Definition 1 (*assumed values*) An assumed value (for some variable) is either of the form $\underline{\alpha X}$ or $\underline{\in S}$, where X is either a variable or a constant, $\alpha \in \{=, <, \leq, >, \geq, \neq\}$ and S represents a set of constants. For example, $> x$, $= c$ and $\in \{c_1, c_2, ..., c_n\}$ are three assumed values.

Definition 2 (*extended substitutions*) An extended substitution (ES) is a finite set of the form $\{x_1/v_1, ..., x_l/v_l\}$, where x_i ($i = 1, ..., l$) is a variable and v_i ($i = 1, ..., l$) is a set of pairs of the form (*Prop*, *v*), where *Prop* $\in$ {*p*, *s*, *ni*, *V*, _} (where, "_" means "do not care") and *v* is an assumed value as defined above or "_". In contrast to the traditional substitution concept, the variables $x_1, ..., x_l$ may not be distinct. Each element x_i/v_i is called a binding for x_i, and a variable may have several bindings.

For example, $\delta = \{x/\{(p, \_)\}, y/\{(p, \_), (\_, \in \{car_1', car_2', ..., car_m'\})\}, z/\{(s, >10000)\}$ is a legal ES. Alternatively, this ES can also be written as $\{x/(p, \_), y/(p, \_), y/(\_, \in \{car_1', car_2', ..., car_m'\}), z/(s, >10000)\}$.

4.4 Translation of Simple Algebra Expressions: $\pi(\sigma(R))$

Translation can be pictorially illustrated as shown in Fig. 4.

rule <*antecedent-part*>	$\Rightarrow$	<*consequent-part*> + ES
matching $\uparrow$		$\downarrow$ *derivation*
an algebra expr.	$\rightarrow$	*a set of new algebra expr.*

Fig. 4. Illustration of the query translation process.

In the following, we discuss this process in detail.

Essentially, this process employs two functions. With the first function, we generate an ES by matching the algebra expression to be translated with the corresponding rule's antecedent part. With the second function, we derive a set of new algebra expressions in terms of the ES and the rule's consequent part. These two functions can be defined as follows:

the first function: *substi-production*: $\mathbf{P} \times \mathbf{A} \rightarrow \mathbf{S}$,
the second function: *expression-production*: $\mathbf{P} \times \mathbf{S} \rightarrow \mathbf{A}$,

where **P**, **A** and **S** represent the set of all c-expressions, the set of all algebra expressions and the set of all extended substitutions, respectively.

Obviously, the matching algorithm used in Prolog [29] can not be employed for our purpose, and a bit of modification is required so that not only the assumed values of a variable, but also more information associated with it can be evaluated. As we will see in the following algorithm (for *substi-production*), such informations can be obtained by doing a simple analysis of the algebra expression to be translated (see lines 2-5). In the algorithm, the following definitions are used:

- *assumedValue*($A\ \alpha\ B, T$) returns an assumed value of the form αX, where $A\ \alpha\ B$ is a select condition, T is an RST or a c-expression and $\alpha \in \{=, <, \leq, >, \geq, \neq\}$. X is a constant "c" if B = c, or a variable x if B: x is an attribute descriptor in T.

JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 463-497 (2000)

A Systematic Method for Query Evaluation in Distributed Heterogeneous Databases

YANGJUN CHEN
IPSI Institute, GMD GmbH
64293 Darmstadt, Germany
E-mail: yangjun@darmstadt.gmd.de

In this article, we consider the query evaluation problem in relational multidatabases and develop a method for generating optimal plans for queries submitted to such a system. Three aspects will be discussed: *query transformation*, *join tree balance* and *node allocation*. For query translation, the concept of *relation structure terms* (RST) is introduced. Based on RSTs, we can transform a query into another form automatically by constructing derivation rules for them. Further, we extend the approach for balancing a join tree proposed by Du *et al.* so that more balanced join trees can be obtained. Lastly, we present the concept of *dynamic time tables* for performing node allocation in a dynamic programming manner.

Keywords: heterogeneous databases, join tree balance, relation structure terms, dynamic programming, dynamic time tables

1. INTRODUCTION

A multidatabase system (MDBS) is a database system which integrates pre-existing databases, called component local database systems (LDBSs), to support global applications accessing data at more than one LDB. In such a system, as in a distributed database (DDBS), query optimization is very important but quite different from that in the case of DDBSs. First, due to the heterogeneity of component databases, a query submitted to a multidatabase has to be decomposed and translated so that it can be evaluated against different, possibly heterogeneous, local databases (see [2, 5, 6, 22, 26, 34]). 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 in order to generate an optimal execution plan for a given query. Although in a distributed database load measurement is also considered, it is handled in a different way. That is, the load states can be changed by means of load sharing. For example, one can distribute the system workload from heavily loaded nodes to lightly loaded nodes in a system. But this is not possible in multidatabases due to the autonomy of the local databases.

In this article, we confine ourselves to investigating how to generate optimal execution plans for queries and develop algorithms for this issue, by means of which both join tree balance and the node allocation can be achieved. Many theoretical solutions have been proposed, and different implementations have provided varying approaches to these problems [8, 14, 16, 17, 31, 37]. In [14], a hybrid algorithm was developed to transform a left deep join tree into a balanced bushy join tree based on a simple cost analysis, so that the overall response time can be reduced. However, how to allocate a (join) operation to a local

Received December 28, 1998; revised May 24, 1999; accepted July 19, 1999.
Communicated by Wei-Pang Yang.

database system was untouched. In [8], a new push-down method is proposed without taking the costs of information transmission among different sources into account. In [17], a third approach was proposed to finding both the node allocation and the sequence of join operations using linear programming. But the balance problem was not taken into account there. Moreover, in that method, only the communication costs are considered (see Appendix B of [17]), and the load measurements are not addressed at all. Similarly, in [37], a fourth method was suggested for finding an optimal join sequence with node allocation from a query graph in an exhaustive but overhead-distributed manner. But tree balance was not considered. A further important approach was proposed by Evrendilek *et al*. [16]. This approach works in a three-phase fashion. In the first phase, static node allocation of decomposed subqueries is conducted. In the second phase, a join sequence of subqueries is generated in terms of weight functions, by means of which both the cost and the selectivity of join operations are considered. In the third phase, a bush join tree is produced using a cost recurrence relation calculated in a bottom-up manner over the nodes of the join tree being constructed. The algorithm requires $O(n^3)$ time, where n is the number of nodes (or subqueries which are executed at local sites in the first phase) involved in the global query. From exact analysis of the cost recurrence relation given in [16], however, we can see that only the "static" load states of local machines are considered for generating a balanced join tree. That is, the changes of the load states of local sites during the join operations themselves are not considered. Therefore, a generated join tree produced in terms of this cost recurrence relation may be unbalanced since a site may get another join operation after some join operation has been assigned to it, without being aware that the site may become heavily loaded as the join operations proceed. Of course, we can assign more than one join operation to a site if after the assignment of one or more join operations, its work load remains low compared to the other sites. But this can be done only according to reasonable cost estimation.

In contrast, in our method, both tree balance and (dynamic) node allocation are considered. First, we propose a method for translating local queries automatically. Then, we refine the algorithm proposed by Du *et al*. [14] to find a more balanced tree from a left deep join tree by extending the basic transformation employed by them. Next, we introduce the concept of a dynamic time table and devise an efficient algorithm for node allocation, based on both the communication and workloads of local database systems. In particular, dynamic changes of the workloads are considered by maintaining a dynamic time table for the current join operations. We argue that the load measurement and its dynamic changes are important for the query evaluation in a multidatabase and affect the performance significantly. We also note that the load measurement will not damage the autonomy since no interference with local databases occurs. The system enquires about and changes the load states of local databases only by issuing queries and (decomposed) subqueries as local users. Due to its autonomy, a local system has the right to reject cooperation. In this case, the next replicate (residing in another local system) will be selected if it is available. Then, the planner will be invoked once again with the rejecting local system excluded. We can maintain a list of the names of the replicates for each relation in the global system, sorted according to the average response time of the local systems in which they reside. The node allocation algorithm examines each element of the list until a plan is generated or aborted. If a local system becomes very busy during the evaluation process and its response time greatly increases, we can treat it as a rejecting one, getting it off, select another replicate and execute the planner again. The time complexity of our algorithm is bounded by $O(n^2)$.

The remainder of this article is organized as follows. In section 2, we present our system architecture to provide background information for the subsequent discussion. Then, in section 3, we give an overview of the plan generation process for optimizing a query evaluation in a multidatabase environment. In section 4, an automatic method for translating queries is discussed in some detail. In section 5, we present our strategy for balancing a join bushy tree, based on the basic transformation step proposed in [14], which is extended to a more powerful transformation step in our implementation. In section 6, we discuss the communication costs and the load measurements and give an efficient method for allocating operations to sites. Section 7 is a short conclusion.

2. SYSTEM ARCHITECTURE

In this section, we present our system architecture, which consists of three-layers: an FSM-client, FSM and FSM-agents as shown in Fig. 1 (where FSM stands for "Federated System Manager"). The task of the FSM-client layer is application management as a suite of application tools is provided which enables users and DBAs to access the system. The FSM layer is responsible for merging potentially conflicting local databases, defining global schemas, and conducting global query treatment. In addition, a centralized management is supported in this layer. The FSM-agents layer corresponds to local system management and addresses all issues w.r.t. schema translation, local transactions and query processing.

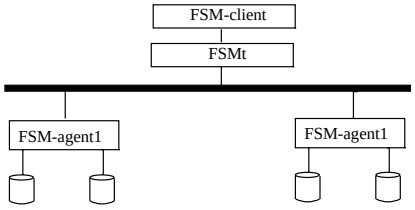


Fig. 1. System architecture.

Based on this architecture, each component database is installed in some FSM-agent and must be registered in the FSM. Then, for a component relational database, each attribute value will be implicitly prefixed with a string of the form:

<FSM-agent name>.<database system name>.
<database name>.<relation name>.<attribute name>

where "." denotes string concatenation. For example, through *FSM-agent*1.*informix*. *PatientDB.patient - records.name*, the attribute "name" from thes relation "patient-records" in a database named "PatientDB" will be referenced.

For ease of exposition, in the following, we discuss query translation in a simple setting in which each local database involved in a query is relational and decomposition of "global" relations occurs.

3. PLAN GENERATION FOR QUERY EVALUATION

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. Following the traditional methods for estimating costs [17], we can develop an efficient algorithm for constructing execution plans for any such query submitted to a (relational) multidatabase. The algorithm can simply be described as shown in Fig. 2.

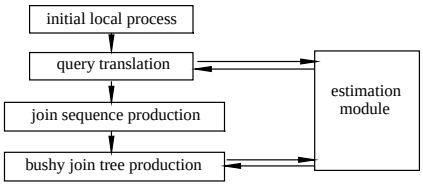


Fig. 2. Process of generating an execution plan for a given query.

The algorithm mainly consists of four phases:
– the initial local process,
– local query translation
– join sequence production, and
– bushy join tree production.

Additionally, an estimation module is used to derive the sizes of temporary relations based on the approach developed in [17].

(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. In addition, if no schema conflict exists, the initial local process will perform the local projection operations for each relation over the previously identified attributes in order to reduce the size of each relation.

(2) *Query translation.* After the initial local process, the query is translated into a locally computable form. This can be done automatically if the relevant metadata for specifying semantic conflicts are built correctly. We distinguish between two types of semantic conflicts, i.e., schema conflict and data conflict. By schema conflict, we mean that an attribute value of a global relation corresponds to an attribute name or a relation name in the corresponding local database or *vice versa*. To declare such a conflict, a rule-based method is developed, a detailed description of which can be found in section 4. For resolving data conflict, we associate each attribute A of a global relation with a set of mapping functions, $F^{A}_{DB_i,B}$, (i = 1, ..., n), with each being used for the value correspondences of attribute A and attribute B from the local database DB_i. An $F^{A}_{DB_i,B}$ may be a simple string "default," indicating that all actual values of B comprise a subset of A; a set of triples of the form (a, b; χ) indicating that a of A corresponds to b of B to degree

$\chi \in [0, 1]$ (where χ is used to support the fuzzy set concept; see [4] for a detailed discussion); or a simple function of the form $y = f(x)$ (such as $y = 2.5 \cdot x$), where y and x are variables ranging over the domains of A and B, respectively. In addition, the size estimation procedure given in [17] for the selection and projection operations is used to estimate the sizes of the local transformed relations.

(3) *Join sequence production.* After the above two processes, we determine the optimal order for the temporary relations resulting from these processes. This can be done in the same way as discussed in [9, 10, 20] or by using A* search as described in [42] to avoid cartesian products and to minimize the size of intermediate relations. However, the algorithm proposed in [10] for generating bushy join trees directly from a query graph is not used in our system based on the following considerations:

(i) The bushy join tree generated by [10] is not balanced. Therefore, an extra process is needed to balance such a tree just as for a left deep join tree. (Note that in [10], tree balance is accomplished by mean of processor allocation, which is completely unsuitable for multidatabases.)

(ii) The time complexity of the algorithm for finding such a bushy join tree is $O(n \cdot e)$, where n and e are the numbers of relations and corresponding joins involved in the original query, while the time complexity of the algorithm for finding a left deep join tree is $O(n^2)$ (see [10]). As we will see later, a recursive algorithm for transforming a left deep join tree into a balanced bushy join one requires only $O(n^2)$ time. Therefore, theoretically, the strategy developed based on the transformation of left deep join trees will have a better time complexity than does the algorithm based on the approach for generating bushy join trees [10].

(4) *Bushy join tree production.* In principle, the method proposed by Du *et al.* [14] can be extended to construct balanced bushy join trees (from optimal join sequences) in the simple case where each participating relation resides in a different database. The only difference consists in the load measurement and more exact calculation of communication costs as well as a labeling strategy for the node allocation. However, in the case where more than one relation resides in the same database, a more sophisticated method is needed to achieve an optimal plan. We addressed these issues in section 6 in detail.

4. LOCAL QUERY TRANSLATION

Let the global query be of the form:

$$\pi_{A_1 \ldots A_l}(\sigma_{sc_1 \ldots sc_m}(R_1 \bowtie R_2 \ldots R_n \bowtie R_{n+1})),$$

where $A_1, \ldots, A_l$ are attributes appearing in $R_1, \ldots,$ and R_{n+1}, and sc_i ($i = 1, \ldots, m$) is of the form $B \; \alpha \; v$, or $B \; \alpha \; C$ (called the selection condition) with B and C representing the attributes in $R_1, \ldots,$ and R_{n+1}, v representing a constant and α being one of the operators $\{=, <, \leq, >, \geq, \neq\}$. After the initial step, such a global query can be transformed into the following form: $\pi_c(\sigma_d(\pi_{c1}(\sigma_{d1}(S_1^{j_i})) \bowtie \ldots \pi_{ci}(\sigma_{di}(S_i^{j_i})) \bowtie \ldots))$, where c and d (as well as c_i and d_i) rep-

resent the corresponding projection attributes and selection conditions, respectively, $S_i^{j_i}$ represents a join sequence of the form $R_{i_1} \bowtie R_{i_2} \ldots \bowtie R_{i_j}$, and each R_{i_m} corresponds to a local relation at site j_i. Obviously, each $\pi_{ci}(\sigma_{di}(S_i^{j_i}))$ can be further transformed into the form $\pi_g(\sigma_f(\pi_{g1}(\sigma_{f1}(R_{i_1})) \bowtie \ldots \pi_{gl}(\sigma_{fl}(R_{i_l}))))$, and each $\pi_{gt}(\sigma_{ft}(R_{i_t}))$ has to be translated so that it can be evaluated locally. To do this, the schema and data conflicts of local databases have to be recognized and resolved. Schema conflicts can be eliminated by invoking an inference engine while data conflicts can be removed by establishing a series of mappings in advance. In the following, we discuss schema conflicts in detail.

From [22, 26, 27, 32], there are five types of schema conflicts as shown in Fig. 3. They are:

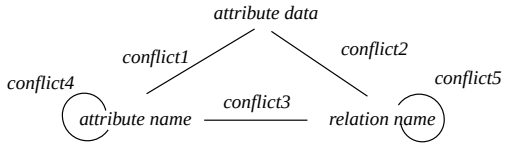


Fig. 3. Illustration for schema conflicts.

1. when an attribute value in one database appears as a relation name in another database,
2. when an attribute value in one database appears as a relation name in another database,
3. when an attribute name in one database appears as a relation name in another database,
4. when an attribute name in one database corresponds to several different attribute names in another database, and
5. when a relation name in one database corresponds to several different relation names in another database.

In the following, we mainly focus on the first three types of semantic conflicts for simplicity. (But as we will see later, types 4 and 5 can be tackled in the same way as we tackle types 1, 2 and 3.) First, we consider three local databases storing car prices.

Example 1 Consider three databases representing information about car prices. The local schemes are as follows:

DB_1: R_1 (time, car, price),
DB_2: R_2 (time, car1, ..., carn),
DB_3: car_1' (time, price),
... ...
car_m' (time, price).

In DB_1, there is one single relation, with one tuple per month and car, storing the actual price. In DB_2, there is one single relation, with one tuple per month and one attribute per car, named by the car name and storing its price. Finally, DB_3 has one relation per car, named by the car name; each relation has one tuple per month storing the price.

If we want to integrate these three databases, and if the global schema is chosen to be the same as R_1 (in DB_1), then a query of the form $\pi_{time,car}(\sigma_{price > 10000}(R_1))$ against the global schema has to be translated into the following form:

for each $y \in \{car_1', ..., car_m'\}$ **do**
$\{\pi_{time}(\sigma_{price>10000}(y))\}$

so that it can be evaluated in DB_3.

Several approaches to this problem have been proposed in the literature [22, 26]. In [22], a higher order language was suggested to deal with schema discrepancies and to issue higher order queries (queries containing variables which range over both data and metadata). However, how to translate a query automatically when data and schema conflicts occur was not considered. In [26], an F-logic-based algorithm was developed; but no clear translation rules were defined there.

To overcome this difficulty, we propose a method for performing the task in a logical fashion.

In order to develop a mechanism to perform query translation automatically, we must introduce the concept of *relation structure terms* (RST) in order to capture higher-order information w.r.t. a local database. Then, for RSTs w.r.t. some heterogeneous databases, we define a set of derivation rules to specify the semantic conflicts among them.

4.1 Relation Structure Terms

In our system, an RST is defined as follows:

$$[re_{\{R_1, ..., R_m\}} \mid a_1: x_1, a_2: x_2, ..., a_l: x_l, y: z_{\{A_1, ..., A_n\}}],$$

where *re* is a variable ranging over the relation name set $\{R_1, ..., R_m\}$, y is a variable ranging over the attribute name set $\{A_1, ..., A_n\}$, $x_1, ..., x_l$ and z are variables ranging over respective attribute values, and $a_1, ..., a_l$ are attribute names. In the above, each pair of the form $a_i: x_i$ ($i = 1, ..., l$) or $y: z$ is called an attribute descriptor. Obviously, such an RST can be used to represent either a collection of relations possessing the same structure or a part of the structure of a relation. For example, $[re_{\{car'_1, ..., car'_m\}} \mid$ time: x, price: $y]$ represents any relation in DB_3 while an RST of the form $[re_{\{R_2\}} \mid$ time: x, y: $z_{\{car_1, ..., car_n\}}]$ (or simply $[R_2 \mid$ time: x, y: $z_{\{car_1, ..., car_n\}}]$) represents a part of the structure of R_2 with the form: R_2(time,..., car_i, ...) in DB_2. Since such a structure allows variables for relation names and attribute names, it can be regarded as a higher order predicate quantifying both data and metadata. When the variables (of an RST) appearing in the relation name position and attribute name positions are all instantiated to constants, it degenerates to a first-order predicate. For example, $[R_1 \mid$ time: x_1, car: x_2, price: $x_3]$ is a first-order predicate quantifying tuples of R_1.

The purpose of RSTs is to formalize metadata information. Therefore, they can be used to declare schematic discrepancies. In fact, by combining a number of RSTs into a deductive rule, we can specify exactly some semantic correspondences of heterogeneous local databases.

For convenience, an RST can be simply written as $[re_{\{R_1, ..., R_m\}} \mid a_1: x_1, a_2: x_2, ..., a_l: x_l, y: z_{\{A_1, ..., A_n\}}]$ if the possible confusion can be avoided by the context.

4.2 Derivation Rules

For RSTs, we can define derivation rules in a standard way as implicitly universally quantified statements of the form: $\gamma_1 \,\&\, \gamma_2 \,...\, \&\, \gamma_l \Leftarrow \tau_1 \,\&\, \tau_2 \,...\, \&\, \tau_k$, where both γ_i's and τ_j's are (partly) instantiated RSTs or normal predicates of first-order logic. For example, using the following two rules,

$r_{DB_1\text{-}DB_3}$: $[y \mid \text{time: } x, \text{ price: } z] \Leftarrow [R_1 \mid \text{time: } x, \text{ car: } y, \text{ price: } z], y \in \{car_1', car_2', ..., car_m'\}$,
$r_{DB_3\text{-}DB_1}$: $[R_1 \mid \text{time: } x, \text{ car: } y, \text{ price: } z] \Leftarrow [y \mid \text{time: } x, \text{ price: } z], y \in \{car_1'', car_2'', ..., car_l''\}$,

the semantic correspondence between DB_1 and DB_3 can be specified. (Note that in $r_{DB_3\text{-}DB_1}$, car_1'', car_2'', ..., and car_l'' are the attribute values of "car" in R_1.)

Similarly, using the following rules, we can establish the semantic relationship between DB_1 and DB_2:

$r_{DB_1\text{-}DB_2}$: $[R_2 \mid \text{time: } x, y\text{: } z] \Leftarrow [R_1 \mid \text{time: } x, \text{ car: } y, \text{ price: } z], y \in \{car_1, car_2, ..., car_n\}$,
$r_{DB_2\text{-}DB_1}$: $[R_1 \mid \text{time: } x, \text{ car: } y, \text{ price: } z] \Leftarrow [R_2 \mid \text{time: } x, y\text{: } z], y \in \{car_1'', car_2'', ..., car_l''\}$.

Finally, in a similar way, the semantic correspondence between DB_2 and DB_3 can be constructed as follows:

$r_{DB_3\text{-}DB_2}$: $[R_2 \mid \text{time: } x, y\text{: } z] \Leftarrow [y \mid \text{time: } x, \text{ price: } z], y \in \{car_1, car_2, ..., car_n\}$,
$r_{DB_2\text{-}DB_3}$: $[y \mid \text{time: } x, \text{ price: } z] \Leftarrow [R_2 \mid \text{time: } x, y\text{: } z], y \in \{car_1', car_2', ..., car_n'\}$.

In the remainder of this paper, a conjunction consisting of RSTs and normal first-order predicates is called a c-expression (standing for "complex expression"). For a derivation rule of the form $B \Leftarrow A$, A and B are called the antecedent part and the consequent part of the rule, respectively.

4.3 Extended Substitutions

A third important concept is that of so-called *extended substitution*.

Note that an attribute involved in such an algebra expression may either appear in sc_i ($i = 1, ..., m$), or/and in $\{A_1, ..., A_l\}$, or may not be involved in any operation at all. To characterize this feature, we associate each attribute with a label which consists of a subset $ap \subseteq \{p, s, ni, V\}$, where p, s, ni and V stand for 'project,' 'select,' 'not-involved' and 'the current values of the attribute,' respectively.

In terms of an algebra expression q, we can instantiate the variables appearing in the consequent part of a rule which matches q. Then, by means of constant propagation, the antecedent part of the rule can also be instantiated; and what we want now is to derive a set of new algebra expressions in terms of it. Unfortunately, such a derivation can not be done only by means of constant propagation since both the higher order information (e.g., information about iterations over relation/attribute names) and the necessary control mechanism are absent. To this end, we introduce the concept of extended substitutions.

- $aV(q(x_1, ..., x_k), x_{ij})$ returns an assumed value (for x_{ij}), where q is a first order predicate, x_i ($i = 1, ..., k$) may be a variable, a constant or a set of constants but $x_{i_j} \in \{x_1, ..., x_k\}$ must be a variable. For example, $aV(x \in \{c_1, c_2, ..., c_n\}, x) = \in \{c_1, c_2, ..., c_n\}$. (Note that here the predicate $q(x)$ is of the form $x \in \{c_1, c_2, ..., c_n\}$.)

Algorithm: *substi-production*(P, e)

(*P is a c-expression and e is an algebra expression.*)
input: P: a c-expression; e: an algebra expression;
output: ES: an extended substitution;
begin
1 ES := ϕ;
2 **if** e is of the form $\pi_{A_1}...A_l(\sigma\{\sigma_{sc_1}...{}_{sc_m}(R))$ **then** {
3 construct two sets for e:
4 PA := $\{A_1, ..., A_l\}$;
(*PA contains the attributes involved in project operations.*)
5 SC := $\{sc_1, ..., sc_i, ..., sc_m\}$;
(*SC contains all select conditions.*)
6 **for** each $a \in$ PA **do**
7 {let A: x be an attribute descriptor of some RST in P;
8 if $a = A$ **then** ES := ES $\cup$ $\{x/(p, -)\}$}
9 **for** each $a \in$ SC **do**
10 {let A: x be an attribute descriptor of some RST in P;
11 assume that a is of the form: $B\ \beta\ C$;
12 **if** $B = A$ **then**
{v := *assumedValue*($B\ \beta\ C$, P); ES := ES $\cup$ $\{x/(s, v)\}$;}}
13 **for** each predicate of the form: $q(x_1, ..., x_k)$ in P **do**
14 **for** each variable x_{ij} in q **do**
15 {$v := aV\ (q(x_1, ..., x_k), x_{ij})$; ES := ES $\cup$ $\{x_{i_j} / (-, v)\}$
end

Example 2 Consider the algebra expression $e = \pi_{price}(\sigma_{car = Mercedes \wedge time = 'July\ 1994'}(R_1))$. If we want to translate it into an algebra expression which can be evaluated against DB_2 as shown in Example 1, then the rules for specifying the semantic discrepancies between DB_1 and DB_2 will be considered, and the matching rule will be $r_{DB_1\text{-}DB_2}$. Its antecedent part P is of the form $[R_1 \mid \text{time: } x, \text{ car: } y, \text{ price: } z]$, $y \in \{car_1, car_2, ..., car_n\}$.

First, executing lines 2-5, we have

PA = {price},
SC = {car = Mercedes, time = 'July 1994'}.

Then, executing lines 6-8, we obtain

ES = {z/(p, -)}.

Next, after lines 9-12 are performed, ES is of the following form:

ES = {z/(p, -), x/(s, = 'July 1994'), y/(s, = Mercedes)}.

Finally, by executing lines 13-15, a new item, y/{(–, Î {car_1, car_2, ..., car_n} (constructed in terms of the predicate y Î {car_1, car_2, ..., car_n}), is inserted into ES. Therefore, the final ES is of the form:

{z/(p, -), x/(s, = 'July 1994'), y/{(s, = Mercedes)},
(-, ∈ {car_1, car_2, ..., car_n})}}.

Note that in the final ES, the pair (-, Î {car_1, car_2, ..., car_n} should be eliminated if "Mercedes" Î {car_1, car_2, ..., car_n} holds since "y = Mercedes" subsumes "y Î {car_1, car_2, ..., car_n}". In addition, if "Mercedes" does not belong to {car_1, car_2, ..., car_n}, then *substi-production* should report a "nil" to indicate that matching did not succeed, and that translation can not be done based on the rule. In fact, if {car_1, car_2, ..., car_n} does not contain "Mercedes," then any query concerning "Mercedes" submitted to DB_2 will evaluate to "nil." In the algorithm, however, such checks are not described for simplicity. It is easy to extend this algorithm to a complete version.

After the ES is evaluated, we can derive a set of new algebra expressions in terms of it, the RSTs and the first-order predicates appearing in the consequent part of the rule. This can be done by executing the following algorithm, which generates not only two sets, PA and SC, (from which an algebra expression can be constructed), but also a set of iteration control statements of the form for ... do, a set of checking statements with the form: if ... then, and a set of print statements. Together with PA and SC produced by the algorithm, such statements enable us to generate a complete query.

The main idea is as follows.

Consider a variable appearing in an RST. It may be a variable ranging over the relation names, a variable ranging over the attribute names or a variable ranging over some attribute values. Then, in terms of its bindings recorded in the corresponding ES, we can immediately fix its assumed value. On the other hand, which statements are associated with it can also be determined through synthetic analysis of its assumed value and its properties.

Algorithm: *expression-production*(P, δ)

(*P is a c-expression and δ is an ES.*)}
input: P: a c-expression; δ: an ES;
output: PA: project attributes; SC: select conditions;
FS: iteration control statements; CS: checking statements;
begin
1 SC := ϕ; PA := ϕ; FS := ϕ; CS := ϕ;
(*SC, PA, FS and CS are global set variables.*)
2 construct V_1, a set of variables (in P) ranging over attribute values;
3 construct V_2, a set of variables (in P) ranging over attribute names;
4 construct V_3, a set of variables (in P) ranging over relation names;
5 **for** each $x \in V_1$ **do**

```
   6  call attr-value-handling(x, P, δ);
7  for each x ∈ V2 do
   8  call attr-or-rel-name-handling(x, P, δ, 0);
9  for each x ∈ V3 do
  10  call attr-or-rel-name-handling(x, P, δ, 1);
end
```

From the above algorithm, we can see that two subprocedures are called to deal with different cases. That is, *attr-value-handling* is used to tackle variables ranging over attribute values and *attr-or-rel-name-handling* is employed to deal with variables ranging over attribute names or variables ranging over relation names. Below, we give a formal description for each. First, we define the following operation:

- *conditionProduction*($x\alpha y$) returns a select condition of the form $E\alpha F$ if E:x and F:y are two attribute descriptors in the corresponding c-expression.

```
Algorithm: attr-value-handling(x, P, δ)
begin
1  let A:x be an attribute descriptor of some RST in P;
2  (*Here A is an attribute name or a variable.*)
3     if x/(p, -) is a binding in δ then PA := PA ∪ {A};
4     if there exist bindings: x/(s, v1), ..., x/(s, vk) in δ then
5        {for i = 1, ..., k do
6           {sci := conditionProduction(xvi); SC := SC ∪ {sci};}
end
```

```
Algorithm: attr-or-rel-name-handling(x, P, δ, Int)
begin
1  if Int = 0 then find x:z,
     which is an attribute descriptor of some RST in P
2  else find [x |...], which is an RST in P;
   (*If Int = 0, x is a variable ranging over attribute names.*)
3  if there exist x/(s, v1), ..., x/(s, vk) in δ then
4     {for i = 1, ..., k do
5        {if vi is of the form =c, then replace x with c in all the newly produced PAs and
         SCs as well as iteration control statements, checking statements and printing
         statements
         (*see below*)
6         else
7           {let vi be of the form: αX;
8           generate a statement of the form: if xαX then;}}}
                (*produce a checking statement*)
9  if there exists a binding of the form x/(-, ∈ {c1, c2, ..., cm}),
10 then generate a statement of the form:
              for each x ∈ {c1, c2, ..., cm} do;
     (*produce an iteration statement*)
```

11 if there exist bindings $x/(-, v_1')$, ..., $x/(-, v_l')$ in δ with each $v_i' \neq$ " $\in \{c_1, c_2, ..., c_m\}$",
12 then {**for** $i = 1, ..., l$ **do** {generate a statement of the form if xv_i' then;}}
13 if there exist a binding of the form: $x/(p, -)$ in δ, **then**
14 generate a output statement of the form print(x);
end

The result of these algorithms can be thought of as being composed of four parts: a set of iteration control statements, a set of checking statements, a set of printing statements and an algebra expression derivable from PA and SC produced by the algorithm. If for each variable x (in the algebra expression) ranging over the relation names or ranging over the attribute names, there is a statement of the form: for each $x \in \{c_1, c_2, ..., c_m\}$ do, where $c_1, c_2, ..., c_m$ are constants, then this result corresponds to a program which can be correctly executed. We do this as follows.

First, we suffix each iteration statement and each checking statement with an open bracket "{" and suffix each printing statement with a semi-comma. Then, we change the newly generated algebra expression e' with "if e' then" and suffix it with "{". Next, we put them together in the order: iteration statements – checking statements – algebra expression – printing statements. Finally, we put the same number of close brackets "}" at the end of the sequence of the elements. For example, for the algebra expression $e = \pi_{time, car}(\sigma_{price>10000}(R_1))$, the following elements will be generated based on rule $r_{DB_1\text{-}DB_3}$:

"for each $y \in \{car_1', car_2', ..., car_m'\}$ do",
"print(y)",
"$\pi_{time}(\sigma_{price>10000}(y))$".

Then, the corresponding code will be of the form:

for each $y \in \{car_1', car_2', ..., car_m'\}$ do
{if $\pi_{time}(\sigma_{price>10000}(y))$ **then**
{print(y);}}.

According to the above discussion, the entire process for translating a simple algebra expression of the form: $\pi(\sigma(R))$ can be outlined as follows.

Algorithm: *simple-query-translation*(r, e)

input: r: a derivation rule; e: an algebra expression;
output: a program corresponding to the translated query;
begin
δ := *substi-production*(antecedent-part of r, e);
S := *expression-production*(consequent-part of r, δ);
generate a program in terms of S;
end

4.5 About Other Types of Semantic Conflicts

As mentioned earlier, semantic conflicts of types 4 and 5 can be handled in the same way as we handle types 1, 2 and 3 conflicts. This can be illustrated using a simple example.

Consider two databases containing information about *parent*, *brother* and *uncle*. The local schemas are as follows:

DB: *parent*(name, child), *brother*(name, brother_name).
DB′: *uncle*(name, nephew_niece).

In DB, *parent* is a relation, with one tuple per person and one of his/her children; *brother* is another relation, storing the pairs of the form (<person>, <one of his/her brothers>). In DB′, there is only one single relation, with one tuple per man and one of his nephews or nieces.

Then, their semantic correspondence can be established as follows:

$r_{DB\text{-}DB'}$:
[*uncle* | name: *x*, nephew_niece: *y*] ⇐ [*parent* | name: *z*, child: *y*],
brother | name: *z*, brother_name: *x*]

$r_{DB'\text{-}DB}$:
[*parent* | name: *x*, child: *y*], [*brother* | name: *x*, brother_name: *z*]
⇐ [*uncle* | name: *z*, nephew_niece: *y*].

Accordingly, all the techniques discussed above (with a bit of modification) can be used to translate a query involving both *parent* and *brother*, or *uncle*. In this way, the semantic conflicts of type 4 can be resolved.

The same analysis applies to the semantic conflicts of type 5.

5. BALANCING A JOIN TREE

After local query translation, the size of each "relation" can be determined if the corresponding estimation information is available. Now, we can consider the transformed global query as a normal one with the difference being that the "relations" may be distributed to different sites, and we can generate a left deep join tree for it using the traditional methods [9, 10, 20], without taking the relation distribution into account for the time being.

Then, based on such a left deep join tree, we can further generate a balanced join by performing a tree transformation. In the following, we first present the concept of basic transformation and its extension in 5.1. Then, in 5.2, a recursive algorithm for the tree transformation is described.

5.1 Transformation of Left Deep Join Trees

The algorithm proposed by Du *et al*. can be used to transform a left deep join tree into a balanced bushy join tree. The main idea behind that method is repeated application of basic transformations to the join sequence (called the left deep join tree in [14]). According to [14], a basic transformation is a transformation step which takes a segment from a left deep join tree as the input and then translates 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); here n_a and n_b are the UAN and the LAN of the selected segment, respectively)
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 the original UAN (n_a) is removed from the left deep join tree.
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)).

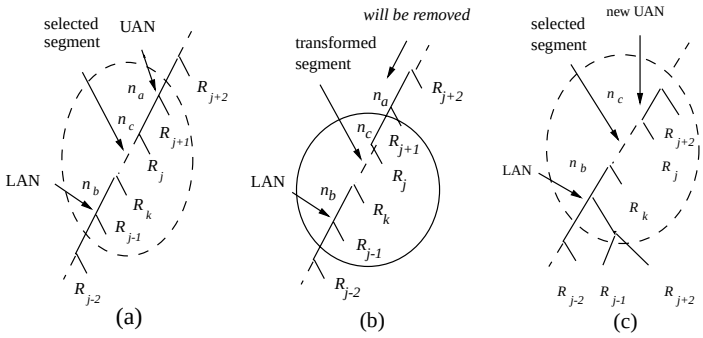


Fig. 5. Illustration of basic transformation.

Example 3 Consider the bushy join tree shown in Fig. 6(a). The cost of each base relation and the response time of each join node are shown in the figure. For ease of exposition, we assume that the cost of each local join is 5 units of time, and that between the members of each pair of relations there exists a join predicate. If we take the UAN and LAN as shown in Fig. 6(a), a basic transformation will translate the tree into the form shown in Fig. 6(b).

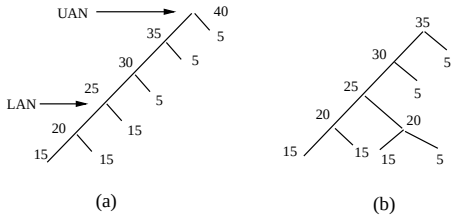


Fig. 6. Basic transformation.

The primary use of 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 the original number, the query response time can be improved if the new join can be executed in parallel without resulting in any 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 this is possible. To this end, we introduce the concept of extended basic transformation.

Definition 3 *Extended basic transformation* (*EB*-transformation) is a transformation step which takes a subtree T as the input and translates 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; and the original UAN (n_a) is removed from the left deep join tree.
3. Let n_b be the chosen LAN. Then, n_b remains unchanged but its right (or left) child node is replaced with a new join node between n_b's right (or left) node and the right node of the original UAN. Such a transformation is denoted *EB*-transformation(T, n_b).

The advantage of *EB*-transformation over basic transformation [14] can be seen in the following example.

Example 4 Consider the bushy join tree shown in Fig. 7(a).

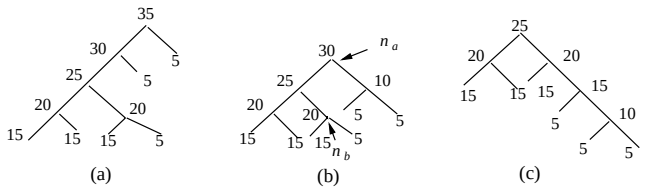


Fig. 7. Comparison of basic transformation and *EB*-transformation.

The cost of each base relation and the response time of each join node are shown in the figure. As in Example 3, we assume that each local join operation has a cost of 5 units of time, and that between the members of each pair of relations there exists a join predicate. Using the hybrid algorithm in [14], this bushy join tree can be translated into the tree shown in Fig. 7(b). But further balancing 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 4.3 of [14]). However, if we use *EB*-transformation as the basic transformation step, we can translate the tree shown in Fig. 7(b) into the tree shown in Fig. 7(c) by taking n_a (the root of the tree) as the UAN and n_b (an interior node) as the LAN. From Fig. 7(b), we see that n_a and n_b are not on the same segment.

In general, we have the following proposition.

Proposition 1 For any binary join tree T, if we take T's root as the UAN, then applying *EB*-transformation to T is no worse than applying basic transformation (as proposed by Du *et al.*) to T.

Proof: The proof is straightforward. See Fig. 8.

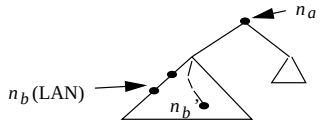


Fig. 8. Illustration of basic transformation and *EB*-transformation.

Since when we used the basic transformation of Du *et al*., UANs and LANs must be on the same segment, the possibility that an appropriate LAN for the root n_a will be selected by it is smaller than that if *EB*-transformation is applied, if the depth-first search method is employed in the implementation of *EB*-transformation to traverse the corresponding subtree. For example, if an appropriate LAN on the segment exists, say n_b (see Fig. 7), then both basic transformation and *EB*-transformation will find it. However, if such a LAN does not exist, basic transformation will do nothing for the current UAN. In contrast, *EB*-transformation will try another LAN, say n_b', if it exists. Therefore, a more balanced join tree can be obtained by applying *EB*-transformation than by applying the basic transformation of Du *et al*. □

Obviously, this improvement is at the cost of more searches of a subtree. Therefore, more time will be required by an *EB*-transformation step than by a basic transformation step. However, by developing a recursive algorithm for generating balanced join trees (with the *EB*-transformation being used), this drawback can be overcome. As we will see later, the entire time complexity of the recursive algorithm is not worse than that of Du's hybrid algorithm. But more balanced binary trees can be generated (see subsection 5.2).

Intuitively, for any join subtree, what we want is those *EB*-transformations by means of which the response time can be improved. This leads to another important concept.

Definition 4 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) (where 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. A time improving *EB*-transformation is denoted *TIEB*-transformation(T, n_b).

In order to facilitate time improvement checking for a LAN, we associate each node in a join tree with a pair of the form (*rt*, *Card*), where *rt* represents the response time of the subtree rooted at this node and *Card* stands for the cardinality of the intermediate result of this subtree. Given the response time and the cardinality for each base relation, the pairs of interior nodes can be computed as follows. Let v be an interior node, and let a and b be two child nodes of it. Then:

$$rt(v) = max\ \{rt(a), rt(b)\} + rt(a \bowtie b), \quad (1)$$

$$Card(v) = size\text{-}estimation\ (Card(a), Card(b)), \quad (2)$$

if pipelining is not considered. (The effect of pipelining was examined in [7, 30].) In fact, in our implementation only sort-merge join and nested join are taken into account. It is worth mentioning that due to its stable performance, sort-merge join is the most prevalent join method used in some database products to handle both equal and nonequal join queries [1]. Hash-based join, though having good average performance, suffers from the problem of hash bucket overflow and is, thus, avoided in many commercial database products.

Note that both *rt* and *size-estimation* can be calculated using the formulas provided in [36], in [41] or in [17]. For each selected LAN in a join tree, the pairs associated with the nodes on the path connecting the LAN and the left child of UAN will be computed anew in a bottom-up manner based on the newly generated join between the right child node of the UAN and the right (or left) child node of the LAN. See Fig. 9 for an illustration.

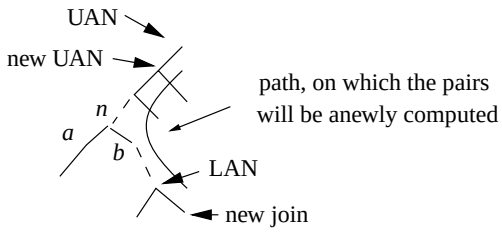


Fig. 9. Illustration of recomputation of response times and estimated sizes.

In addition, three heuristics are utilized to find a time improving LAN as quickly as possible:

1. A node will be considered as a candidate for a LAN only if its right (or left) child node has join predicates with the right child node of the original UAN.
2. If the right child node of a LAN has join predicates only with the right child node of the original UAN, this LAN will be selected.
3. If (2) is not the case, i.e., the right (or left) child node of each such LAN has join predicates with other base relations, then we select the LAN with the best time improvement.

As with the basic transformation step in [14], a time improving *EB*-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 *TIEB*-transformations through cost computation. In this process, the cost formulas provided in [36] or in [41] can be used. In the following, we outline a top-down process.

We traverse 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 algorithm differentiates among three cases:

1. If its left child node is a leaf node, then 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, then no transformation will be applied to the UAN. But recursive invocations of the algorithm on both the right and left subtrees will be performed.

3. If the response time of the left subtree is greater than that of the right subtree, then the subtree rooted at the left child node of the UAN is traversed to find a LAN, by means of which a *TIEB*-transformation can be performed. If no such 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. That is, the right (or left) child node of the LAN will be replaced with the join between the right (or left) child node of the LAN and the right child node of the original UAN, which will subsequently be removed.

Example 5 We will trace a sample transformation to illustrate the above top-down process. See the left deep join tree shown in Fig. 10(a), in which there exist join predicates between relations A and B, B and C, C and D, D and E, and C and F. As in Example 3, we assume for clarity of explanation that each local join operation has a cost of 5 units of time. Furthermore, to make the trace non-trivial, we assume that each local relation has an initial cost (shown in the figure).

First, by taking the root as the UAN and the node marked with 25 as the LAN, it is translated into the form shown in Fig. 10(b). Then, taking the node marked with 30 as the LAN (note that the node marked with 35 becomes the new UAN), this tree will be changed into the tree shown in Fig. 10(c). Finally, the balanced tree shown in Fig. 10(d) can be obtained.

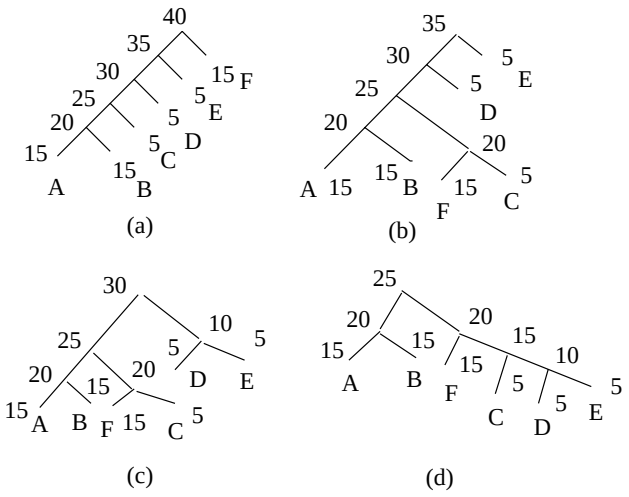


Fig. 10. Sample trace of top-down approach.

Although the top-down approach works in some cases using *TIEB*-transformation, it suffers from a critical problem; that is, the algorithm balances a node before its child nodes are balanced, which may keep us from finding a more balanced join tree. In the following subsection, we propose a recursive algorithm to overcome this deficiency.

5.2 Recursive Algorithm for Balancing Left Deep Join Trees

To demonstrate the problem with the top-down approach stated above, let us examine the left deep join tree shown in Fig. 11(a). For ease of exposition, we assume that between the members of each pair of relations there exists at least one join predicate.

After a *TIEB*-transformation step (by taking the nodes marked with 40 and 35 as the UAN and LAN, respectively), it will be translated into the form as shown in Fig. 11(b). Since the left child subtree of the root is not balanced, this transformation makes a global balance impossible. Fig. 11(c) shows the final bushy join tree which is not well balanced. (See Fig. 12(d) for a comparison.)

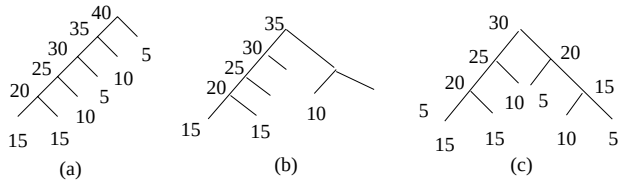


Fig. 11. Example of top-down transformation.

To improve this behavior, we propose the following recursive algorithm, which works better than both the above top-down strategy and the hybrid algorithm proposed in [14]. The key idea behind the algorithm is that before a tree is balanced, the left child subtree of its root is 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 *TIEB*-transformation can be performed. With the *TIEB*-transformation, this strategy leads to a more balanced tree.

Procedure *tree-transformation* (*LDT, root*)
(**LDT* stands for a left deep (or bushy) join tree; *root* represents its root.*)
begin
  if the height of *LDT* $h \leq 2$ **then** return;
  let *LDT*' and *r-node* be the left child subtree and the right child node of the root, respectively;
  let *a* be the root of *LDT*';
  call *tree-transformaiton* (*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
  EB-transformation can be applied and if at the same time, it is the best one compared to the other transformable nodes;
  then {make the *EB*-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

The above algorithm works by means of recursive procedure calls. Thus, each recursive call returns nothing. But some subtree will be changed by each procedure call. Therefore, whenever the control is switched over to a calling procedure, it will handle a new subtree whose root's left subtree has been balanced. The entire tree will be balanced after the highest recursive call is executed.

In the following, we present another trace of tree transformation to illustrate the major idea behind our recursive algorithm.

Example 6 Fig. 12(a) shows a left deep join tree, 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. As before, each join operation is assumed to take 5 units of time.

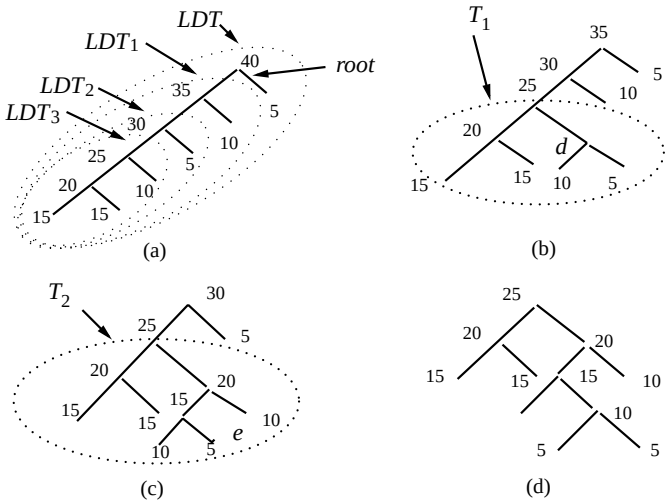


Fig. 12. Tracing a sample example.

At the beginning of the procedure, three recursive calls will be executed successively:

tree-transformation (LDT, *root*) $\rightarrow$ *recursive-call*
tree-transformation (LDT_1, *a*) $\rightarrow$ *recursive-call*
tree-transformation (LDT_2, *b*) $\rightarrow$ *recursive-call*
tree-transformation (LDT_3, *c*).

After the return from *tree-transformation*(LDT_3, *c*) to *tree-transformation*(LDT_2, *b*), the first transformation (as shown in Fig. 12(b)) occurs, and tree T_2 is the corresponding local balanced tree. Then, the control is returned to *tree-transformation*(LDT_1, *a*). During the execution of it, tree T_2 is searched, and node *d* is selected as the LAN (see Fig. 12(b) again). Consequently, tree T_1 is constructed as shown in Fig. 12(c). Afterwards, the control is returned to *tree-transformation*(LDT, *root*). During the execution of it, T_1 is traversed, and node *e* is selected as the LAN. (See Fig. 12(c) again.) Lastly, the tree shown in Fig. 12 (d) is generated.

From this example, we see that the result obtained using *tree-transformation*( ) is better than that obtained using the top-down strategy. In fact, it is also better than the result obtained using the hybrid method in [14]. In this example, the hybrid algorithm behaves accidently like a top-down strategy. (See [14] for a detailed description.) However, from the description of the hybrid algorithm, we know that it is a combination of a top-down and a bottom-up process. During execution, a top-down process switches over to a bottom-up process when some condition is satisfied, and *vice versa*. Therefore, when a hybrid process accidently becomes top-down execution, our method works better than it does. But one may argue that in some cases, when a bottom-up process is involved, the hybrid algorithm may outperforms ours.

In the following, we give another example to demonstrate that this is not the case, either.

Example 7 Examine the left deep join tree shown in Fig. 13(a). In terms of the mechanism of the hybrid algorithm [14], a bottom-up process is invoked, which translating the tree shown in Fig. 13(a) into the one shown in Fig. 13(b) in one basic transformation step. Further, by means of a second basic transformation, the tree shown in Fig. 13(b) is translated into the tree shown in Fig. 13(c). Any further transformation is not possible, but using *tree-transformation*( ), the tree shown in Fig. 13(d) can be obtained.

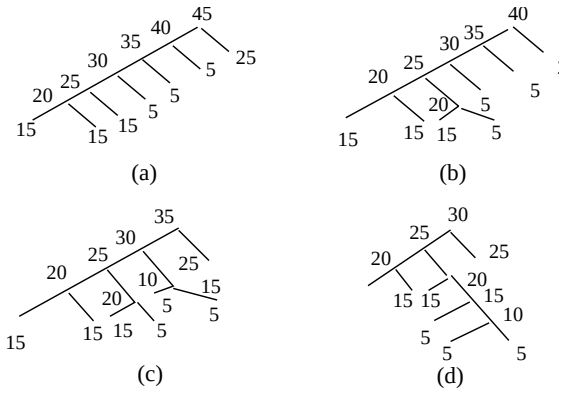


Fig. 13. Tree transformation.

In addition, the time complexity of *tree-transformation*( ) is the same as that of the hybrid algorithm, in which O(n·log$n$) basic transformations are required, where n is the number of base relations involved in the join tree. In contrast, *tree-transformation*( ) needs only n basic transformations since there are only n recursive calls altogether and by means of each recursive call, at most one *EB*-transformation can be performed. If we do not take node allocation into account, the transformation used in our method takes the same amount of time as does that used in [14]. However, the complexity of tree traversal in our method is O(n^2) time. This is the case because O(n) nodes are accessed by each depth-first search of some subtree and there are n subtree traversals in the worst case. But the complexity of tree traversal in the method proposed by Du *et al*. is also on the order of O(n^2) (see [13]).

In the following, we prove a proposition to show that our recursive algorithm generally aids tree balance.

Proposition 2 Let T be a bushy binary tree, and let T' and T" be two balanced bushy join trees obtained by applying *tree-transformation*( ) and the hybrid algorithm proposed by Du *et al*. to T, respectively. Then, $rt(T') \leq rt(T")$, where $rt(T)$ stands for the response time of T.

Proof: We prove the proposition by induction on the height of T: h.

Basis: When $h \leq 3$, the proof is trivial (see Proposition 1).

Induction step: Suppose that for some k, for all T with $h \leq k$, $rt(T') \leq rt(T")$ holds. We consider a bushy join tree with height $k + 1$ (see Fig. 14(a)).

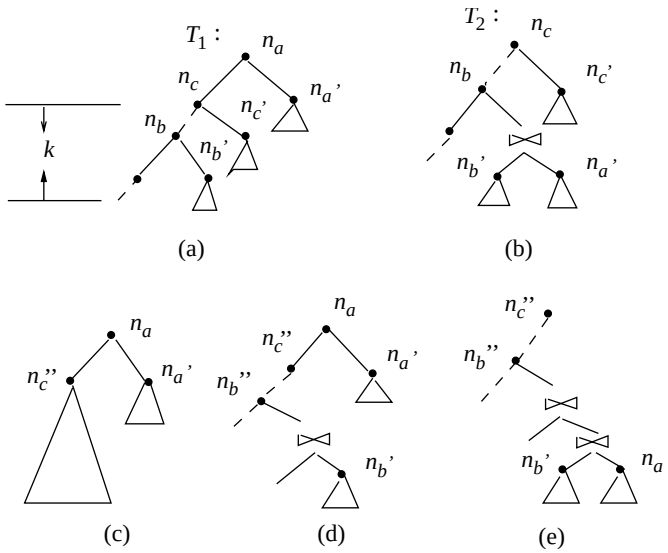


Fig. 14. Illustration of tree transformation.

If we apply Du's hybrid algorithm to T_1, then one of the following three cases may happen in its first main loop:

1. During top-down traversal, a UAN and a LAN are determined.
2. During bottom-up traversal, a UAN and a LAN are determined.
3. No transformation can be made.

We first consider case (1). Without loss of generality, assume that the root (n_a) is chosen as the UAN, and that the corresponding LAN is n_b (see T_1 shown Fig. 14(a)). Then, T_1 will be transformed into T_2 as shown in Fig. 14(b). On the other hand, if we use *tree-transformation*( ) for T_1, then at the moment when the subtree rooted at the left child of n_a is searched to find a LAN for n_a (see Fig. 14(c) for an illustration, where n_c" represents the root of the transformed left subtree of T_1), we find the following two cases:

(i) n_b has not been removed.
(ii) n_b has been removed.

If n_b has not been removed, then *tree-transformation*( ) will do the same transformation as shown in Fig. 14(b). Then, in effect, the tree obtained by applying *tree-transformation*( ) to T_1 is the same as that obtained by applying *tree-transformation*( ) to T_2. Thus, in terms of the induction suppose, we have:

$$rt(T_1') = rt(T_2') \leq rt(T_2'') \leq rt(T_1'').$$

If n_b has been removed, then without loss of generality, we assume that the corresponding tree is of the form shown in Fig. 14(d) (in which n_b'' stands for the root of the left subtree of n_b before it is removed). Then, *tree-transformation*( ) finds n_b' (or a node with better response time in terms of the heuristics discussed in subsection 5.1); and the resultant tree is of the form shown in Fig. 14(e). From this, we see that

$$rt(T_1') \leq rt(T_2').$$

This is because if we apply *transformation*( ) to T_2 directly, then nodes n_b' and n_a' can only be moved together (as the join node between them). But if we apply *transformation*( ) to T_1, then better transformation may be performed due to the lack of this restriction. Thus, the following inequalities hold:

$$rt(T_1') \leq rt(T_2') \leq rt(T_2'') \leq rt(T_1'').$$

Case (2) can be proved in a way similar to that in case (1).

For case (3), we denote T_3 as the left subtree of T_1. Then, in terms of the induction suppose, we have

$$rt(T_3') \leq rt(T_3'').$$

Note that $rt(T_1') = rt(T_3') + t'$ and $rt(T_1'') = rt(T_3'') + t''$, where t' and t'' represent the elapsed times for the join between T_3' and the right subtree of n_a and the join between T_3'' and the right subtree of n_a, respectively. Thus, $rt(T_1') \leq rt(T_1'')$. □

Together with Example 7, this proposition shows that in all the cases, our recursive algorithm outperforms the hybrid algorithm proposed in [14].

In order to demonstrate that the improvement in response time is not trivial, we examine an extreme situation, in which the reduction in response time is tremendous. To this end, we show a tree (called a *basic tree*, denoted T_1) in Fig. 15(a), whose response time is $5 \cdot a_1$ units of time, where we assume that the cost of each local join is a_1 units of time. Using our algorithm, this tree can be transformed into the form shown in Fig. 15(b). Its response time is $4 \cdot a_1$ units of time. But this transformation is not possible using the hybrid algorithm proposed in [14]. Similarly, we can construct a second basic tree T_2 as shown in Fig. 15(c), where $a_2 = 4 \cdot a_1$ and the cost of each local join is a_2 units of time. We denote the tree shown in Fig. 15(d) as $T_1 + T_2$. Then, our algorithm can improve the response time by $a_1 + a_2$ units

of time. In the same way, we can define the *n*th *basic tree* T_n and construct $T_1 + T_2 + ... + T_n$ as shown in Fig. 15(e), where $a_{i+1} = 4 \cdot a_i$ ($i = 1, ..., n - 1$). Using our algorithm, its response time can be reduced by $\sum_{i=1}^{n} a_i$ units of time. But no improvement can be made at all using Du's algorithm.

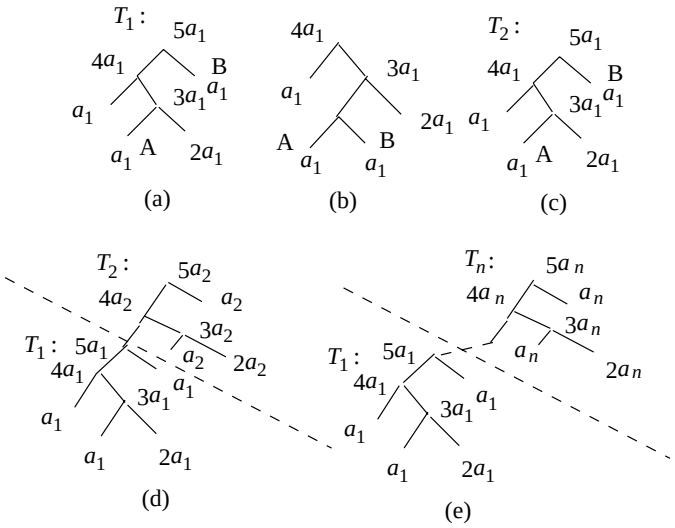


Fig. 15. Tree transformation.

6. NODE ALLOCATION

Once a balanced bushy join tree has been generated, we should apply a node allocation to it by taking relation distribution into account. This can be done by assigning a label to each node of the tree to indicate at which site the corresponding (join) operation will be performed.

6.1 Communication Costs and Load Measurements

In order to achieve reasonable node allocation, we have to know both the load states of the local databases and the communication costs of the network. To this end, we maintain a dynamic matrix for the transmission rate between each pair of local databases and a dynamic table for load measurements, which is changed periodically to indicate the actual load states of networks as well as of local databases (see Fig. 16(a) and (b)). Note that changing the dynamic table periodically is the only small workload imposed on the network by our global federated system manager.

Fig. 16(a) shows a sample transmission cost matrix, in which each c_{ij} corresponds to the rate of transmission from DB_i to DB_j. Such a matrix has to be changed periodically to record the actual transmission rates so that the actual load states of networks (accordingly, the data transmission delay) can be observed. In the table shown in Fig. 16(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 using the queuing analysis.

transmission cost c_{ij} = per bit x 10^{-4}

i \ j	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	2.0
DB_2	4.0	5.0
DB_3	7.0	3.0
DB_4	8.0	1.5

(b)

Fig. 16. Transmisson cost matrix and load measurement.

Clearly, we have the following formulas for computing response times, assuming that the query arrival rate obeys the Poisson distribution, and that the service time is exponentially distributed [33]:

$$\textit{system load}: \rho = \lambda \cdot s, \tag{3}$$

$$\textit{average response time} = \begin{cases} t_q = s/(1-\rho), & \rho < 1; \\ \infty & \rho \geq 1, \end{cases} \tag{4}$$

where λ represents the current query arrival rate and s represents the average service time. For example, from the first entry in the table shown in Fig. 16(b), we know that the response time of DB_1 is $t_q = s/(1 - \rho) = 6.0/(1 - 0.2) = 7.5$.

The response time can be more exactly estimated by taking the join costs into account. For this purpose, we divide the costs of joins into several classes and record the average service time for each. For example, we can classify the costs of joins into four categories, J1, J2, J3 and J4, in terms of the cost function $|R_i| + |R_j| + |R_i \bowtie R_j|$ [41], such that $0 \leq cost_{J1} < 50 \times 10^6$, $50 \times 10^6 \leq cost_{J2} < 100 \times 10^6$, $100 \times 10^6 \leq cost_{J3} < 150 \times 10^6$ and $150 \times 10^6 \leq cost_{J4} < \infty$. Then, according to [33], a sample load measurements can be of the form as shown in Fig. 17, in which each type has a different service time.

	average service time (sec.)				current query rate λ (per minute)
	sJ1	sJ2	sJ3	sJ4	
DB1	1.0	2.4	4.0	6.0	2.0
DB2	1.5	2.0	3.2	4.0	5.0
DB3	2.0	3.3	5.0	7.0	3.0
DB4	2.1	3.7	6.1	8.0	1.5

Fig. 17. Refined load measurement.

Accordingly, the average response time can be calculated as follows (see [33]):

$$\text{system load}: \rho = \lambda \cdot \frac{S_{J_1} + S_{J_2} + S_{J_3} + S_{J_4}}{4}, \tag{5}$$

$$\text{average response time}: t_{qi} = S_{J_i}/(1-\rho) \ (i = 1,\ 2,\ 3,\ 4). \tag{6}$$

Note that we can also compute each s_{Ji} above using the regression cost formulas for "join queries" proposed in [43] if the access methods used in the local databases and the information concerning the index structures (such as index-based join, clustered-index-based join, or sequential scan join) are known at the global schema level. Then, the index structures have to be taken into account for join classification. Conventional cost models [17, 36, 41] can estimate the size of intermediate results. Then, in terms of it, the transmission costs, the workload of the local database systems and the response time of each node can be estimated dynamically. Our approach is applicable regardless of the model used for estimation of intermediate result sizes or the probabilistic distribution used to derive the average response time of a local database system. We assume that some method of estimating the sizes of intermediate results is available.

In our multidatabase system, a relation may have several replicates distributed over local databases involved in the federation. Then, for each relation, the FSM (Federated System Manager) maintains a list of the names of its replicates, sorted according to the average response time of the local systems, in which they reside. A local system has the right to reject cooperation due to autonomy. If this happens, the next replicate (residing in another local system) is selected, and the planner is executed once again. By the execution, the rejecting local system is not considered any more. This process repeats until a plan is generated or aborted. If a local system becomes very busy during evaluation and its response time severely degenerates, we can treat it as a rejecting one. That is, we get it off, select another replicate and execute the planner again. (Due to space limitations, a detailed discussion of this issue will be given in another paper now in preparation.)

6.2 Node Allocation Strategy

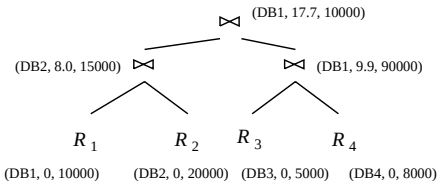
Given a balanced join tree, we label its leaf nodes (corresponding to the relations involved in the query) with the site numbers to indicate where a relation resides. Then, we try to label the remaining nodes to complete a node allocation so as to minimize the total response time. In fact, our method can be regarded as an extended version of the method proposed in [19], i.e., a dynamic programming strategy with the dynamic time tables being integrated (see below).

First, we define the following data structure, which is used in our model for node allocation.

Definition 4 A *dynamic time table* associated with a join tree T (denoted $dt\text{-}table_T$) is a set of pairs of the form (DB, v), where DB is a 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 *current allocation* of DB and is denoted $dt\text{-}table_T(DB)$.

For example, for the tree with node allocation, shown in Fig. 18(a), we have a dynamic time table as shown in Fig. 18(b), where each node in the tree is associated with a triple: (l, rt, S).

Here, l is a label which indicates 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.



DB_i	response time
DB_1	(9.9)
DB_2	(8.0, 17.7)
DB_3	( )
DB_4	( )

Fig. 18. Tree with node allocation and its dynamic time table.

Definition 5 *opt-rt*(*a*, *k*) is defined as the shortest response time of the subtree rooted at *a* such that *a* is labeled with *k*. If node *a* is pre-labeled with a label different from *k*, then *opt-rt*(*a*, *k*) = ∞ (which will prevent a pre-labled node *a* from being labled with any other lables.) If node *a* is pre-labeled with the same label as *k*, then *opt-rt*(*a*, *k*) = 0.

Consider a node *a* which has children $\alpha_1, \alpha_2, ..., \alpha_p$. If the labels for $\alpha_1, \alpha_2, ..., \alpha_p$ and the corresponding response times have been fixed, say $k_1, k_2, ..., k_p$ and $rt_1, rt_2, ..., rt_p$, respectively, then the response time of the subtree rooted at *a* is the response time of site *k* plus the time at which the last operand arrives from some α_j ($1 \le j \le p$) since an operation can not be performed at a site until all the operands have arrived at it and, at the same, it is idle. (Note that pipelining is not considered here.) Thus, the following equation holds:

$$opt\text{-}rt(a, k) = max\{max_{\alpha_j} \{c_{k_j k} \cdot |\alpha_j| + rt_j\}, dt\text{-}table_T(k)[\text{last}]\} + t_{ka}, \quad (7)$$

where α_j ($1 \le j \le p$) stands for the child nodes of *a*, $dt\text{-}table_T(k)[last]$ represents the last element in the list referenced by $dt\text{-}table_T(k)$, i.e., the response time of the last operation performed at site *k*, t_{ka} represents site *k*'s average response time calculated in terms of the average service time (w.r.t. the cost class of the join represented by *a*) and its load state, and $|\alpha_j|$ is the size of α. Note that if $dt\text{-}table_T(k)$ is an empty list, then $dt\text{-}table_T(k)$[last] returns 0.

More strictly, we have the following definition.

Definition 6 Let *L* be the set of all labels. Let *a* and *k* represent a node and some label in *L*, respectively. *opt-rt*(*a*, *k*) and another quantity *opt*(*a*) can be calculated recursively as follows:

(i) If *a* is a leaf node, unlabeled or pre-labeled with *k*, then *opt-rt*(*a*, *k*) = 0. If *a* is pre-labeled with a label other than *k*, then *opt-rt*(*a*, *k*) = ∞ (which will prevent a pre-labled node from being labled with any other lables).

(ii) *opt*(*a*) is defined as $min_{i \in L}\{opt\text{-}rt(a, i)\}$, i.e., the shortest response time of the subtree rooted at *a* irrespective of the label of *a*. In addition, a function *f*(*opt*(*a*)) is defined as a label *l* such that *opt*(*a*) = *opt-rt*(*a*, *l*).

(iii) If *a* is not a leaf node, then

$$opt-rt(a,k)=\begin{cases}\infty, & \text{if } a \text{ is pre-labeled with a label other than } k;\\ \max\{\max\limits_{\alpha_j}\{c_{f(opt(a_j),k}\cdot|\alpha_j|+opt(\alpha_j\},\\ dt-table_T(k)[\text{last}]\}+t_{ka}, & \text{otherwise,}\end{cases} \tag{8}$$

$$opt(a)=\min_{i\in L}\{opt-rt(a,i)\}, \tag{9}$$

$$dt\text{-}table_T(l):=dt\text{-}table_T(l)\cup\{opt\text{-}rt(a,l)\} \quad \text{if } opt\text{-}rt(a,l)=\min_{i\in L}\{opt\text{-}rt(a,i)\}. \tag{10}$$

The following example traces a sample labeling of a tree based on Definition 6.

Example 8 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 those shown in Figs. 16(a) and (b). (For the sake of simplicity, we will not consider the cost classification of joins here. Instead, we will assume that for a local database system, the response times of computing joins are the same, i.e., for any a $t_{ka} = t_k$, to clarify the main idea the strategy.) We will demonstrate how to compute *opt-rt* for the tree shown in Fig. 19, where each R_i (i = 1, 2, 3, 4) represents a base relation residing in DB_i. In this tree, the sizes of intermediate results are assumed. But they can be calculated using the formulas suggested in [17, 36, 41]. In addition, each leaf node is marked with a pair of the form (l, N), where l is a label used to indicate that the corresponding relation resides at site l and N represents its cardinality:

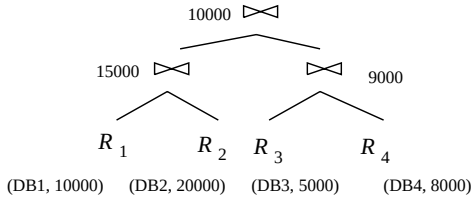


Fig. 19. A partly labeled tree.

$t_1 = s_1/(1 - \lambda_1 \cdot s_1) = 6.0/(1 - 0.2) = 7.5$ sec,
$t_2 = s_2/(1 - \lambda_2 \cdot s_2) = 4.0/(1 - 1/3) = 6.0$ sec,
$t_3 = s_3/(1 - \lambda_3 \cdot s_3) = 7.0/(1 - 0.35) = 9.33$ sec,
$t_4 = s_4/(1 - \lambda_4 \cdot s_4) = 8.0/(1 - 0.2) = 10.0$ sec,

First, we calculate the response time of each local database based on its actual load measurement.

Then, it is useful to think of *opt-rt*, *opt* and *f*(*opt*(...)) as three tables as shown in Fig. 20. Definition 4 can be applied to fill in the columns in the table in a left to right manner. In Fig. 21, the change of the dynamic time table is demonstrated.

opt-rt :	R_1	R_2	R_{12}	R_3	R_4	R_{34}	R
DB_1	0.0	∞	11.5	∞	∞	9.9	18.5
DB_2	∞	0.0	8.0	∞	∞	14.0	17.7
DB_3	∞	∞	12.33	0.0	∞	10.13	21.93
DB_4	∞	∞	14.0	∞	0.0	10.5	22.6

opt:	0.0	0.0	8.0	0.0	0.0	9.9	17.7

f(opt(...)):	DB_1	DB_2	DB_2	DB_3	DB_4	DB_1	DB_2

Fig. 20. Calculation of o*pt-rt*, *opt* and *f*(*opt*(...)) for tree of Fig. 19.

At the beginning, each *dt-table*$_T$(DB_i) is empty (see Fig. 21(a)). After the column for R_{12} is filled, the dynamic time table takes to the form shown in Fig. 21(b). Note that this column is computed using the values in the columns for R_1 and R_2. For example, using the formula given in Definition 5, *opt-rt*(R_{12}, DB_1) = *max*{*max* $\{c_{f(opt(R_1)),1} \cdot |R_1| + opt(R_1)$, *max* $\{c_{f(opt(R_2)),1} \cdot |R_2| + opt(R_2)\}$, *dt-table*$_T$($DB_1$)[last]} + t_1 = *max*{*max* 0, $2 \times 10^{-4} \times 20000 + 0\}$, 0} + 7.5 = 11.5. (In Appendix A, the detailed computations for this example are given; also, notice that the classifications of join costs are not considered in this example.)

Further, after the column for R_{34} and then the column for R are filled, the dynamic time table is changed to the forms shown in Figs. 21(c) and 21(d), respectively. In addition, the table for *f*(*opt*(...)) gives the last node allocation.

DB_i	response time
DB_1	( )
DB_2	( )
DB_3	( )
DB_4	( )

DB_i	response time
DB_1	( )
DB_2	(8.0)
DB_3	( )
DB_4	( )

DB_i	response time
DB_1	(9.9)
DB_2	(8.0)
DB_3	( )
DB_4	( )

DB_i	response time
DB_1	(9.9)
DB_2	(8.0, 17.7)
DB_3	( )
DB_4	( )

Fig. 21. Change of dynamic time table.

7. CONCLUSIONS

In this article, a new method for evaluating queries submitted to a relational multidatabase has proposed. This method mainly consists of four processes: the initial local process, the query translation, the join sequence production and bushy join tree production.

In order to implement an automatic query translation strategy, two important concepts, the relation structure term and extended substitution, have been introduced. Based on these concepts, many algorithms can be developed to perform query translation automatically even if schema conflicts exist. A second difficult problem is the generation of balanced join trees from a join sequence since, in essence, this problem is *NP*-hard [25]. To get a better approximately optimal solution to this problem, we have extended the basic

transformation step used in [14] and developed a recursive algorithm to obtain more balanced join trees. This algorithm has been proved to be significantly better than the one found by [14]. The third problem is that join operations have to be distributed reasonably. For this purpose, a new concept of the dynamic time table has been introduced and integrated into the dynamic programming strategy proposed in [19] to implement a node allocation algorithm. Lastly, the queuing theory has been utilized to calculate the response times of local database systems in terms of their actual load states.

APPENDIX A. COMPUTATION FOR EXAMPLE 7

In this Appendix, node allocation for a balanced join tree is illustrated using the tree shown in Fig. 19 and the assumed system state shown in Fig. 16.

First, for the base relations R_1, R_2, R_3 and R_4, we have:

$opt\text{-}rt(R_1, DB_1) = 0$; $opt\text{-}rt(R_1, DB_2) = \infty$; $opt\text{-}rt(R_1, DB_3) = \infty$;
$opt\text{-}rt(R_1, DB_4) = \infty$; $opt(R_1) = 0$; $f(opt(R_1)) = DB_1$.
$opt\text{-}rt((R_2, DB_1) = \infty$; $opt\text{-}rt(R_2, DB_2) = 0$; $opt\text{-}rt((R_2, DB_3) = \infty$;
$opt\text{-}rt((R_2, DB_4) = \infty$; $opt(R_2) = 0$; $f(opt(R_2)) = DB_2$.
$opt\text{-}rt(R_3, DB_1) = \infty$; $opt\text{-}rt(R_3, DB_2) = \infty$; $opt\text{-}rt(R_3, DB_3) = 0$;
$opt\text{-}rt(R_3, DB_4) = \infty$; $opt(R_3) = 0$; $f(opt(R_3)) = DB_3$.
$opt\text{-}rt(R_4, DB_1) = \infty$; $opt\text{-}rt(R_4, DB_2) = \infty$; $opt\text{-}rt(R_4, DB_3) = \infty$;
$opt\text{-}rt(R_3, DB_4) = 0$; $opt(R_4) = 0$; $f(opt(R_4)) = DB_4$.

The momentary contents of $dt\text{-}table_T$ are:

$dt\text{-}table_T(DB_1) = ( )$; $dt\text{-}table_T(DB_2) = ( )$; $dt\text{-}table_T(DB_3) = ( )$; $dt\text{-}table_T(DB_4) = ( )$.

Then, $opt\text{-}rt(R_{12}, DB_i)$ (i = 1, 2, 3, 4) can be computed as follows:

$$opt\text{-}rt(R_{12}, DB_1) = max\{max\{c_{f(opt(R_1)),1}\,|\,R_1\,| + opt(R_1)\},\ max\{c_{f(opt(R_2)),1}\,|\,R_2\,| + opt(R_2)\},\ dt\text{-}table_T(DB_1)[\text{last}]\} + t_1$$
$$= max\{max\{0,\ 2 \times 10^{-4} \times 20000 + 0\},\ 0\} + 7.5 = 11.5;$$
$$opt\text{-}rt(R_{12}, DB_2) = max\{max\{c_{f(opt(R_1)),2}\,|\,R_1\,| + opt(R_1)\},\ max\{c_{f(opt(R_2)),2}\,|\,R_2\,| + opt(R_2)\},\ dt\text{-}table_T(DB_2)[\text{last}]\} + t_2$$
$$= max\{max\{2 \times 10^{-4} \times 10000 + 0,\ 0\},\ 0\} + 6.0 = 8.0;$$
$$opt\text{-}rt(R_{12}, DB_3) = max\{max\{c_{f(opt(R_1)),3}\,|\,R_1\,| + opt(R_1)\},\ max\{c_{f(opt(R_2)),3}\,|\,R_2\,| + opt(R_2)\},\ dt\text{-}table_T(DB_3)[\text{last}]\} + t_3$$
$$= max\{max\{3 \times 10^{-4} \times 10000 + 0,\ 1 \times 10^{-4} \times 20000 + 0\},\ 0\} + 9.33 = 12.33;$$
$$opt\text{-}rt(R_{12}, DB_4) = max\{max\{c_{f(opt(R_1)),4}\,|\,R_1\,| + opt(R_1)\},\ max\{c_{f(opt(R_2)),4}\,|\,R_2\,| + opt(R_2)\},\ dt\text{-}table_T(DB_4)[\text{last}]\} + t_4$$
$$= max\{max\{3 \times 10^{-4} \times 10000 + 0,\ 2 \times 10^{-4} \times 20000 + 0\},\ 0\} + 1.0 = 14.0;$$

Now the contents of $dt\text{-}table_T$ become:

$dt\text{-}table_T(DB_1) = ( )$; $dt\text{-}table_T(DB_2) = (8.0)$; $dt\text{-}table_T(DB_3) = ( )$;
$dt\text{-}table_T(DB_4) = ( )$.

Subsequently, $opt\text{-}rt(R_{34}, DB_i)$ (i = 1, 2, 3, 4) can be evaluated:

$$
\begin{aligned}
&opt\text{-}rt(R_{34}, DB_1) = max\{max\{c_{f(opt(R_3)),1} \mid R_3 \mid + opt(R_3)\},\\
&\quad max\{c_{f(opt(R_4)),1} \mid R_4 \mid + opt(R_4)\}, dt\text{-}table_T(DB_1)[\text{last}]\} + t_1\\
&= max\{max\{3 \times 10^{-4} \times 5000 + 0,\ 3 \times 10^{-4} \times 8000 + 0\}, 0\} + 7.5 = 9.9;\\
&opt\text{-}rt(R_{34}, DB_2) = max\{max\{c_{f(opt(R_3)),2} \mid R_3 \mid + opt(R_3)\},\\
&\quad max\{c_{f(opt(R_4)),2} \mid R_4 \mid + opt(R_4)\}, dt\text{-}table_T(DB_2)[\text{last}]\} + t_2\\
&= max\{max\{1 \times 10^{-4} \times 5000 + 0,\ 2 \times 10^{-4} \times 8000 + 0\}, 8.0\} + 6.0 = 14.0;\\
&opt\text{-}rt(R_{34}, DB_3) = max\{max\{c_{f(opt(R_3)),3} \mid R_3 \mid + opt(R_3)\},\\
&\quad max\{c_{f(opt(R_4)),3} \mid R_4 \mid + opt(R_4)\}, dt\text{-}table_T(DB_3)[\text{last}]\} + t_3\\
&= max\{max\{0,\ 1 \times 10^{-4} \times 8000 + 0\}, 0\} + 9.33 = 10.13;\\
&opt\text{-}rt(R_{34}, DB_4) = max\{max\{c_{f(opt(R_3)),4} \mid R_3 \mid + opt(R_3)\},\\
&\quad max\{c_{f(opt(R_4)),4} \mid R_4 \mid + opt(R_4)\}, dt\text{-}table_T(DB_4)[\text{last}]\} + t_4\\
&= max\{max\{1 \times 10^{-4} \times 5000 + 0, 0\}, 0\} + 1.0 = 10.5;
\end{aligned}
$$

Then, we have $opt(R_{34})$ = 9.9 and $f(opt(R_{34})) = DB_1$.
The contents of $dt\text{-}table_T$ are changed to:

$$
\begin{aligned}
&dt\text{-}table_T(DB_1) = (9.9);\ dt\text{-}table_T(DB_2) = (8.0);\\
&dt\text{-}table_T(DB_3) = (\ );\ dt\text{-}table_T(DB_4) = (\ ).
\end{aligned}
$$

Lastly, $opt\text{-}rt(R, DB_i)$ (i = 1, 2, 3, 4) can be computed in the same way as above:

$$
\begin{aligned}
&opt\text{-}rt(R, DB_1) = max\{max\{c_{f(opt(R_{12})),1} \mid R_{12} \mid + opt(R_{12})\},\\
&\quad max\{c_{f(opt(R_{34})),1} \mid R_{34} \mid + opt(R_{34})\}, dt\text{-}table_T(DB_1)[\text{last}]\} + t_1\\
&= max\{max\{2 \times 10^{-4} \times 15000 + 8.0,\ 0 + 9.9\}\ 9.9\} + 7.5 = 18.5;\\
&opt\text{-}rt(R, DB_2) = max\{max\{c_{f(opt(R_{12})),2} \mid R_{12} \mid + opt(R_{12})\},\\
&\quad max\{c_{f(opt(R_{34})),2} \mid R_{34} \mid + opt(R_{34})\}, dt\text{-}table_T(DB_2)[\text{last}]\} + t_2\\
&= max\{max\{0 + 8.0,\ 2 \times 10^{-4} \times 9000 + 9.9\}, 8.0\} + 6.0 = 17.7;\\
&opt\text{-}rt(R, DB_3) = max\{max\{c_{f(opt(R_{12})),3} \mid R_{12} \mid + opt(R_{12})\},\\
&\quad max\{c_{f(opt(R_{34})),3} \mid R_{34} \mid + opt(R_{34})\}, dt\text{-}table_T(DB_3)[\text{last}]\} + t_3\\
&= max\{max\{1 \times 10^{-4} \times 15000 + 8.0,\ 3 \times 10^{-4} \times 9000 + 9.9\}\ 0\} + 9.33 = 21.93;\\
&opt\text{-}rt(R, DB_4) = max\{max\{c_{f(opt(R_{12})),4} \mid R_{12} \mid + opt(R_{12})\},\\
&\quad max\{c_{f(opt(R_{34})),4} \mid R_{34} \mid + opt(R_{34})\}, dt\text{-}table_T(DB_4)[\text{last}]\} + t_4\\
&= max\{max\{2 \times 10^{-4} \times 15000 + 8.0,\ 3 \times 10^{-4} \times 9000 + 9.9\}\ 0\} + 1.0 = 22.6;
\end{aligned}
$$

Similarly, $opt(R)$ = 17.7 and $f(opt(R)) = DB_2$ can be derived, and $dt\text{-}table_T$ is now of the form:

$$
\begin{aligned}
&dt\text{-}table_T(DB_1) = (9.9);\ dt\text{-}table_T(DB_2) = (8.0, 17.7);\\
&dt\text{-}table_T(DB_3) = (\ );\\
&dt\text{-}table_T(DB_4) = (\ ).
\end{aligned}
$$

REFERENCES

1. C. Baru et al, "An overview of DB2 parallel edition," in *Proceedings of ACM SIGMOD*, 1995, pp. 460-462.
2. BBMS98 M. L. Barja, T. Bratvold, J. Myllymaki, and G. Sonnenberger, "Informia: a mediator for integrated access to heterogeneous information sources," in *Proceedings of 7th International Conference on Information and Knowledge Management (CIKM), ACM*, 1998, pp. 234-241.
3. Y. Breitbart, P. Olson, and G. Thompsom, "Database integration in a distributed heterogeneous database system," in *Proceedings of 2nd IEEE Conference on Data Engineering*, 1986, pp. 301-310.
4. W. Benn, Y. Chen, and I. Gringer, *A Rule-based Strategy for Schema Integration in a Heterogeneous Information Environment*, Technical Report/CSR-96-01, Computer Science Department, Technical University of Chemnitz-Zwickau, Germany, 1996.
5. Y. Chen and W. Benn, "On the query treatment in federated systems," in *Proceedings of 8th International DEXA Conference on Database and Expert Systems Application*, 1997, pp. 583-592.
6. Y. Chen and W. Benn, "A systematic method for query evaluation in federated relational databases," Technical Report, CSR-97-03, Computer Science Department, Technical University of Chemnitz-Zwickau, Germany, 1997.
7. M. S. Chen, M. L. Lo, P. S. Yu, and H. C. Young, "Applying segmented right-deep trees to pipelining multiple hash joins," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, 1995, pp. 656-668.
8. I. A. Chen and D. Rotem, "Integrating information from multiple independently developed data sources," in *Proceedings of 7th International Conference on Information and Knowledge Management (CIKM), ACM*, 1998, pp. 242-250.
9. M. S. Chen and P. S. Yu, "A graph theoretical approach to determine a join reducer sequence in distributed query processing," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 1, 1995, pp. 152-165.
10. M. S. Chen, P. S. Yu, and K. L. Wu, "Optimization of parallel execution for multi-join queries," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 3, 1996, pp. 416-428.
11. S. Ceri and J. Widom, "Managing semantic heterogeneity with production rules and persistent queues," in *Proceedings of 19th International Very Large Data Base Conference*, 1993, pp. 108-119.
12. W. Du, R. Krishnamurthy, and M. Shan, "Query optimization in heterogeneous DBMS " in *Proceedings of 18th International Very Large Data Base Conference*, 1992, pp. 277-291.
13. W. Du, M. Shan, and U. Dayal, "Reducing multidatabase query response time by tree balancing," DTD Technical Report, Hewlett-Packard Labs., 1994.
14. W. Du, M. Shan, and U. Dayal, "Reducing multidatabase query response time by tree balancing," in *Proceedings of 15th International ACM SIGMOD Conference on Management of Data*, 1995, pp. 293-303.
15. Y. Dupont, "Resolving fragmentation conflicts schema integration," in *Proceedings 13th International Conference on the Entity-Relationship Approach*, 1994, pp. 513-532.

16. C. Evrendilek, A. Dogac, S. Nural, and F. Ozcan, "Multidatabase query optimization, " *International Journal of Distributed and Parallel Databases*, Vol. 5, No. 1, 1997, pp. 77-114.
17. C. J. Egyhazy, K. P. Triantis, and B. Bhasker, "A query processing algorithm for a system of heterogeneous distributed databases," *International Journal of Distributed and Parallel Databases*, Vol. 4, No. 1, 1996, pp. 49-79.
18. G. Harhalakis, C. P. Lin, L. Mark, and P. R. Muro-Medrano, "Implementation of rule-based information systems for integrated manufacturing," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 6, 1994, pp. 892-908.
19. W. Hasan and R. Motwani, "Coloring away communication in parallel query optimization," in *Proceedings of 21st International Very Large Datab Base Conference*, 1995, pp. 239-250.
20. W. Hong and M. Stonebraker, "Optimization of parallel query execution plans in XPRS," *International Journal of Distributed and Parallel Databases*, Vol. 1, No. 1, 1993, pp. 9-32.
21. J. L. Koh and Arbee L.P. Chen, "Integration of heterogeneous object schemas," in *Proceedings of 12th International Conference on the Entity-Relationship Approach*, 1993, pp. 297-314.
22. R. Krishnamurthy, W. Litwin, and W. Kent, "Language features for interoperability of databases with schematic discrepancies," in *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1991, pp. 40-49.
23. P. Johannesson, "Using conceptual graph theory to support schema integration," in *Proceedings of 12th International Conference on the Entity-Relationship Approach*, 1993, pp. 283-296.
24. W. Litwin and A. Abdellatif, "Multidatabase interoperability," *IEEE Computing Magazine*, Vol. 19, No. 12, 1986, pp. 10-18.
25. J. M. Lucas, D. R. van Baronaigien, and F. Ruskey, "On rotation and the generation of binary trees," *International Journal of Algorithms*, Vol. 15, No. 1, 1993, pp. 343-366.
26. A. Lefebvre, P. Bernus, and R. W. Topor, "Querying heterogeneous databases: a case study," in *Proceedings of 3rd Australian Database Conference*, 1993, pp. 186-198.
27. C. Lee, C. Chen, and H. Lu, "An aspect of query optimization in multidatabase systems, " *SIGMOD Record*, Vol. 24, No. 3, 1995, pp. 28-33.
28. 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, No. 5, 1995, pp. 533-562.
29. J. W. Lloyd, *Foundation of Logic Programming*, Springer-Verlag, Berlin, 1987.
30. M. L. Lo, M. S. Chen, C. V. Ravishankar, and P. S. Yu, "On optimal processor allocation to support pipelined hash joins," in *Proceedings of ACM SIGMOD*, 1993, pp. 69-78.
31. H. Lu, B. Ooi, and C. Goh, "Multidatabase query optimization: issues and solutions," in *Proceedings of 3rd International Workshop on Research Issues in Data Engineering*, 1993, pp. 137-143.
32. C. LEE and M. Wu, "A hyperrelational approach to integration and manipulation of date in multidatabase systems," *International Journal of Cooperative Information Systems*, Vol. 5, No. 4, 1996, pp. 395-429.

33. J. Martin, *Systems Analysis for Data Transmission*, New Jersey, Prentice-Hall, 1972.
34. B. Reinwald, H. Pirahesh, G. Krishnamoorthy, and G. Lapis, "Heterogeneous query processing through SQL table function," in *Proceedings of 15th International Conference on Data Engineering, IEEE*, 1999, pp. 366-373.
35. M. P. Reddy, B. E. Prasad, P. G. Reddy, and A. Gupta, "A methodology for integration of heterogeneous databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 6, 1994, pp. 920-933.
36. P. Selinger, M. M. Astrahan, D. D. Chamberling, A. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1979, pp. 23-34.
37. S. Salza, G. Barone, and T. Morzy, "Distributed query optimazation in loosely coupled multidatabase systems," in *Proceedings of 5th International Conference on Database Theory*, 1995, pp. 40-53.
38. W. Sull and R. L. Kashyap, "A self-organizing knowledge representation schema for extensible heterogeneous information environment," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 2, 1992, pp. 185-191.
39. S. Spaccapietra, P. Parent, and Y. Dupont, "Model independent assertions for integration of heterogeneous schemas," *VLDB Journal*, Vol. 1, No. 1, 1992, pp. 81-26.
40. S. Spaccapietra and P. Parent, "View integration: a step forward in solving structural conflicts," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 2, 1994, pp. 258-274.
41. J. L. Wolf, D. M. Dias, and P. S. Yu, "A parallel sort merge join algorithm for managing data skew," *IEEE Parallel and Distributed Systems*, Vol. 4, No. 1, 1993, pp. 70-86.
42. H. Yoo and S. Lafortune, "An intelligent search method for query optimization by semijoins," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 2, 1989, pp. 226-237.
43. Q. Zhu and P. Larson, "A query sampling method for estmating local cost parameters in a multidatabase system," in *Proceedings of 10th International Conference on Data Engineering*, 1994, pp. 144-153.

Yangjun Chen (³¯¶§÷⊗ received his BS degree in information system engineering from the Technical Institute of Changsha, China, in 1982, and his Diploma and Ph.D. degrees in computer science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as an assistant professor at the Technical University of Chemnitz-Zwickau, Germany. Dr. Chen is currently a senior engineer at German National Research Center of Information Technology. His research interests include deductive databases, federated databases, multimedia databases, constraint satisfaction problem, graph theory and combinatorics. He has more than 50 publications in these areas.