4|\} + t_1$
$= \max\{0, 0 + 1.0, 0 + 2.4\} + 7.5 = 9.9.$

(*Note that before the calculation, the dynamic time table shown in Fig. 10(a) will be changed. See (3) in the above algorithm.*)

site 2: $\max\{dt\text{-}table_T(2)[last], rt_{R3} + c_{22}\cdot|R_3|, rt_{R4} + c_{42}\cdot|R_4|\} + t_2$
$= \max\{8.0, 0 + 0.0, 0 + 1.6\} + 6.0 = 14.0$

site 4; $\max\{dt\text{-}table_T(4)[last], rt_{R3} + c_{24}\cdot|R_3|, rt_{R4} + c_{44}\cdot|R_4|\} + t_4$
$= \max\{0, 0 + 1.0, 0 + 0.0\} + 10.0 = 10.1.$

Then, $rt = \min\{9.9, 14.0, 1.1\} = 9.9$, $l = 1$ and $i = |dt\text{-}table_T(1)| + 1 = 1$.

Accordingly, the dynamic time table will be changed to the form as shown in Fig. 12.

DB_i	response times
DB_1	(9.9)
DB_2	(8.0)
DB_4	( )

Fig. 12. Modification of dynamic time table

As demonstrated in this general top-down algorithm, the recursive modification will be caused for the quadruples associated with the ancestors of the newly generated join node. For this example, the quadruple (l', rt', S', i') associated with the parent of the new join node will be re-computed as follows:

site 1: $\max\{dt\text{-}table_T(1)[last], rt_{tm1} + c_{21}\cdot|tm_1|, rt_{tm2} + c_{11}\cdot|tm_2|\} + t_1$
$= \max\{9.9, 8.0 + 3.0, 9.9 + 0.0\} + 7.5 = 18.5,$

where tm_1 and tm_2 represent two temporary relations $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$, respectively.

site 2: $\max\{dt\text{-}table_T(2)[last], rt_{tm1} + c_{22}\cdot|tm_1|, rt_{tm2} + c_{12}\cdot|tm_2|\} + t_2$
$= \max\{8.0, 8.0 + 0.0, 9.9 + 1.8\} + 6.0 = 17.7.$

site 4; $\max\{dt\text{-}table_T(4)[last], rt_{tm1} + c_{24}\cdot|tm_1|, rt_{tm2} + c_{14}\cdot|tm_2|\} + t_4$

$= \max\{0, 8.0 + 3.0, 9.9 + 2.7\} + 10.0 = 22.6.$

Then, $rt' = \min\{18.5, 17.7, 22.6\} = 17.7$, $l' = 2$ and $i' = |dt\text{-}table_T(2)| + 1 = 2$. (See Fig. 11(b).)

From the above discussion, especially, formulas (7) and (8), we can see that a bushy join tree like that of Fig. 9(b) can never be generated due to the usage of dynamic time tables. Therefore, the first problem stated in the previous subsection is solved. Additionally, since the choice of a label for a newly generated join node is made by calculating (8), all sites will be considered for a node allocation. Thus, the second problem is also removed. In the following subsection, we discuss the third problem.

4.3. General algorithm for constructing balanced bushy join trees

To demonstrate the third problem stated in Subsection 4.1, let us see the left deep join tree shown in Fig. 13(a). As in Example 2, we do not take the communication costs into account here and at the same time assume that each local database has the same response time for clarity of explanation. But, to make the trace non-trivial, we assume that each local relation has an initial cost.

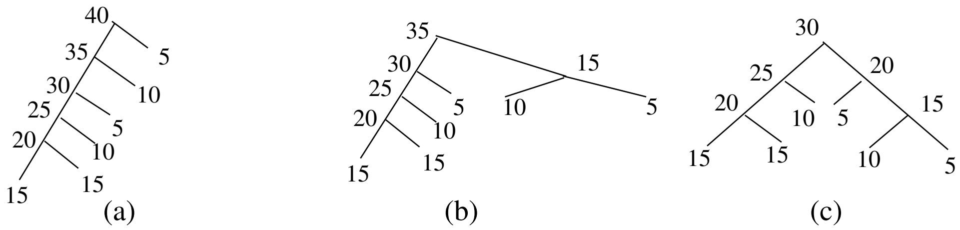


Fig. 13. Example of top-down transformation

After a *TIEB*-transformation step, it will be translated into the form as shown in Fig. 13(b). Since the left child subtree of the root is not balanced, this transformation makes a global balance impossible. Fig. 13(c) shows the final bushy join tree which is not well balanced (see Fig. 14(c) for a comparison.)

To overcome this insufficiency, we propose the following recursive algorithm which works better not only than the above general top-down strategy but also than the hybrid algorithm proposed in [DSD95]. The key idea of the algorithm is that before a tree is balanced, the left child subtree of its root will be first balanced. Then, we take the root as the UAN and traverse the subtree in a depth-first manner to find a LAN so that the *TIEB*-transformation can be executed.

Procedure *tree-transformation*(*LDT*, *root*)
(**LDT* stands for a left deep (or bushy) join tree; *root* represents its root.*)
begin
  let *LDT*' and *r-node* be the left child subtree and the right child node of *root*, respectively;
  let *a* be the root of *LDT*';
  call *tree-transformation*(*LDT*', *a*);   (*recursive call of *tree-transformation**)
  let *T* be the balanced bushy join tree found by *tree-transformation*(*LDT*', *a*);
  perform a depth-first traversal of *T* to find a time improving LAN in *T*;
  if *lan* is such a node found in *T*, for which the *TIEB*-transformation can be applied
    and at the same time, it is the best compared to the other transformable nodes;
  then {make the *TIEB*-transformation:
      remove *root* and its right child subtree from *LDT*;
      replace *lan*'s right (or left) child node *LS* with a new join node between *LS* and *r-node*;}
  return;
end

In the following, we trace a sample transformation to illustrate the major idea of the recursive algorithm.

Example 5. Fig. 14(a) is the same original left deep join tree as shown in Fig. 13(a), in which each relation node is labeled with its cost and each join node is labeled with the response time of the subtree rooted at that node.

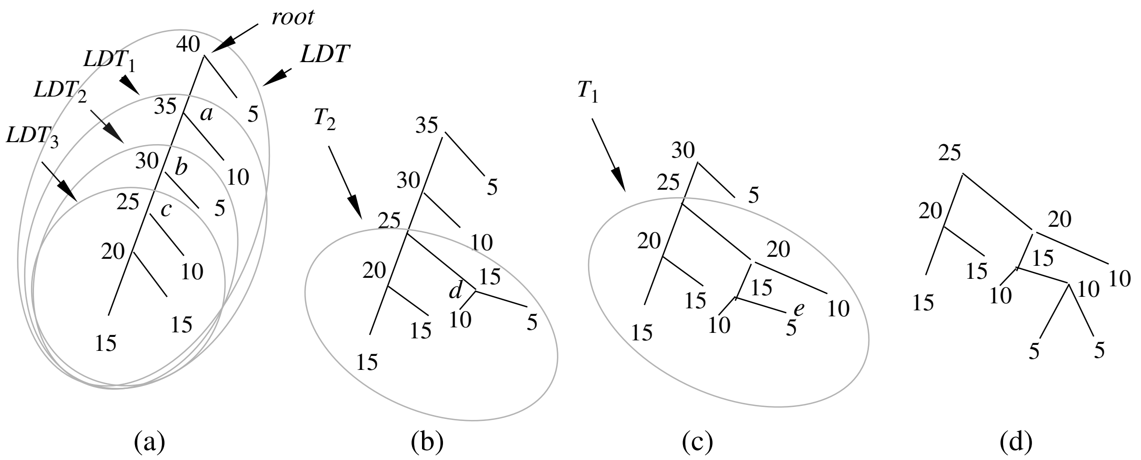


Fig. 14. Tracing a sample example

After the return from *tree-transformation*(LDT_3, *c*) to *tree-transformation*(LDT_2, *b*), the first transformation (as shown in Fig. 14(b)) happens and tree T_2 is the corresponding local balanced tree. Then, the control will be returned to *tree-transformation*(LDT_1, *a*). During the execution of it, tree T_2 will be searched and node *d* will be selected as the LAN (see Fig. 14(b) again). Consequently, tree T_1 will be constructed as shown in Fig. 14(c). Afterwards, the control will be returned to *tree-transformation*(*LDT*, *root*). During the execution of it, T_1 will be traversed and node *e* will be selected as the LAN (see Fig. 14(c) again). Lastly, the tree shown in Fig. 14(d) will be generated. It is a more balanced tree than that shown in Fig. 13(c).

From this example, we see that the result obtained by using *tree-transformation*( ) is better than that by using the top-down strategy. In fact, it is also better than the result by using the hybrid method of [DSD95]. It can be seen from the transformation process shown in Fig. 15. For this example, the hybrid algorithm behaves accidently like a top-down strategy (see Appendix B).

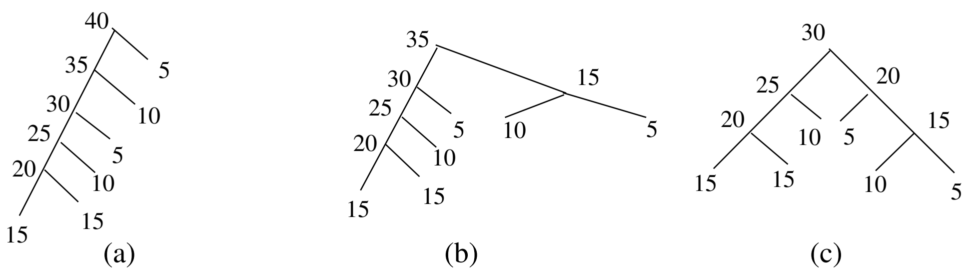


Fig. 15. Example of hybrid transformation (proposed by Du *et al.* [DSD95])

In addition, the time complexity of *tree-transformation*( ) is much better than that of the hybrid algorithm, by which O(n·log$n$) basic transformations are required, where n is the number of relations involved in the join tree. In contrast, *tree-transformation*( ) needs only n basic transformation, since there are only n recursive calls altogether and by each recursive call at most one *TIEB*-transformation may be performed. If we do not take the node allocation into account, a transformation step in our method will take the same time as that in [DSD95]. However, the complexity of the tree traversal in our method is

$O(n^2)$ time. This is the case as $O(n)$ nodes will be accessed by each depth-first search of some subtree and there are n times of subtree traversals in the worst case. But the complexity of the tree traversal of the method proposed by Du *et al*. is also on the order of $O(n^2)$ (see [DSD94]).

5. Conclusion

In this paper, a new algorithm for balancing left deep join trees is proposed. It works not only according to the communication costs, but also to the load measurements. In this way, both the tree balance and node allocation can be obtained. Furthermore, the basic transformation step used in [DSD95] is extended to a more powerful transformation step so that a more balanced join tree can be found using our method. In addition, a simple queuing analysis has been utilized to calculated the response times of local database systems in terms of their actual load states.

Appendix A. Statistical estimation of temporary relations

In our system, the following informations are maintained for the size estimation of temporary relations:

width(r): the size of a tuple in a relation r;

tuple(r): the number of tuples in a relation r;

att_num(v, r, A): the number of tuples with value v of attribute A in relation r;

Key(v, r, A): the boolean value to indicate the presence of the value v for the key attribute A in relation r. The value is "1" if v is present in r and "0" otherwise;

M(A, B, r): the association matrices maintaining the information w.r.t. relation tuples for which attributes have specific values. $M(A, B, r)(v_i, v_j)$ denotes the number of tuples in r with attribute values $A = v_i$ and $B = v_j$.

In terms of such informations, the sizes of temporary relations can be estimated as follows.

Selection. Let $\sigma_{A\,\text{<comp>}\,\text{Constant}}(r)$ be the selection operation where one is choosing tuples from relation r for which attribute A satisfies the condition "A <comp> Constant". Here, <comp> $\in \{=, <, \leq, >, \geq, \neq\}$. The size estimation of attribute A in the result *output-r* is calculated as follows.

If A is a non-key attribute in relation r, then

$$att\_num(v, output\text{-}r, A) = \begin{cases} att\_num(v, r, A) & \text{if } v \text{ <comp> Constant is } true, \\ 0 & \text{otherwise.} \end{cases}$$

If A is a key attribute in relation r, then

$$Key(v, output\text{-}r, A) = \begin{cases} Key(v, r, A) & \text{if } v \text{ <comp> Constant is } true, \\ 0 & \text{otherwise.} \end{cases}$$

Projection. Let $\pi_{A,\,B,\,\ldots}(r)$ be the projection operation on relation r. The estimation size of attribute A in *output-r* can be derived as follows:

$$att\_num(v, output\text{-}r, A) = att\_num(v, r, A).$$

Join. Let r_1 and r_2 be two relations being joined ($r_1 \underset{r_1.A=r_2.A}{\bowtie} r_2$) and *output-r* be the result of the join operation. We distinguish among three cases.

case 1: If both $r_1.A$ and $r_2.A$ are non-key attributes, then the size estimation of A in the result *output-r* can be derived as follows:

$$att\_num(v, output\text{-}r, A) = att\_num(v, r_1, A) \times att\_num(v, r_2, A).$$

case 2: If $r_1.A$ is the key attribute while $r_2.A$ is the non-key attribute, the size estimation of A in the result *output-r* can be computed by:

$$att\_num(v, output\text{-}r, A) = Key(v, r_1, A) \times att\_num(v, r_2, A).$$

case 3: If both $r_1.A$ and $r_2.A$ are key attributes, then attribute A in *output-r* will also be the key attribute of *output-r*. This information about A in *output-r* is derived as follows:

$$Key(v, output\text{-}r, A) = Key(v, r_1, A) \times Key(v, r_2, A).$$

Lastly, after a selection or join operation, the estimation w.r.t. the association matrices $M(A, B, r)$ can be derived in two steps:

first step: **for** all $v \in$ domain(B) **do**
    {$att\_num(v, output\text{-}r, B) := 0;$}

second step: **for** all $v_i \in$ dommain(B) **do**
    for v_j such that $att\_num(v, r, A) \neq 0$ **do**
        {$att\_num(v_i, output\text{-}r, B) := att\_num(v_i, output\text{-}r, B) + M(A, B, r)(v_j, v_i);$
        $M(A, B, output\text{-}r)(v_i, v_j) := M(A, B, r)(v_i, v_j);$}.

Appendix B. Hybrid algorithm proposed by Du *et al.*

In this appendix, we list simply the hybrid algorithm of [DSD95] for comparison purpose. The algorithm takes a left deep join tree as the input, which is identified by its root and its leftmost leaf node, and consists of two processes: a bottom-up and top-down process. In the following description, N_1 represents the grandparent of the left most leaf node of an input tree, while N_2 represents its root. In addition, the existence of the following functions with the straightforward meanings are assumed: left-child( ), right-child( ), sibling( ), response-time( ), parent( ), leftmost-leaf( ) and find-LAN( ).

Algorithm *Hbrid*(N_1, N_2)
begin
  repeat
    L_1 := left-child(N_1); R_1 := right-child(N_1);
    L_2 := left-child(N_2); R_2 := right-child(N_2);
    if response-time(L_1) ≤ response-time(R_2) **then** (*bottom-up*)
      {n := find-LAN(N_1);
       if $n \neq null$ **then**
         {make a basic transformation with N_1 being the UAN and n being the LAN;
         if response-time(*njn*) > response-time(sibling(*njn*)) **then** (**njn* represents the new join node.*)
           Hbrid(parent(leftmost-leaf(*njn*)), *njn*);
         N_1 := paren(N_1);
       }
    else
      {m := find-LAN(N_2); (*top-down*)
       if $m \neq null$ **then**

{make a basic transformation with N_2 being the UAN and *m* being the LAN;
if response-time(*njn*) > response-time(sibling(*njn*)) **then**
Hbrid(parent(leftmost-leaf(*njn*)), *njn*);
}
until $N_1 = N_2$
end

In the algorithm, two processes runs alternately: one proceeds top-down from the root, and the other proceeds bottom-up from the leaf. At any given time, at most one UAN is chosen according to the following criterion: N_1 is chosen if the response time of its left child subtree is smaller than that of N_2's right child subtree, and N_2 is chosen otherwise. The algorithm stops when the two UANs meet.

References

BOT86 Y. Breitbart, P. Olson, and G. Thompsom, "Database integration in a distributed heterogeneous database system," in *Proc. 2nd IEEE Conf. Data Eng.*, 1986, pp. 301 - 310.

CB96 Y. Chen and W. Benn, "On the Query Translation in Federative Relational Databases", in *Proc. of 7th Int. DEXA Workshop*, Zurich, Switzerland, Sept. 1996, pp. 491 - 498.

Co81 R. Cooper, *Introduction to Queuing Theory*, 2nd ed. New York: North Holland, 1981.

CW93 S. Ceri and J. Widom, "Managing Semantic Heterogeneity with Production Rules and Persistent Queues", in *Proc. 19th Int. VLDB Conference*, Dublin, Ireland, 1993, pp. 108 -119.

DSD94 W. Du, M. Shan and U. Dayal, "Reducing Multidatabase Query Response Time by Tree Balancing", *DTD Technical Report*, Hewlett-Packard Labs., 1994.

DSD95 W. Du, M. Shan and U. Dayal, "Reducing Multidatabase Query Response Time by Tree Balancing", in *Proc. 15th Int. ACM SIGMOD Conference on Management of Data*, San Jose, california, 1995, pp. 293 -303.

Du94 Y. Dupont, "Resolving Fragmentation conflicts schema integration", in *Proc. 13th Int. Conf. on the Entity-Relationship Approach*, Manchester, United Kingdom, Dec. 1994, pp. 513 - 532.

ETB96 C.J. Egyhazy, K.P. Triantis and B. Bhasker, "A Query Processing Algorithm for a System of Heterogeneous Distributed Databases", *Int. Journal of Distributed and Parallel Databases*, 4, 49 - 79, Dec. 1996.

HLM94 G. Harhalakis, C.P. Lin, L. Mark and P.R. Muro-Medrano, "Implementation of Rule-based Information Systems for Integrated Manufacturing", *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 6, 892 - 908, Dec. 1994.

HS93 W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS", *Int. Journal of Distributed and Parallel Databases,* Vol. 1, No. 1, 9 - 32, 1993.

KC93 J-L. Koh and Arbee L.P. Chen, "Integration of Heterogeneous Object schemas", in *Proc. 12th Int. Conf. on the Entity-Relationship Approach*, Arlington, Texas, USA, Dec. 1993, pp. 297 - 314.

KLK91 R. Krishnamurthy, W. Litwin and W. kent, "Language features for interoperability of databases with schematic discrepancies", *In Proc. of the ACM SIGMOD Conf. on Management of Data*, pp. 40-49, DEnver, Colorado, May 1991.

Jo93 P. Johannesson, "Using Conceptual Graph Theory to Support Schema Integration", in *Proc. 12th Int. Conf. on the Entity-Relationship Approach*, Arlington, Texas, USA, Dec. 1993, pp. 283 - 296.

LA86 W. Litwin and A. Abdellatif, "Multidatabase interoperability," *IEEE Comput. mag.*, vol. 19, No. 12, pp. 10 - 18, 1986.

LBT93 A. Lefebvre, P. Bernus and R.W. Topor, "Querying heterogeneous databases: a case study", in *Proc.*

3th Australian Database Conf., Brisbane, February 1993, pp. 186-198.

LHSC95 E. Lim, S. Hwang, J. Srivastava, D. Clements and M. Ganesh, "Myriad: design and implementation of a federated database prototype", *Software-Practice and Experience*, Vol. 25(5), 533 - 562, May 1995.

LOG93 H. Lu, B. Ooi and C. Goh, "Multidatabase Query Optimization: Issues and Solutions", *In Proc. of 3th Int. Workshop on Research Issues in Data Engineering*, pp. 137 - 143, Vienna, Austria, April 1993.

RPRG94 M.P. Reddy, B.E. Prasad, P.G. Reddy, and A. Gupta, "A methodology for integration of heterogeneous databases," *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 6, 920 - 933, Dec. 1994.

SK92 W. Sull and R.L. Kashyap, "A Self-organizing knowledge representation schema for extensible heterogeneous information environment," *IEEE Trans. on Knowledge and Data Engineering*, vol. 4, No. 2, 185 - 191, April 1992.

SPD92 S. Spaccapietra and P. Parent, and Yann Dupont, "Model independent assertions for integration of heterogeneous schemas", *VLDB Journal*, No. 1, pp. 81 - 26, 1992.

SP94 S. Spaccapietra and P. Parent, "View integration: a step forward in solving structural conflicts", *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 2, 258 - 274, April 1994.