Data Allocation in Distributed Database Systems

PETER M. G. APERS
Vrije Universiteit

The problem of allocating the data of a database to the sites of a communication network is investigated. This problem deviates from the well-known file allocation problem in several aspects. First, the objects to be allocated are not known a priori; second, these objects are accessed by schedules that contain transmissions between objects to produce the result. A model that makes it possible to compare the cost of allocations is presented; the cost can be computed for different cost functions and for processing schedules produced by arbitrary query processing algorithms.

For minimizing the total transmission cost, a method is proposed to determine the fragments to be allocated from the relations in the conceptual schema and the queries and updates executed by the users.

For the same cost function, the complexity of the data allocation problem is investigated. Methods for obtaining optimal and heuristic solutions under various ways of computing the cost of an allocation are presented and compared.

Two different approaches to the allocation management problem are presented and their merits are discussed.

Categories and Subject Descriptions: C.2.4 [**Computer Communication Networks**]: Distributed Systems]; D.2.8 [**Software Engineering**]: Metrics; H.2.2 [**Database Management**]: Physical Design; H.2.4 [**Database Management**]: Systems

General Terms: Algorithms, Design, Measurements, Theory

Additional Key Words and Phrases: Allocation, complexity analysis, database, greedy method, partitioning

1. INTRODUCTION

The demand for more and more information both by industry and government leads to databases that will exceed the physical limitations of centralized systems and to the integration of already existing databases, which may be geographically dispersed. Advances in the areas of both computer networks and databases make it possible to build these distributed databases. Computers can easily be connected to form a network, making it possible for them to communicate with each other. On top of such a network a distributed database management system can be built in such a way that the distribution of logical and physical components of the databases is kept hidden from the users.

Author's address: Computer Science Department, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.

The advantages of a distributed database compared with a centralized one are increased availability, decreased access time, easy expansion, and possible integration of existing databases [1, 37]. The acceptance and widespread usage of distributed databases will highly depend on their efficiency. Therefore, it is important to supply a database management system with tools to efficiently process queries and to determine allocations of the data such that the availability is increased, the access time is decreased, and/or the overall usage of resources is minimized.

The problem of allocating files in a computer network has been extensively studied. However, the results obtained cannot be straightforwardly applied to distributed databases. One deviation from the file allocation problem is the unit of allocation. Tuples occur in the same relation because they contain data from the same set of domains to describe the same entity or relationship. However, one group of tuples may be mainly used in New York and another group in Amsterdam. Obviously, splitting the relation into fragments and locating one fragment on one side of the ocean and the other on the other side will tremendously decrease intercontinental traffic.

Another deviation from the file allocation problem is the way the relations are accessed. From current research on distributed query processing we know it is common that more than one relation is accessed in a query and that complex processing schedules, which include transmissions between relations stored at different sites, are used.

In this paper we will present a model that makes it possible to compare the cost of possibly not yet completely specified allocations for schedules produced by arbitrary query processing algorithms. The model is general enough to be used in both branch-and-bound and heuristic algorithms for minimizing various cost functions. For minimizing the total transmission cost, a method is proposed to determine the fragments to be allocated from the relations in the conceptual schema and the queries and updates executed by the users. Under restrictive conditions it can be shown that these fragments are the smallest ones that have to be considered. For the same cost function, it is shown that the problem of determining a nonredundant data allocation to minimize total transmission cost is NP-hard. Methods for obtaining optimal and heuristic solutions under various ways of computing the cost of an allocation are presented and compared. Centralized and decentralized approaches to the problem of managing data allocations are presented and their merits are discussed.

This paper is organized as follows. In Section 2 an overview is given of previous work in the area of file and data allocation. The differences between the well-known file allocation problem and the data allocation problem in distributed databases are discussed in Section 3. Section 4 contains the introduction of a model to compute the cost of allocations, the way updates are treated, and a way of determining the fragments to be allocated. In Sections 5 and 6 the problem of computing optimal and heuristic data allocations when using a static approach to compute the cost of an allocation are investigated. In Section 7 the same is done for a dynamic approach. Static or dynamic refers to adjusting or not adjusting the query processing schedules when computing the costs of various allocations in the process of determining a final allocation. Section 8 discusses

the problem of managing data allocations by one or several database administrators. We end with a summary of the results obtained.

2. OVERVIEW OF PREVIOUS RESEARCH ON THE FILE AND DATA ALLOCATION PROBLEM

The file allocation problem has many disguises. In this section we will not attempt to cover all the related research; only the main line of research will be discussed. For a more complete discussion of the file allocation problem we refer to [24].

Before distributed database management systems were investigated, networks already existed for many years. The problem of where to allocate a file and its copies, given a known set of retrievals and updates and their execution frequencies such that a cost function is minimized, is known as the *file allocation problem.*

Chu was probably the first to work on the file allocation problem. In [17, 18] he presented a simple model that only allows for a nonredundant allocation of the files. The optimization goal is to minimize total transmission cost subject to available secondary storage at each site and a given maximum on the expected retrieval time. The result is a zero-one programming problem subject to nonlinear constraints, which can be solved with standard linear integer programming techniques.

The model proposed by Casey [12] allows for multiple copies. To do so, a distinction must be made between queries and updates, because an update must access all copies and a query needs to access only one. The optimization goal is to minimize the cost in dollars of the transmissions plus the storage cost of the files. In [21] it was shown that the file allocation problem modeled this way is NP-complete [4, 19, 22].

In [28, 29] both the allocation of the application programs that access the files and the allocation of files themselves were discussed. Data can be stored relatively easily at different sites or transmitted from one site to another. However, programs, because of the programming language in which they are written, are not as portable as one might wish. A second important aspect discussed by Levin and Morgan is the change in the access pattern over time.

Another approach to the file allocation problem is to allow for changes in the hardware as well as in the allocation of the files. In [30] the capacity of the communication channels may be determined besides the allocation of the files. The resulting model is a nonlinear integer programming problem for which a heuristic approach is used to reach a solution.

In [36] an attempt is made to consider the file allocation in the environment of a distributed database. Although queries that access more than one relation are allowed, the underlying assumption—that the query is processed at the result site without transmissions between the sites where the relations are located—reduces the whole problem again to the file allocation problem.

In [5, 6, 8] we considered the integration of query processing and data allocation. A heuristic algorithm was developed to handle both the nonredundant and redundant case. Determining the unit of allocation was discussed and different ways of managing data allocations were compared in [7, 8].

Later, in [13, 14], a similar goal was pursued in the context of a general distributed database design methodology. In [14], only predefined schedules consisting of a number of basic operations on relations were considered.

In [38], limited processing and network capacities are assumed; transmission of the result of a query to the result site is not considered; only transmissions between fragments are considered, rendering the problem into a cluster analysis problem.

In [41] the file allocation problem is discussed for a local area network with broadcasting facilities. It turns out that some of the NP-hard problems can be solved in polynomial time for such an environment.

3. ALLOCATION PROBLEM IN DISTRIBUTED DATABASES

In this section we will discuss and elaborate on the reasons why the file allocation problem does not adequately describe the allocation problem in distributed databases and state the data and operation allocation problem.

A distributed file system differs greatly from a distributed database. The solutions for the file allocation problem do not characterize solutions to the allocation problem in a distributed database for the following reasons:

—The objects to be allocated are not known prior to allocation. Relations, which describe logical relationships between data, are not suited as units of allocation because users at different sites might be interested in different fragments of a relation.

—The way the data are accessed is far more complex. In the file allocation problem the only transmissions required to combine data from different files are transmissions from sites containing files to the result site, where the result is computed. In current research on distributed query processing we observe that to process a query, data transmissions between sites where fragments are allocated are also needed. This means that the fragments cannot be allocated independently.

To capture these aspects, the file allocation problem is generalized into the *data and operation allocation problem*:

> *Given the queries and updates, the frequencies of their usage, and the sites where the results have to be sent, determine* (1) *the fragments to be allocated, and* (2) *allocate these fragments, possibly redundant, and the operations on them to the sites of the computer network such that a certain cost function is minimized.*

For short, we will often use the term *data allocation* when we mean data and operation allocation.

Before going on we will elaborate on the above-mentioned deviations from the file allocation problem.

3.1 Data Allocation

The various ways of splitting a relation into fragments and the terminology for the different allocations will be discussed in this section.

Grouping together complete tuples is called *horizontal splitting*, and grouping together attribute values of all tuples is called *vertical splitting*. If the relations

are split horizontally and/or vertically and the resulting fragments are placed at different locations, the database is called *partitioned.* If copies of relations or fragments are placed at several locations, the database is called *replicated.*

Example 1. Figure 1(a) shows the relation *WINE* with attributes *YEAR, NAME, PRODUCER, AREA,* and *COUNTRY.* Each tuple represents a wine for which the grapes were grown in a certain area, picked in a certain year, and bottled by a certain producer.

Figure 1(b) shows the relation *WEATHER,* containing the attributes *YEAR, AREA, COUNTRY, SUN, and RAIN. SUN* stands for the hours of sun and *RAIN* for the number of millimeters of rain in a particular area in a particular year. The two relations will be used in the examples to come.

One way to partition the relation *WINE* is to split it based on countries that produce wines, assuming those are France, Italy, and the USA:

$$WINE\_F = WINE\{COUNTRY = \text{France}\}$$
$$WINE\_I = WINE\{COUNTRY = \text{Italy}\}$$
$$WINE\_U = WINE\{COUNTRY = \text{USA}\}.$$

Locating *WINE_F* in Paris, *WINE_I* in Rome, and *WINE_U* in San Francisco is an example of a partitioned allocation.

An example of a vertical split is

$$WEATHER\_R = WEATHER[YEAR, AREA, COUNTRY, RAIN]$$
$$WEATHER\_S = WEATHER[YEAR, AREA, COUNTRY, SUN].$$

Locating *WEATHER_R* in Oslo, *WEATHER_S* in Rome, and *WEATHER* in New York is an example of a partitioned and replicated allocation.

Note that the primary key of relation *WEATHER,* which is *YEAR, AREA,* and *COUNTRY,* is part of both fragments *WEATHER_R* and *WEATHER_S*; this is necessary to be able to update both fragments.

3.2 Query Processing

To get an idea of the problems involved in distributed query processing, we will discuss some of the problems involved. Assume we want to process the query:

> *Give the name and the year of wines and the hours of sun of areas where the grapes were picked and where more than 1,700 mm. of rain fell*

stated by a user in Amsterdam.

Some of the distributed query processing algorithms require that the database management system supplies a *materialization* of the fragments. This means that for each fragment a single copy has to be selected such that together with other copies a consistent view of the database is given. Here we assume that the materialization looks like: fragment $(WEATHER\{RAIN > 1{,}700\})[YEAR, SUN, AREA]$ at the site in New York and the fragments *WINE_F, WINE_I,* and *WINE_U* in Paris, Rome, and San Francisco, respectively.

The query may have many *processing schedules* for executing it, of which we will discuss only two.

WINE

YEAR	*NAME*	*PRODUCER*	*AREA*	*COUNTRY*
1970	Margaux	Chateau Margaux	Bordeaux	France
1972	Beaune	Louis Latour	Bourgogne	France
1978	Chianti Classico	Villa Antinori	Toscana	Italy
1976	Cabernet Sauvignon	Christian Brothers	Napa Valley	USA

(a)

WEATHER

YEAR	*AREA*	*COUNTRY*	*SUN*	*RAIN*
1970	Ardennes	Belgium	1551	1105
1976	Napa Valley	USA	3022	601
1970	Bordeaux	France	2008	900

(b)

Fig. 1. (a) relation *WINE* and (b) relation *WEATHER*.

Schedule 1. Transmit $(WEATHER\{RAIN > 1{,}700\})[YEAR, SUN, AREA]$ from New York to Paris, Rome, and San Francisco, and compute the joins based on *YEAR* and *AREA* at the respective locations. After that, the results are transmitted to Amsterdam. If the size of the selected and projected relation *WEATHER* is 18,000 and the sizes of the results are 400, 800, and 200 bytes, respectively, the total number of bytes transmitted is $3 \times 18{,}000 + 400 + 800 + 200 = 55{,}400$.

Schedule 2. Transmit the fragments *WINE_F*, *WINE_I*, and *WINE_U* to New York, where they are united and the join based on *YEAR* and *AREA* is computed between this union and *WEATHER*. If the sizes of the three fragments of the relation *WINE* are 12,000, 15,000, and 20,000, respectively, and the size of the result is 1,400 bytes, the total number of bytes transmitted is $12{,}000 + 15{,}000 + 20{,}000 + 1{,}400 = 48{,}400$.

Clearly, the first schedule is more expensive in terms of the number of bytes transmitted; however, most of the transmissions and computations are done in parallel, resulting in a smaller response time.

The purpose of query processing algorithms is to determine processing schedules for queries such that a certain cost function is minimized. There are different ways of measuring the cost of a schedule. The cost function may include the cost to transmit data and/or the cost to execute a certain operation. Here we will confine ourselves to the *total transmission cost*. This cost function just adds up the costs of all the data transmitted in the schedule. As far as the model introduced in the following sections is concerned, this confinement is merely for presentational reasons, because the model does not depend on it. The methods, however, are especially designed for minimizing the total amount of data transmitted.

To correctly represent processing schedules we also need to know something about executing them. The processing schedules contain data transmissions from one site to another and local processing at the different sites. To let the individual operations and transmissions cooperate in the way described in the schedule we

require synchronization and forking processes before and after every operation and transmission. A *synchronization process* lets an operation wait until its input(s) are completely or partly available. For example, the union of the fragments *WINE_F*, *WINE_I*, and *WINE_U* in Schedule 2 may start its execution if its operands are only partially locally available. A *forking process* allows the result of an operation or transmission to be the input of one or more other operations or transmissions. For example, the forking process after the selection and projection on the relation *WEATHER* creates copies of the result and gives them to the synchronization processes of the transmissions for transmission to different sites.

For a more detailed discussion on distributed query processing we refer to the current research on this topic [9–11, 20, 23, 26, 32, 34, 39, 40, 42–44].

4. UNIT, REPRESENTATION, AND COST OF DATA ALLOCATIONS

In Subsections 4.1 and 4.2, notions are introduced to represent allocations and to compute their costs. In Subsection 4.3 the cost of updating several copies is discussed. A way of determining the unit of allocation is proposed in Subsection 4.4.

4.1 Representation of Data Allocations

In this section we introduce some notions to describe a model to represent data allocations. We assume that the unit of allocation is a fragment.

To allocate the fragments we have to know the processing schedules of all the queries and updates that access these fragments. However, these schedules depend on the allocation of the fragments that we want to determine. One way of solving this circular problem is to do an exhaustive search to find an optimal allocation. For a large number of fragments this is not feasible. Therefore, the representation model should be general enough to allow for both heuristic and branch-and-bound approaches. Both approaches have in common that allocations in which only part of the data is allocated can occur. To represent these allocations and compare their costs we introduce some notions.

A *nucleus-site* is a pair (FS, OS), where FS is a set of fragments and OS is a set of operations. An *operation* is a triple (i, f, x), where x is the execution time of the operation and f the frequency with which the ith transaction of which the operation is part is executed. A *transaction* consists of a set of operations. There are two types of nucleus-sites, namely *physical sites* (PhS) and *virtual sites* (VS). A physical site represents a site in the computer network and a virtual site represents a fictitious site, the purpose of which will be explained in a moment. Both types of nucleus-sites are used to represent allocations.

Putting fragment F in the set of fragments of PhS corresponds to allocating F to the site in the computer network corresponding to PhS. If part of the allocation looks like $VS = (\{F_1, F_2\}, \{\})$, it represents that fragments F_1 and F_2 are to be allocated to the same site in the computer network but that which site is not specified. Below we will introduce two operations on allocations and discuss their differences.

A physical site may have assigned to it a set of virtual sites; this set will be called the *assigned set*. A virtual site can be *assigned* to at most one physical site, which means that it is placed in the assigned set of that physical site.

The *union* of two nucleus-sites is a nucleus-site whose set of fragments is the union of the sets of fragments of the two nucleus-sites, whose set of operations is the union of the sets of operations, and whose assigned set is the union of the assigned sets. The result of a union of two virtual sites is again a virtual site, and the union of a virtual site and a physical site is that physical site. Note that the union between two physical sites is not defined.

The difference between an assignment and a union is that applying the union to an allocation means changing it permanently. The assignment, however, allows for changing the allocation back and forth. To make the assignment look like a temporary union, the cost of an allocation, which will be defined later, should not be different whether a virtual site is assigned to or united with a physical site. These two operations make it possible to search through the space of possible allocations by branch-and-bound techniques and heuristic algorithms.

A *completely specified allocation* is an allocation in which all fragments and operations are in sets of physical sites. A *partially specified allocation* is one where some of the fragments or operations are still in the corresponding sets of virtual sites.

4.2 The Cost of a Data Allocation

In this subsection we will introduce tools to compute the cost of a completely and partially specified allocation.

The computation of the cost of an allocation is done by means of a *processing-schedules graph.* Such a multigraph consists of

(1) PhS-nodes, for the physical sites,

(2) VS-nodes, for the virtual sites, and

(3) edges, for the data transmission between two nodes, PhS- or VS-nodes.

The edges, which are directed, are labeled with a triple (i, f, d), where d stands for the amount of data transmitted between the sites that correspond to the nodes in the processing-schedules graph for processing the ith transaction, and f stands for the frequency with which this transaction is executed. Because most of the time we are interested in the processing-schedules graph and not merely in the allocation itself, we will talk about the nodes in the processing-schedules graph as if they were the physical or virtual sites themselves.

First, we show how to construct a processing-schedules graph and how it is graphically represented, and then we show how this construction is used to compute the cost of an allocation. The basic idea is that given an allocation we can *construct* a processing-schedules graph by (1) creating PhS- and VS-nodes for the physical and virtual sites, respectively; (2) creating edges that represent the transmissions of the processing schedules, which are computed by an arbitrary query processing algorithm based on the allocation; and (3) adding the operations of the processing schedules to the operation-sets of the physical and virtual sites.

The processing schedules of the queries and updates are computed based on the allocation of the fragments distributed over a hypothetical network. This hypothetical network has a site for every physical and virtual site, and they are connected with each other by communication channels of the same bandwidth; however, if *VS* is assigned to *PhS* we assume that sending data between *VS* and

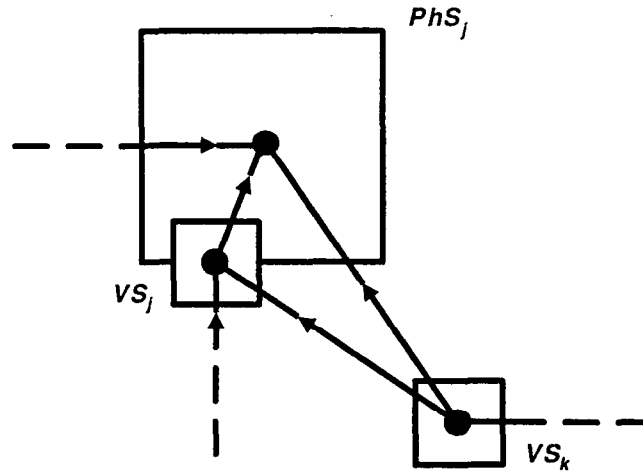


Fig. 2. Graphical representation of a physical site with virtual sites.

PhS does not cost anything. It is just as if the data of *VS* and *PhS* were stored at the same site.

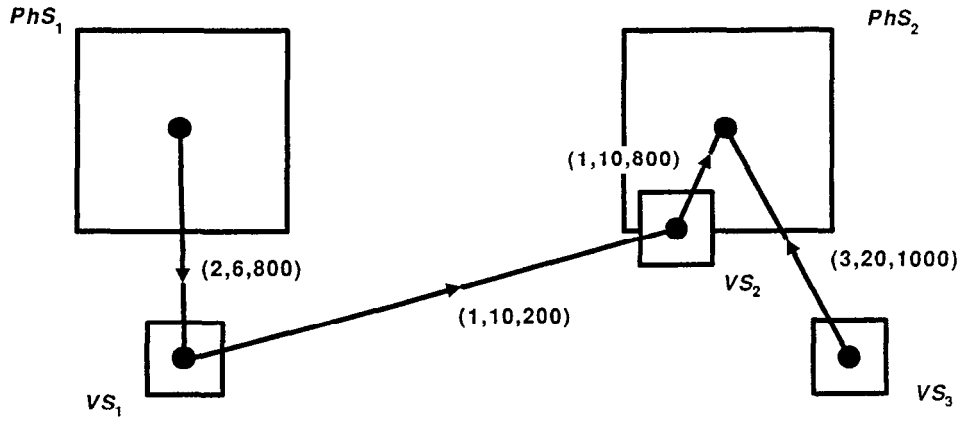
The node of a nucleus-site in the processing-schedules graph is *graphically represented* by a box; in this box there is a black dot representing the nucleus-site itself. The boxes that represent the elements of the assigned set are placed on the edges of the box such that they do not overlap with each other. Figure 2 shows part of a processing-schedules graph with one physical site, PhS_i, and two virtual sites, VS_j and VS_k, of which VS_j is assigned to PhS_i.

Example 2. Figure 3 shows a processing-schedules graph of a partially specified allocation for three transactions. There are two physical sites, PhS_1 and PhS_2, and three virtual sites, VS_1, VS_2, and VS_3, of which VS_2 is assigned to PhS_2. Transaction 1, which is executed ten times per unit of time, computes a join between F_1, allocated to VS_1, and F_2, allocated to VS_2. A distributed query processing algorithm can determine that sending data between VS_2 and PhS_2 does not cost anything. Its processing schedule consists of the selections O_1 and O_2, which are elements of the sets of operations of VS_1 and VS_2, respectively. The result of O_1, whose size is 200 bytes, is transmitted to VS_2, where the join (O_3) with the result of O_2 is computed. Finally, this result, 800 bytes in size, is sent to the physical site PhS_2.

Transaction 2, which is executed six times per unit of time, represents updates (O_4) by a user at PhS_1 of fragment F_1, which is allocated to VS_1. The edge from PhS_1 to VS_1 represents the transmission of the actual changes supplied by the user.

Transaction 3 retrieves (O_5) data from F_3, allocated to VS_3. It is executed 20 times per unit of time.

To compute the *cost of a completely (partially) specified allocation* we start by constructing the processing-schedules graph based on the allocation and the processing schedules. This processing-schedules graph contains all the



$$VS_1 = (\{F_1\},\{O_1,O_4\})$$
$$VS_2 = (\{F_2\},\{O_2,O_3\})$$
$$VS_3 = (\{F_3\},\{O_5\})$$

Fig. 3. An example of a processing-schedules graph.

information necessary to check constraints, such as bandwidth, CPU-utilization, availability, etc., and to determine the cost of the individual queries and updates.

For example, for each communication channel and each CPU, all transmissions and operations are known. Based on that, the expected waiting times for the channels can be computed. For each physical site the expected waiting time of its operations is determined as if all the sets of operations of the virtual sites that are assigned to it were united. The expected waiting time of the operations of a virtual site *VS* that is assigned to a physical site *PhS* is computed as if the set of operations of only *VS* and *PhS* were united. Note that although more than one virtual site may be assigned to a physical site, the expected waiting times of their operations are computed for each virtual site independently.

Possible costs of an allocation could be the sum of the total transmission costs of queries and updates weighted by their execution frequencies, the average response time of queries and updates, etc., possibly subject to one of the constraints mentioned above. If a constraint is violated the cost of the corresponding allocation is infinite.

In this paper we will confine ourselves to the total transmission cost of an allocation defined as the sum of the total transmission costs of queries and updates weighted by their execution frequencies. We will show how this can be computed. For each transaction i we extract from the processing-schedules graph the transmissions labeled with the identification number i that are not connecting a physical site with one of its assigned virtual sites. This value is multiplied by the execution frequency of transaction i and summed up by the total cost. Note the algorithmic way the cost of the allocation is computed.

Example 3. From the processing-schedules graph of the partially specified allocation discussed in Example 2, we will construct the processing schedules of

the three transactions, taking into account the assignment of VS_2 to PhS_2. Because VS_1 and VS_3 are not assigned to any physical site the adjacent edges will be part of the schedules of the queries and updates. For simplicity, we confine ourselves to the transmissions in the processing schedules.

Transaction 1.

$F_1 \xrightarrow{200}$ result site

total transmission cost = 200

Transaction 2.

result site $\xrightarrow{800} F_1$

total transmission cost = 800

Transaction 3.

$F_3 \xrightarrow{1,000}$ result site

total transmission cost = 1,000

This results in a total transmission cost of the allocation of $10 \times 200 + 6 \times 800 + 20 \times 1{,}000 = 26{,}800$.

In this subsection the costs of completely and partially specified allocations were computed by means of a processing-schedules graph. This graph was constructed by giving the allocation to a query-processing algorithm that returns a processing schedule for each query. The transmissions and operations in such a schedule are incorporated in the processing-schedules graph and the physical and virtual sites.

4.3 Forking Processes and Forking Graphs

So far, we have discussed the way a processing-schedules graph can be constructed given a partially specified allocation and how the cost of such an allocation can be determined from it. Our goal is to obtain a completely specified allocation by manipulating partially specified allocations such that a given cost function is minimized. Changing an allocation may have an effect on the processing schedules of the transactions.

The placement of forking processes in a schedule depends on the allocation. Therefore, we will introduce a *forking graph*, which enables us to more efficiently handle forking processes when changing partially specified allocations. The cases in which forking processes are used are listed below. The first case is concerned with the notification of the processing schedule of a query or update to all sites involved; the second case concerns the notification of the tuples to be updated to the copies of a fragment in an update transaction. A third case will be seen when discussing the splitting of relations. All cases have in common that the forking process is used to start a parallel computation. The representation of a forking graph is shown in Figure 4. Such a forking graph will be a subgraph of a processing-schedules graph and consists of a *notification node* and a set of *receiving nodes*. All the nodes are VS-nodes. All edges in a forking graph are labeled with the same triple, because to each receiving site the same amount of data will be transmitted with the same frequency. Each receiving node is part of

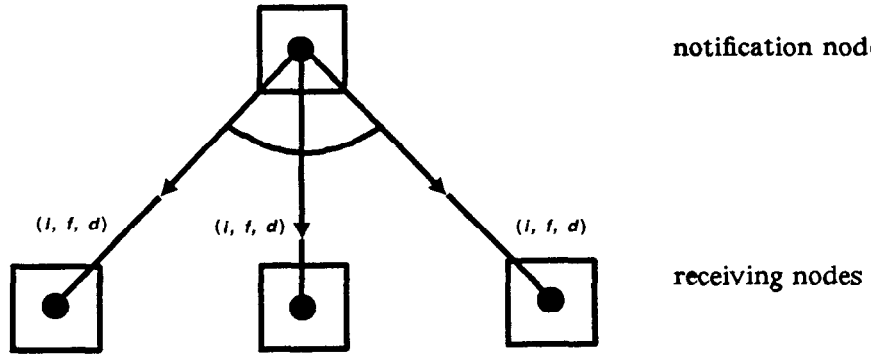


Fig. 4. Representation of a forking-graph.

a schedule for a query that references the fragment allocated to that receiving node.

Imagine that this forking graph is part of an update schedule. After the tuples that have to be changed are determined, the actual changes are sent to the copies of the fragment. At the notification node the changes are computed and the copies are located at the receiving nodes.

What kind of changes can occur in the allocation? Two copies that were located at different virtual sites can be allocated to the same virtual site. This means that the two virtual sites are united and two copies of the same fragment are put in the fragment-set of the resulting virtual site. Having two identical copies at one nucleus-site is, as far as efficiency is concerned, useless and, therefore, only one is maintained. If in a forking-graph two receiving nodes are united, one of the edges to these nodes disappears. Also, if one of the copies of the fragment is allocated to the site corresponding to the notification node, there is no need to transmit data to it. Therefore, if a receiving node is united with a notification node, the edge between them is deleted.

Besides the removal of an edge representing a superfluous transmission, operations directly involved with this transmission and operations that worked on superfluous copies are also removed from the operation-sets.

4.4 Unit of Allocation and Processing Schedules

Having explained how a processing-schedules graph for a given allocation can be constructed and how it is used to compute the cost of that allocation, we will now discuss how to determine the unit of allocation. In [7] we gave a global outline of the splitting algorithm. Later, in [15] a similar goal was pursued; in [31] only vertical partitioning is considered.

Let us assume that we have a set of queries and updates and the frequencies of their usage. As far as the relational operations are concerned, queries and updates are the same and therefore we only discuss queries. A query will use only fragments of the relations in the global conceptual schema. These fragments are characterized by selections and projections.

First, we take a look at just one relation, say R. A selection is a Boolean expression of a number of simple clauses $A\ \theta\ a$, where A is an attribute, θ is a

comparison operation, and a is a value from the domain of A. Each of these Boolean expressions describes a subset of the tuples of the relation R. We are interested in the intersections, caused by overlapping subsets, which can be uniquely identified by saying in which selections they participate.

The partition caused by the overlapping subsets form the *horizontal split* of R. If there are n selections there will, in general, be $(2^n - 1)$ fragments. The *vertical split* of R is done based on the attributes. Each attribute together with the primary key is put in a separate fragment. The reason for this will become clear after Theorem 2. If there are m attributes of which k form the primary key, there will be $m - k + 1$ fragments after vertical splitting. Combine the above results: The number of fragments caused by horizontal and vertical splits equals $(2^n - 1)(m - k + 1)$, where n is the number of selections, m is the number of attributes, and k is the number of attributes in the primary key.

Example 4. Assume we have relation *WEATHER* with relational schema:

WEATHER(*YEAR, AREA, SUN, RAIN*).

Also the following queries are given:

Q_1 = *WEATHER*{*AREA* = Bordeaux AND *SUN* > 2,000}[*YEAR, AREA, SUN*]
Q_2 = *WEATHER*{*RAIN* < 1000}[*YEAR, AREA, RAIN*].

From the selections in these queries and the attributes in the schema we determine the following fragments:

F_1 = *WEATHER*{*AREA* = Bordeaux *AND SUN* > 2,000
AND RAIN ≥ 1,000}[*YEAR, AREA*]
F_2 = *WEATHER*{*AREA* = Bordeaux *AND SUN* > 2,000
AND RAIN ≥ 1,000}[*YEAR, AREA, SUN*]
F_3 = *WEATHER*{*AREA* = Bordeaux *AND SUN* > 2,000
AND RAIN ≥ 1,000}[*YEAR, AREA, RAIN*]
F_4 = *WEATHER*{*AREA* = Bordeaux *AND SUN* > 2,000
AND RAIN < 1,000}[*YEAR, AREA*]
F_5 = *WEATHER*{*AREA* = Bordeaux *AND SUN* > 2,000
AND RAIN < 1,000}[*YEAR, AREA, SUN*]
F_6 = *WEATHER*{*AREA* = Bordeaux *AND SUN* > 2,000
AND RAIN < 1,000}[*YEAR, AREA, RAIN*]
F_7 = *WEATHER*{(*AREA* ≠ Bordeaux *OR SUN* ≤ 2,000)
AND RAIN < 1,000}[*YEAR, AREA*]
F_8 = *WEATHER*{(*AREA* ≠ Bordeaux *OR SUN* ≤ 2,000)
AND RAIN < 1,000}[*YEAR, AREA, SUN*]
F_9 = *WEATHER*{(*AREA* ≠ Bordeaux *OR SUN* ≤ 2,000)
AND RAIN < 1,000}[*YEAR, AREA, RAIN*].

Note that simply applying this method might generate fragments that contain only a few or even one tuple [16]. This is, of course, not desirable. Therefore this method should be handled with care and some clustering may be done by the database designer, for example, by only considering view definitions.

Before discussing whether further splitting is necessary as far as minimizing the cost of data allocations is concerned, we show the extension for queries that reference more than one relation. If a query contains a join, the fragments

required by the join can also be described by projections and selections. A selection consists of the clauses that are applicable to the corresponding relation and may also include clauses that are obtained by transitivity on clauses on the other relation and the joining clauses.

Assume a set of queries were used to split the relations into fragments and that these fragments are distributed over a computer network. To investigate whether further splitting is necessary, we discuss execution of queries by processing schedules that use the relational algebra operations *select*, *project*, and *join* in the way described below.

The result of a *select* is obtained by taking those fragments whose describing clause is contained in the Boolean expression of the select. What is done with these fragments depends on the next operation in the processing schedule. If there is none, all these fragments are sent to the result site.

The result of a *project* is obtained by taking those fragments whose attributes are part of the project. The result of a project is computed in several phases. First, all fragments with the same set of primary keys are collected at one site, and there local projects are computed (the same as a distributed project in [34]). Then the results are collected at one site where again a project is computed.

Which fragments are involved in a *join* can be determined in the same way as was done for a select and a project. There are many ways to compute a join. Here we will assume that the result of a join is computed as follows. For one relation all fragments with the same set of primary keys are collected at one site. So now the relation is only horizontally split and distributed over several sites. All the fragments of the other relation are sent to these sites and there the joins are computed [20]. What is done with the resulting fragments of the join again depends on the next operation. If there is none, all of them are sent to the result site.

Example 5. The result of the select $RAIN < 1{,}000$ is obtained by taking fragments F_4, F_5, F_6, F_7, F_8, and F_9. The result of the project [*YEAR, AREA, SUN, RAIN*] is obtained by taking the fragments F_2, F_3, F_5, F_6, F_8, and F_9.

Assume that all fragments are located at different sites in a computer network. The processing schedule of query Q_2 of Example 4 may then look like:

(1) consider F_4, F_5, F_6, F_7, F_8, and F_9 as input of the project,
(2) send F_6 to F_5 and F_9 to F_8 and execute local projects.
(3) send the results to the result site and do again a project.

Note that because the primary key is among the attributes in the project, the projects in steps 2 and 3 have no effect.

Now we will discuss whether further splitting the fragments obtained by applying the selection predicates of the queries is necessary. To do so, we need to introduce the notions *static under splitting* and *weighted split* and try to relate them for horizontal and vertical splitting. We call a distributed query processing algorithm *static under splitting* if a split of a fragment F into F' and F'' will only cause changes in a schedule concerning the incoming and outgoing edges of F in such a way that an edge coming from F is now replaced by an edge coming from

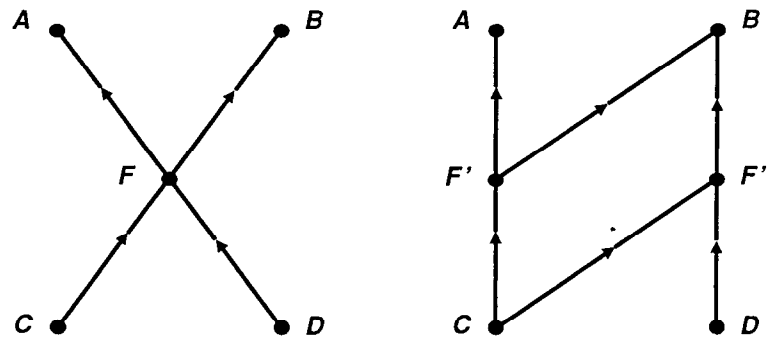


Fig. 5. Changes in schedule caused by static splitting algorithm.

F', or from F'', or from both. Furthermore, an edge going to F is now replaced by an edge going to F', or to F'', or to both.

Figure 5(a) shows part of a processing schedule involving the fragments A, B, C, D, and F; Figure 5(b) shows the changes in the schedule caused by splitting F into F' and F''. The changes reflect the changes by a query processing algorithm that is static under splitting.

A split of F into F' and F'' is called a *weighted split* if an outgoing edge of F labeled with (i, f, d) is replaced by two edges labeled $(i, f, d \mid F' \mid / \mid F \mid)$ and $(i, f, d \mid F'' \mid / \mid F \mid)$ leaving F' and F'', respectively. An incoming edge of F labeled (j, g, e) is replaced by two edges both labeled (j, g, e) going to F' and F''.

The value d stands for the estimated amount of data transmitted in transaction i from fragment F to another fragment. The estimation is done by a query processing algorithm. What the definition of weighted split says is that the data that come from F now come from both F' and F'' and the amount of data is proportional to the sizes of F' and F''. Now we will prove a theorem that relates the notions static under splitting and weighted split for a horizontal split. This is only possible when the split is done randomly. If the split is not random and information about the split is used in query processing, this information should be added to the set of queries on which the relations are split.

THEOREM 1. *For minimizing total transmission cost, a horizontal split of F into F' and F'' that is done randomly is a weighted split if the distributed query processing algorithm is static under splitting.*

PROOF. Assume that we split a fragment horizontally into F' and F''. An outgoing edge of F is the transmission of a result of an operation in which F participated. Because the split is done randomly a tuple of F' is equally likely part of the result as a tuple of F''. The result can be obtained by applying the operation to both F' and F'' and uniting the two results. Because of these two reasons the estimated size of the result produced by F' and F'' is $\mid F' \mid / \mid F \mid$ and $\mid F'' \mid / \mid F \mid$ times the estimated size of the result produced by F.

Because the split is done randomly no information is known about the tuples in F' and F'' and, therefore, the incoming edges are both labeled with the label of the original edge. Hence, the split is weighted. □

A similar result cannot be obtained for a vertical split. A vertical split is not necessarily weighted because the change in the schedule may cause all the incoming edges to go to F' and none to F''. An example of this is the schedule for a project. Assume that F and H are fragments obtained by a vertical split of relation R. The schedule of a project on R will first compute local projects on F and H and possibly send the result of the project on H to the site of F. If F is split vertically into F' and F'', the schedule will change drastically because F no longer exists. A new schedule will first compute local projects on F', F'', and H and then send results of F'' and H to F'. But then the split is not weighted.

Now we come to the main result of this section.

THEOREM 2. *Further splitting the fragments obtained by horizontally splitting the relations based on clauses of queries and vertically splitting them on (primary key, attribute)-pairs will not decrease the total transmission cost, if the schedules are static under splitting.*

PROOF. Because the cost function is the total transmission cost, the labels (i, f, d) are replaced by fd. The fragments obtained cannot be split vertically because they contain only one attribute besides the primary key, which is mandatory.

Let us consider a horizontal split of fragment F into F' and F''. Let us assume that F' and F'' are allocated to different physical sites in the optimal allocation. We will show that allocating F' and F'' to the same physical site will not increase the total transmission cost. Figure 6 shows part of the processing-schedules graph in the split form; r' stands for $|F'|/|F|$ and r'' for $|F''|/|F|$. All virtual sites have been united with physical sites except VS' and VS'', which contain F' and F'', respectively. t_0, t_1, and t_2 stand for the total amount of data weighted by the frequencies of their transactions from a fragment of PhS_0 to F, from F to a fragment of PhS_1, and from F to a fragment of PhS_2, respectively. In the complete processing-schedules graph there may be more incoming edges for VS' and VS'', but suppose that the one labeled with t_0 is the largest of them.

In this partial processing-schedules graph there are six possible assignments such that VS' and VS'' are assigned to different physical sites. In all of them either VS' or VS'' is assigned to PhS_1 or PhS_2. Without loss of generality, assume that VS' is assigned to PhS_1. This means that $r't_1 \geq r't_2$, which implies $r''t_1 \geq r''t_2$. Removing VS'' from the site to which it was assigned and uniting it with VS' changes the total transmission cost as follows:

$$\max(r''t_2, t_0) - t_0 - r''t_1,$$

which is less than or equal to zero. Hence, the two fragments F' and F'' have been brought together without increasing the total transmission cost. So that further splitting the fragments horizontally is not required either. □

In practice, it may happen that distributed query processing algorithms are not static under splitting, but we expect that changes in the processing schedules caused by splitting a fragment can be modified back to the processing schedule before the fragment was split, just as in the theorem, without increasing total transmission cost. Therefore, the fragments F_{ij} will be the objects to be allocated.

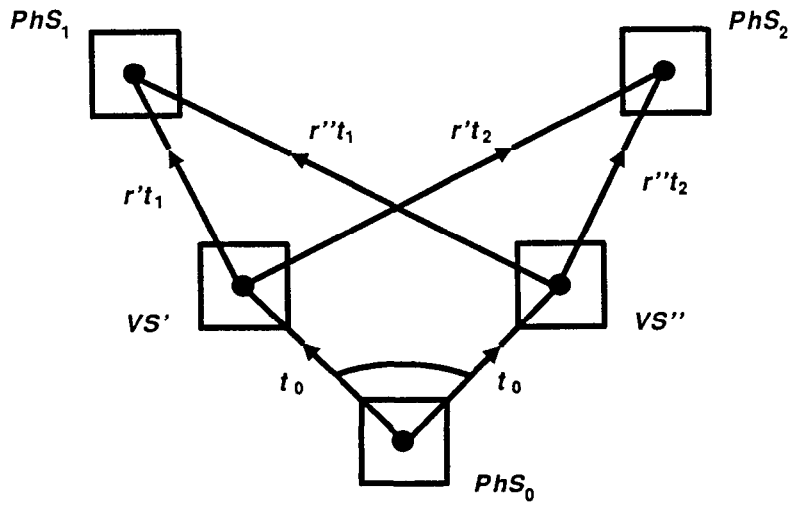


Fig. 6. Partial processing-schedules graph.

When minimizing response time the problem of determining the objects to be allocated is more complicated. A good heuristic to minimize the response time is to allow for as much parallelism as possible. At first glance, it may seem a good idea to just split R horizontally in sets containing an equal number of tuples. However, no information is known about the tuples in the obtained fragments, and, hence, every query or update must access all the fragments, causing less concurrency. Therefore the same approach is taken as for minimizing total transmission cost. The relations are split into fragments based on the queries stated by the users at the sites in the computer network. Further splitting the obtained fragments vertically will not enhance more parallelism. Further splitting it horizontally, on the other hand, is a good idea. Because the clauses of all queries have already been used, this further splitting will be done randomly. Owing to overhead it is better not to place every tuple at a different site. We assume that threshold values regarding the minimum and maximum number of tuples per fragment are given as system parameters. So the fragments obtained for minimizing total transmission cost are horizontally split further based on the threshold values.

The fragments constructed in the way described above are the objects to be allocated. If two fragments that contain the same primary key end up in the same fragment-set of a virtual or physical site, they together can be viewed as one large fragment with only one primary key.

Knowledge about a completely specified allocation is put in a global data dictionary. Such a dictionary may contain data about attributes that are contained in fragments, selections on the relations that define the fragments, number of tuples in the fragments, selectivity of certain attributes, allocation of copies, etc. Each of these items is of interest to different parts of the distributed database management system. Probably each will be accessed by different sites with a

different access pattern. Therefore, an *allocation of the global data dictionary* can be computed in exactly the same way as was done for the relations.

To summarize this subsection we may conclude that just looking at the logical components of a database is not enough to determine the objects to be allocated. Therefore, a way to split relations horizontally, based on the predicates of the queries, and vertically based on the attributes, was proposed.

5. OPTIMAL DATA ALLOCATION USING STATIC SCHEDULES

In this and the coming sections we assume that total transmission cost is to be minimized. First, we will introduce the notion of static processing schedules. Then, we will investigate the complexity of the data allocation problem using static processing schedules and discuss computing optimal allocations.

5.1 Complexity of Data Allocation Problem Using Static Schedules

In Section 4 a way of computing the cost of an allocation was given. The idea was to give the data allocation and the set of queries to a query processing algorithm, determine the processing schedules, put these in a processing-schedules graph, and compute the cost of the allocation based on this processing-schedules graph. The labels (i, f, d) for the edges will now be replaced by fd, because for computing the total transmission cost the identification of the transaction is not required and the execution frequencies can be multiplied by the individual data transmissions.

During any search for an optimal or efficient allocation the costs of many different allocations have to be compared. To avoid recomputing the schedules every time a different allocation is considered, we will use *static processing schedules*. We will explain what we mean by that. An *initial allocation* is an allocation where, for each query and update, a copy of a fragment is created and placed in its own virtual site, and none of the virtual sites are assigned to a physical site. For this initial allocation the processing schedules of the queries and updates are computed and put in a processing-schedules graph. Also, virtual sites containing different copies of the same fragments are interconnected by a forking graph, if the fragment is updated. The processing-schedules graph of an arbitrary allocation A is determined by applying unions and assignments to the initial allocation and adjusting the processing-schedules graph until A is reached. *Adjusting* a processing-schedules graph means that edges between two nucleus-sites that are united are removed. Given this processing-schedules graph we can compute the cost of allocation A, as described in Section 4, by adding up the labels of all edges except the ones between a physical site and its assigned virtual sites. Note that because we start from the initial allocation, where each query and update has its own copies of the fragments it accesses, we do not have to consider the nonredundant and redundant case separately.

Although the computation of the cost of allocations using static allocations is more efficient, finding a nonredundant, minimum total transmission cost allocation is still NP-complete [4, 19, 22].

THEOREM 3. *The problem of whether there exists a completely specified nonredundant allocation with total transmission cost less than or equal to a certain T using static processing schedules is NP-complete.*

PROOF. See Appendix. □

COROLLARY 1. *The problem whether there exists a nonredundant data allocation with total transmission cost less than or equal to T is NP-hard.*

5.2 Computing Optimal Allocations Using Static Schedules

In this section we will use techniques such as *branch-and-bound* [27] or the *Heuristic Path Algorithm* [33] to search the large solution space for determining data allocations to minimize total transmission cost. In [35] it was shown that these techniques are basically the same.

These search techniques construct decision trees. A node in such a tree is identified by the path from the root to that node. Each edge on this path corresponds to a decision taken about the data allocation; an example decision is to unite VS_i with PhS_j. During the search for an optimal data allocation the decision tree constructed so far partitions the space of completely specified allocations into subsets that belong to the leaves. We say that a completely specified allocation *satisfies* a partially specified allocation if it is possible to modify the partially specified allocation by uniting virtual sites with physical sites such that the result is the completely specified allocation. A *subset belonging to a leaf* of a decision tree contains all completely specified allocations that satisfy the partially specified allocations defined by the decisions taken to reach that leaf. The *cost of a subset* is defined as the minimum cost among all solutions in the subset. Ideally, this value is known for each subset; however, normally this is not the case, and then it should be estimated.

For a partially specified allocation we define a *cost-estimator* as the sum of two components: (1) the cost caused by the decisions taken to reach the partially specified allocation from the initial allocation, and (2) an estimate of the cost that will be caused by decisions that still have to be taken to reach a completely specified allocation with least cost that satisfies the partially specified allocation. Depending on the latter, the cost-estimator may underestimate or overestimate the cost of the solution. The cost caused by decisions will be discussed in more detail later on.

The search proceeds as follows. At each iteration a leaf with the smallest cost-estimator is expanded. Expanding a leaf means that for the corresponding partially specified allocation the following decisions are considered: unite one of the virtual sites with each of the physical sites. In the decision tree this is represented by creating new edges under that leaf for every decision; this renders the leaf into an internal node. For each of the leaves of the newly created edges the cost-estimator of the corresponding subset is computed. Then the algorithm goes through the next iteration, again expanding a leaf with the smallest cost-estimator, until a leaf whose corresponding subset contains only one completely specified allocation is expanded; this allocation is chosen as the result.

The cost-estimator of a subset underestimates the cost of the subset; therefore, the Heuristic Path Algorithm will eventually find the optimal completely specified allocation [33]. An estimator with this property is called *admissible*. Obviously, if the cost-estimator only contains the cost caused by decisions taken to reach the partially specified allocation from the initial allocation, the search

deteriorates into an exhaustive search. So the estimator is important: The closer its values are to the real cost the sooner the search terminates.

Before we introduce some notions that are needed to explain the algorithm that computes the cost-estimator of a partially specified allocation, we will take a look at the basic ideas behind it.

If each virtual site in a partially specified allocation is directly or indirectly connected with only one physical site, the cost-estimator could simply be determined based on the transmissions between physical sites. If this is not the case, then we would like to remove certain transmissions such that it becomes true. These transmissions will be searched for by considering paths between physical sites. A path between physical sites can, intuitively, be considered as a chain of nucleus-sites starting at one physical site and going via zero or more virtual sites to the other physical site. To underestimate the cost that will be caused by uniting these virtual sites with physical sites, the cost of the cheapest connection is taken.

The cost-estimator of a partially specified allocation, obtained by uniting virtual sites with physical sites, is computed as follows. A *path* from PhS_i to PhS_j is a sequence of nucleus-sites $NS_0, NS_1, \ldots, NS_m$, where NS_0 is PhS_i and NS_m is PhS_j, $NS_1, NS_2, \ldots, NS_{m-1}$ are virtual sites, and that for $i = 0, 1, \ldots, m-1$ there is at least one edge in the processing-schedules graph between NS_i and NS_{i+1} or that NS_i and NS_{i+1} are nodes in at least one forking graph. The *length of a path* is the number of virtual sites on that path plus 1. The *cost of a path of length greater than 1* is the minimum of the total cost of the edges or forking graphs between two successive nucleus-sites in the sequence defining the path. *Paths of length 1* form a special case. If the two physical sites on that path are merely connected by an edge, the cost of that path is the cost of the edge. If the two physical sites on the path are part of a forking graph, we have to consider all the paths of length 1 concerning that forking graph at once. If k nodes of the forking graph are physical sites, then the total cost of such paths is $k - 1$ times the cost of the forking graph.

Removing a path means the removal of all the edges between the successive nucleus-sites in the sequence defining the path and the complete removal of all the forking graphs in which successive nucleus-sites are part of the processing-schedules graph. The reason that all edges and forking graphs are removed is that we do not know which edge or forking graph will appear in the processing-schedules graph of the completely specified allocation. If all paths are removed each virtual site is connected directly or indirectly with only one physical site.

To compute the cost-estimator of the partially specified allocation, the algorithm *psa_static_cost* shown in Figure 7 is applied. It considers paths between physical sites and sums up their cost. To ensure that the edges and the forking graphs are not used in two different paths, they are removed. A forking graph is replaced by an edge between the notification node and one of the receiving nodes; the choice of receiving node is quite arbitrary. Therefore, it should be interpreted that a forking graph may be used only once in a path.

Example 6. To show how *psa_static_cost* computes the cost-estimator of a partially specified allocation, we will apply it to a simple allocation, shown in Figure 8.

```
proc psa_static_cost = (schedules graph psg)real:
begin
  real sum;
  sum := the sum of the cost of all paths of length 1;
  remove all paths of length 1;
  replace all forking graphs by one edge from their notification nodes to one of the receiving nodes;
  while there exists a path between two physical sites, say P
  do
    sum := sum + cost of path P;
    remove path P
  od;
  psa_static_cost := sum
end
```

Fig. 7. Algorithm *psa_static_cost*.

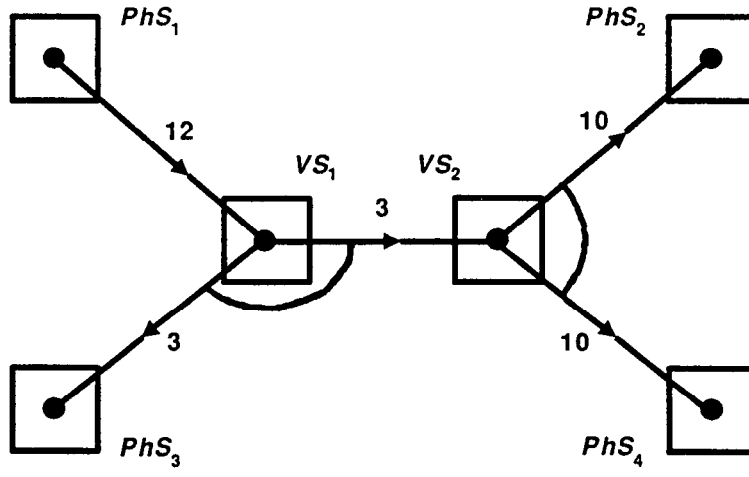


Fig. 8. Processing-schedules graph.

First, paths of length 1 are considered. There is only one, namely between PhS_2 and PhS_4. Because it is part of a forking graph, the whole forking graph is considered at once. Two physical sites are part of it and, therefore, the cost is $(2 - 1) \times 10 = 10$. Then the forking graph is completely removed from the processing-schedules graph.

After this, all remaining forking graphs are replaced by one edge from the notification node to one of the receiving nodes. Here, we assume that this edge connects VS_1 with PhS_3. The only path left is PhS_1, VS_1, PhS_3, with cost equal to 3. Hence, the cost-estimator is $10 + 3 = 13$. Note that if the edge between VS_1 and VS_2 were put in the processing-schedules graph to replace the forking graph, there would be no paths between physical sites, resulting in a value for the cost-estimator of 10. As a rule, *psa_static_cost* should try to replace forking graphs by edges between subgraphs containing physical sites.

The optimal completely specified allocation that satisfies the partially specified allocation is obtained by uniting VS_1 with PhS_1 and VS_2 with either PhS_2 or PhS_4; its cost is $3 + 3 + 10 = 16$.

Now we will show that the result of *psa_static_cost* is always less than or equal to the cost of all completely specified allocations satisfying the partially specified allocation.

THEOREM 4. *Algorithm psa_static_cost is an admissible estimator if static schedules are used.*

PROOF. Assume we are given a partially specified allocation *PSA* and its processing-schedules graph. First consider the cost of paths of length 1. Every completely specified allocation must satisfy *PSA* and, therefore, any path of length 1 represents an edge in the processing-schedules graph, so it will be part of the processing-schedules graphs of all completely specified allocations. Hence algorithm *psa_static_cost* correctly includes the cost of these paths.

The replacement of the forking graphs by one edge cannot increase the cost of the partially specified allocation.

Now paths of greater length are considered. Let us say $NS_0, NS_1, \ldots, NS_m$ is such a path between PhS_i and PhS_j, $m \geq 2$. In a completely specified allocation satisfying *PSA* there exists at least one pair (NS_i, NS_{i+1}) such that NS_i and NS_{i+1} are united with different physical sites. In that case the total cost of the edges between NS_i and NS_{i+1} is part of the cost of the completely specified allocation. The total cost of these edges can be underestimated by taking the minimum total cost of the edges on that path.

Hence algorithm *psa_static_cost* underestimates the cost of any completely specified allocation that satisfies a partially specified allocation. □

COROLLARY 2. *If the Heuristic Path Algorithm uses psa_static_cost, the completely specified allocations produced have minimum total transmission cost if static schedules are used.*

6. HEURISTIC DATA ALLOCATION USING STATIC SCHEDULES

A heuristic algorithm for determining data allocations when using static processing schedules to compute the cost of allocations will be presented. Both theoretical and experimental results will be discussed.

6.1 Algorithm *total_data_allocation*

A well-known heuristic technique to find an efficient solution is to start from an initial solution and to locally optimize this until no improvements are possible. When, during optimization, several improvements are possible, the one that decreases the cost function most is chosen. Algorithms that use this technique are called *greedy* [25].

The heuristic approach that we propose here is based on the following two ideas:

—Virtual sites cannot be united with physical sites independently of each other. Therefore, uniting virtual sites with each other before uniting them with physical sites is considered.

—The label of an edge in the processing-schedules graph gives a measure of how important it is that the adjacent nucleus-sites are united, when minimizing the total transmission cost. The adjacent nodes of the edges with the largest labels are therefore considered first for uniting.

Before introducing the algorithm we introduce some notions. The sum of the labels of the edges that disappear if two nucleus-sites NS_i and NS_j are united, or that one is assigned to the other, is called $LINK_{ij}(= LINK_{ji})$. Remember that although two virtual sites may be assigned to one physical site in a partially specified allocation, the edges between the virtual sites still count when computing the total transmission cost as long as they are not united.

The input of algorithm *total_data_allocation* is the processing-schedules graph of the initial allocation, meaning that every query and update has copies of fragments it accesses, placed in virtual sites of their own, and that none of these virtual sites are assigned to a physical site. This *total_data_allocation* determines a partially specified allocation by assigning every virtual site to the physical site for which $LINK_{ij}$ is maximum. Gradually it works towards a completely specified allocation by considering unions of virtual sites. This is done in decreasing order of their LINK-values. Uniting two virtual sites consists of two actions. First, the two virtual sites, VS_i and VS_j, are removed from the physical sites to which they are assigned. This will increase the total transmission cost with

$$\max_k LINK_{ik} + \max_k LINK_{jk}.$$

The second action is to unite them and to assign the virtual site that results from the union, VS_u, again to the physical site PhS_k for which $LINK_{uk}$ is maximum. This decreases the total transmission cost with

$$\max_k LINK_{uk} + LINK_{ij}.$$

The net result is the difference of these two amounts. The algorithm decides to unite the two virtual sites if the net result is nonpositive. Before VS_u can be assigned its LINK-values, other nucleus-sites have to first be determined.

At every iteration the algorithm takes the pair with the largest $LINK_{ij}$ that has not yet been considered since the last union. This continues until uniting any pair of virtual sites will increase the total transmission cost. The rationale behind the algorithm is to remove the heaviest transmissions first, if uniting the adjacent nodes of these transmissions is cost effective (i.e., decrease the total transmission cost).

In the resulting allocation no two virtual sites will be assigned to the same physical sites. Let us assume that VS_i and VS_j are both assigned to PhS_k. Then

$$LINK_{ik} + LINK_{jk} - (LINK_{uk} + LINK_{ij}) \leq 0,$$

where VS_u is the union of VS_i and VS_j, which contradicts the termination condition of the algorithm. Figure 9 shows the procedural form of algorithm *total_data_allocation*, which minimizes total transmission cost. Note that because *total_data_allocation* started from the initial allocation, it determines both how many copies of a fragment are required and their allocation.

We will show by an example how the algorithm works.

Example 7. Consider again the relations *WINE* and *WEATHER* of Figure 1 and the following two queries and two updates:

Q_1: (*WEATHER*(*YEAR* = *YEAR* AND *AREA* = *AREA*)*WINE*)
[*YEAR*, *AREA*, *NAME*, *PRODUCER*, *COUNTRY*, *SUN*, *RAIN*]

```
proc total_data_allocation = (schedules graph PSG)allocation:
begin
  set P;
  boolean goon := true;
  for i to n
  do assign VS_i to PhS_k with LINK_ik is maximum od;
  while goon
  do
    P := set of pairs of virtual sites that are not yet united;
    goon := false;
    while P ≠ { } and not goon
    do
      take (VS_i, VS_j) from P such that LINK_ij is maximum;
      if max_k LINK_ik + max_k LINK_jk − (LINK_ij + max_k LINK_uk) ≤ 0
        then
          VS_u := union of VS_i and VS_j;
          remove VS_i and VS_j from processing-schedules graph PSG;
          add VS_u and recompute its LINK-values;
          goon := true
      fi
    od
  od;
  unite virtual sites with their physical sites;
end
```

Fig. 9. Algorithm *total_data_allocation*.

Q_2: (*WEATHER*(*YEAR* = *YEAR*)(*WINE*{*AREA* = Napa Valley}))
[*YEAR*, *AREA*, *NAME*, *PRODUCER*, *COUNTRY*, *SUN*, *RAIN*]
U_1: add new wines to *WINE* from Napa Valley.
U_2: add information about the weather.

Because relation *WINE* is accessed in two queries it will be split according to the procedure of Subsection 4.4. Here it is split into two, W' and W'', where

W' = *WINE*{*AREA* = Napa Valley}
W'' = *WINE*{*AREA* ≠ Napa Valley}.

The relation *WEATHER* will not be split because both queries require all its tuples.

For query Q_1 the virtual sites VS_1, VS_2, and VS_3 are created, containing W', W'', and *WEATHER*, respectively. The processing schedule of Q_1 consists of the following data transmissions: The relation *WEATHER* is sent to the two fragments W' and W''. The results of the joins are sent to PhS_1. For query Q_2 the virtual sites VS_4 and VS_5 are created, containing *WEATHER* and W'', respectively. The schedule for Q_2 is: The relation *WEATHER* is sent to fragment W'' where the join is computed and the result is sent to PhS_2. For the updates U_1 and U_2 the virtual sites VS_6 and VS_7 are created, containing W'' and *WEATHER*, respectively. For each query and each update, copies of the fragments involved, allocated to virtual sites, are interconnected in forking graphs. The resulting processing-schedules graph is shown in Figure 10.

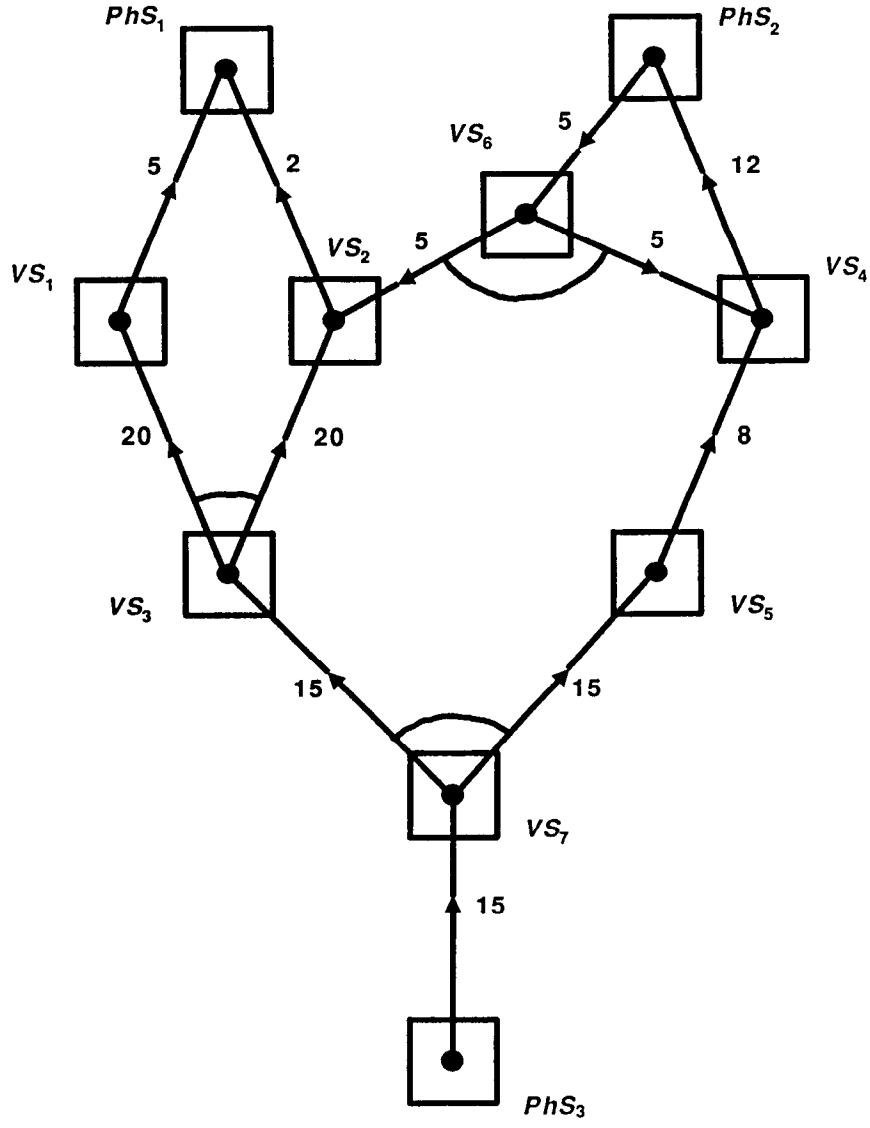


Fig. 10. Processing-schedules graph of Example 7.

Total_data_allocation starts with computing the initial assignment, characterized by the assignment of each virtual site to a physical site for which the sum of the amount transmitted to it plus the amount received from it is largest; VS_1 and VS_2 are assigned to PhS_1, VS_4, VS_5 (arbitrarily), and VS_6 to PhS_2, and VS_3 (arbitrarily) and VS_7 to PhS_3.

Note that the virtual sites that are not directly connected to a physical site are assigned to an arbitrary physical site. This situation is shown in Figure 11.

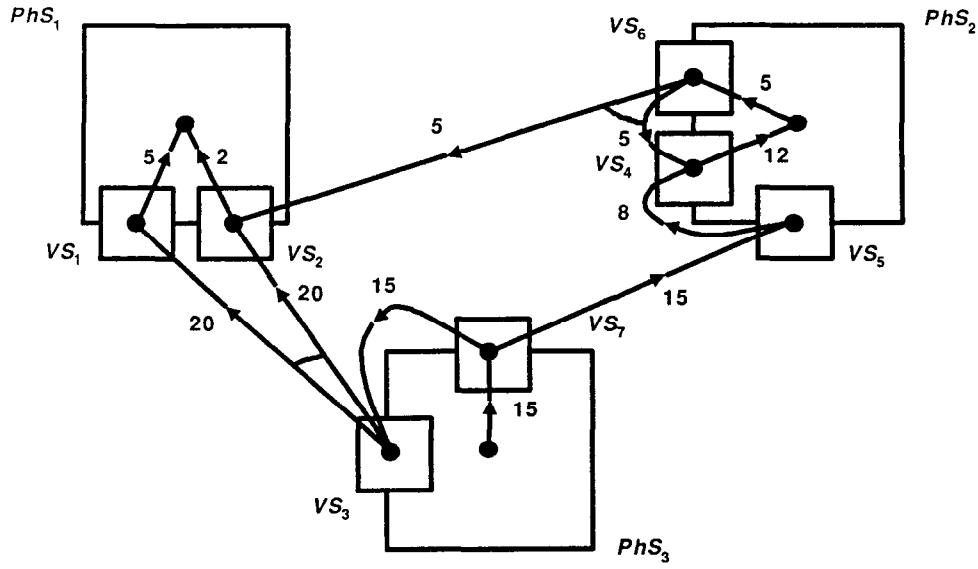


Fig. 11. Processing-schedules graph after initial assignment.

The set P contains all the pairs of the virtual sites that can be united. They are listed below with their LINK-values:

(VS_i, VS_j)	$LINK_{ij}$
(VS_1, VS_2)	20
(VS_1, VS_3)	20
(VS_2, VS_3)	20
(VS_2, VS_4)	5
(VS_2, VS_6)	5
(VS_3, VS_5)	15
(VS_3, VS_7)	15
(VS_4, VS_5)	8
(VS_4, VS_6)	5
(VS_5, VS_7)	15

Because the pair (VS_1, VS_3) has the largest LINK-value it is considered first (among the pairs with LINK-value equal to 20 this choice is arbitrary). To unite VS_1 and VS_3 they have to be first removed from their respective physical sites, PhS_1 and PhS_3. This increases the total transmission cost with:

$$5 + 0.$$

Uniting them and assigning the union, VS_{u13}, to PhS_1 decreases the total transmission cost with

$$20 + 0.$$

The net result, the difference between the two changes, is nonpositive and, therefore, they are united (see Figure 12).

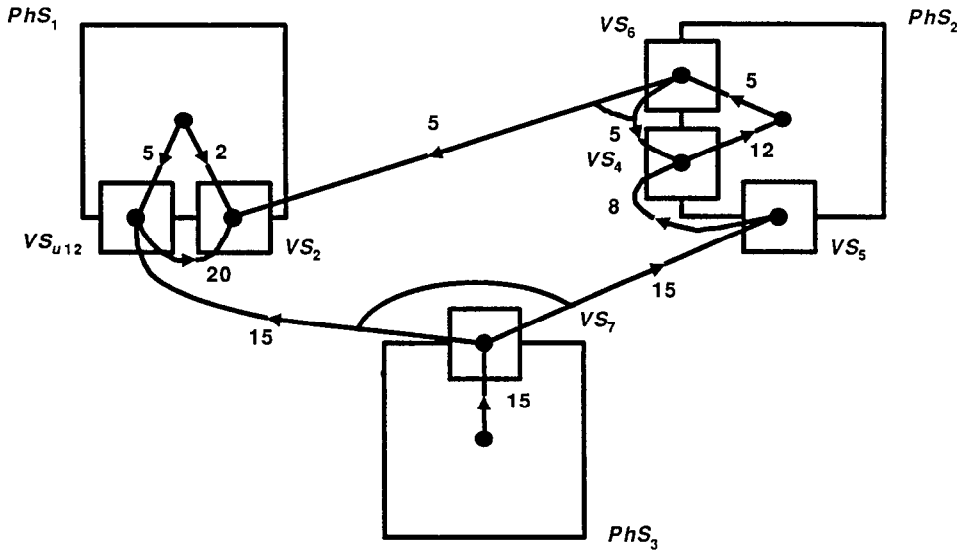


Fig. 12. Processing-schedules graph after the union of VS_1 and VS_3.

The next pair to be considered is (VS_{u13}, VS_2), whose LINK-value is 20. The net result of uniting them is

$$5 + 2 - (20 + 7) = -20 \leq 0.$$

Again, the union, VS_{u123}, decreases the total transmission cost.

The next pair to be considered is (VS_{u123}, VS_7), whose LINK-value is 15. The net result of uniting them is

$$7 + 15 - (15 + 15) = -8,$$

which is nonpositive; therefore, the two virtual sites, which contain copies of the same relation *WEATHER*, are united, resulting in VS_{u1237}. This means that only one copy will be maintained in the system.

Uniting VS_5 and VS_{u1237} decreases the total transmission cost with:

$$0 + 15 - (15 + 15) = -15;$$

the result of the union is VS_{u12357}.

The LINK-value between the two virtual sites, VS_4 and VS_{u12357}, is 8; uniting them increases the total transmission cost with:

$$12 + 15 - (8 + 15) = 4;$$

therefore, the allocation is not changed.

Finally, VS_4 and VS_6 are united because the total transmission cost decreases by 5; the result of the union is VS_{u46}.

Note that at most one virtual site is assigned to a physical site, thus uniting the virtual sites with their physical sites gives a completely specified allocation. The partially specified allocation obtained so far consists of the assignment of VS_{u12357} to PhS_3 and of VS_{u46} to PhS_2.

The final allocation shows that all fragments and relations involved in Q_1 are located at one site, and that only the result has to be transmitted to PhS_1. The data involved in query Q_2 are distributed over two sites: The relation *WEATHER* is located to PhS_3 and the fragment W'' is located to PhS_2. Because this fragment is updated infrequently two copies can be maintained, one at PhS_2 and one at PhS_3.

In the above example we see that at some points we have to choose which pair of virtual sites to unite if the LINK-values are the same. Taking a different choice may lead to different completely specified allocations. We will not discuss obvious improvements to deal with this because it would prohibit obtaining optimality results regarding the solutions, which will be discussed in the next section.

6.2 Optimality Results Concerning Algorithm *total_data_allocation*

As was mentioned before, the algorithm *total_data_allocation* is greedy and does not necessarily obtain a completely specified allocation with the absolute minimum total transmission cost. However, it is important to know how well the algorithm performs. We will do so by showing that for a special class of processing-schedules graphs, the algorithm computes minimum total transmission cost allocations and by discussing simulation results in the next section. But before doing so, we introduce some notions.

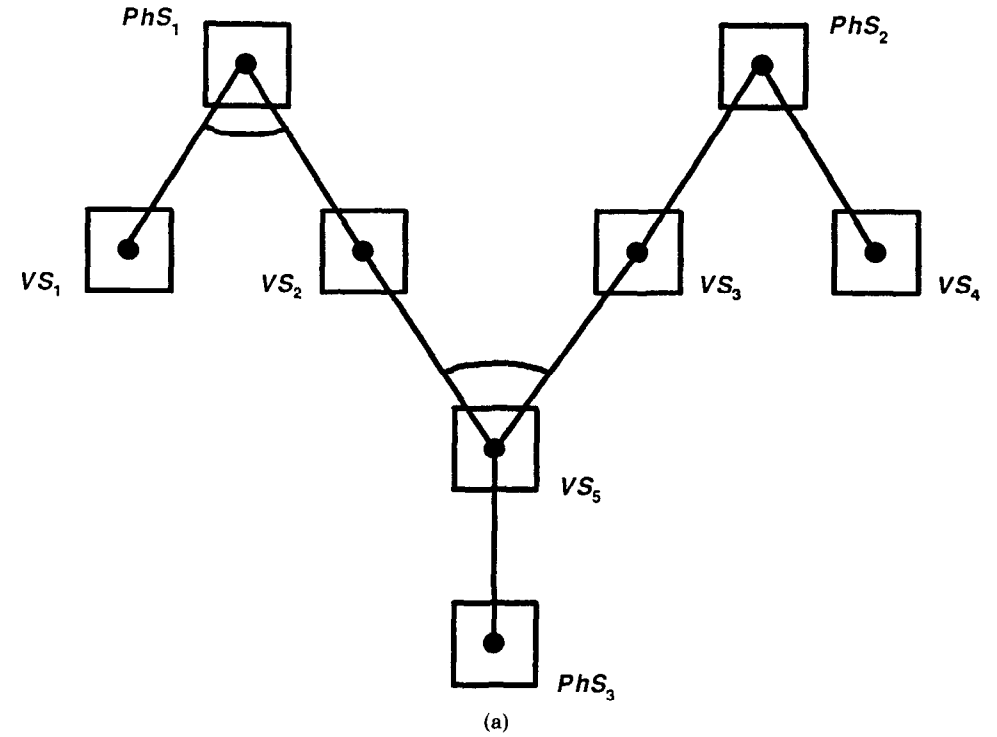
The set of virtual sites can be divided into *clusters*. Two virtual sites, VS_i and VS_j, belong to the same cluster if there is a path $VS_i = VS_0, VS_1, \ldots, VS_m = VS_j$ such that VS_k and VS_{k+1} are adjacent to each other. Two virtual sites are *adjacent* to each other in a processing-schedules graph if there is an edge between the two virtual sites, or if they occur in the same forking graph.

A cluster is called a *simple cluster* if for every pair of virtual sites, VS_i and VS_j, in the cluster the following holds: Removal of all the edges that are adjacent to both VS_i and VS_j and the removal of the forking graph of which both VS_i and VS_j are part causes VS_i and VS_j to no longer be in the same cluster.

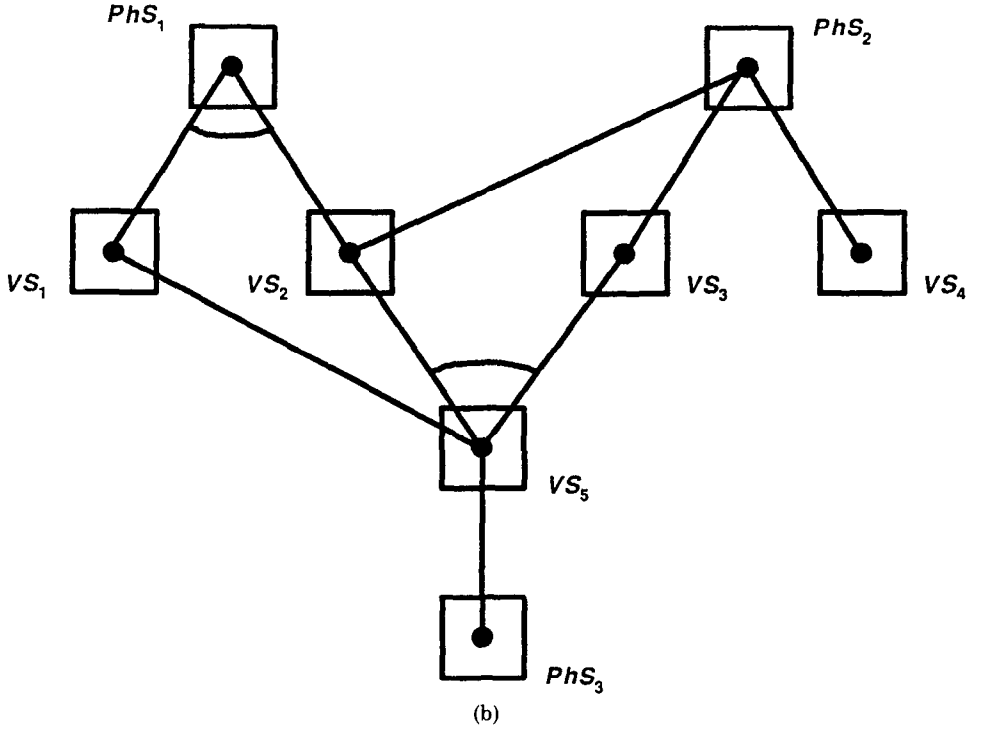
A *simple processing-schedules graph* is defined as a processing-schedules graph for which the clusters are simple and all physical sites are connected by edges with only one virtual site per cluster, or are part of only one forking graph per cluster.

Intuitively, in simple processing-schedules graphs the net change in the total transmission cost if two virtual sites are united is simply based on the transmissions between these two virtual sites and between them and the physical sites.

Example 8. Figure 13(a) shows a simple processing-schedules graph. There are two clusters, C_1 and C_2; C_1 consists of VS_1, VS_2, VS_3, and VS_5 and C_2 of VS_4. Note that VS_1 and VS_2 are connected through the forking graph of which they both are part. The processing-schedules graph in Figure 13(b) is nonsimple. Two edges have been added: between VS_1 and VS_5 and between PhS_2 and VS_2. The first one causes C_1 to no longer be simple. After removal of the forking graph of which VS_2 and VS_5 are part, there is still path between them via VS_1. The second one connects PhS_2 to two virtual sites of the same cluster C_1.



(a)



(b)

Fig. 13. (a) Simple processing-schedules graph; and (b) a nonsimple processing-schedules graph.

THEOREM 5. *The completely specified allocation obtained by algorithm total_data_allocation for simple processing-schedules graphs using static schedules minimizes total transmission cost.*

PROOF. Assume that a completely specified allocation obtained by algorithm *total_data_allocation* does not have minimum total transmission cost. We will show that we can change the optimal allocation into the allocation obtained by our algorithm without losing its optimality.

The optimal solution imposes a partition on the set of virtual sites; the subsets of this partition contain the virtual sites belonging to the different physical sites. Changing the optimal solution means changing the partition.

We will go through the steps of the algorithm. If the algorithm decides to unite two virtual sites that occur in the same subset then there is no problem. The processing-schedules graph will be changed such that the two virtual sites will form only one nucleus-site, and in the subset of the optimal partition they will be replaced by one new element with the same name as the corresponding VS-node.

Similarly, there will be no problem if the algorithm decides not to unite two virtual sites that occur in different subsets.

In the two remaining cases we have to change the optimal partition. Assume this is the first time that the algorithm either decides to unite two virtual sites that occur in different subsets or it decides not to unite two virtual sites that occur in the same subset, and that the involved virtual sites are VS_i and VS_j. This means that $LINK_{ij}$ is the largest of all pairs of virtual sites that are not united.

I. VS_i and VS_j do not occur in the same subset of the optimal partition, while the algorithm wants to unite them. Consider the following cases:

(1) Either VS_i or VS_j or both do not communicate with the physical site to which they are assigned. Without loss of generality, say VS_i. The physical site to which VS_i is assigned will be called PhS and its corresponding subset in the optimal partition, S. If none of the virtual sites of S communicates with PhS, all the virtual sites of S can be moved to the subset containing VS_j without increasing the total transmission cost.

Also, if there are virtual sites in S that send data to PhS, but occur in another cluster than VS_i, all other virtual sites of S that are in the cluster containing VS_i can be moved together with VS_i to the subset containing VS_j without increasing the transmission cost. Now, assume VS_k communicates with PhS and is in the same cluster as VS_i. Then there is a sequence $VS_k, \ldots, VS_t, VS_i$ (see definition cluster). Because the cluster is simple we can split it by removing all edges and forking graphs containing VS_t and VS_i. All virtual sites of S that are in the cluster of VS_i after the split are moved to the subset containing VS_j. This introduces $LINK_{ti}$ data transmissions, which is less than or equal to $LINK_{ij}$, the amount of data transmitted that disappears because VS_i and VS_j are now in one subset. In this subset VS_i and VS_j are replaced by a new element with the same name as the corresponding VS-node in the processing-schedules graph that results from uniting VS_i and VS_j.

(2) Both VS_i and VS_j communicate with the physical site to which they are assigned. Because physical sites are connected with only one virtual site per cluster, or are part of only one forking graph per cluster, it follows that $\max_k LINK_{uk}$ is either equal to $\max_k LINK_{ik}$ or $\max_k LINK_{jk}$, where VS_u is the union of VS_i and VS_j. Hence we only have the following two cases:

(a) $\max_k LINK_{ik} = \max_k LINK_{uk}$ and $\max_k LINK_{jk} \leq \max_k LINK_{uk}$. Assume VS_j occurs in the subset belonging to physical site PhS_l. Moving all the virtual sites of this subset to the subset of VS_i decreases the total transmission cost with:

$$\begin{aligned} LINK_{jl} - LINK_{ij} &\leq \max_k LINK_{jk} - LINK_{ij} \\ &= \max_k LINK_{ik} + \max_k LINK_{jk} \\ &\quad - \max_k LINK_{uk} - LINK_{ij} \leq 0 \end{aligned}$$

(b) $\max_k LINK_{ik} \leq \max_k LINK_{uk}$ and $\max_k LINK_{jk} = \max_k LINK_{uk}$. The same as under (a), only the elements of the subset of VS_i are moved to the subset of VS_j.

II. VS_i and VS_j occur in the same subset of the optimal partition, while the algorithm does not want to unite them.

Similarly, we can prove that separating VS_i and VS_j in the optimal solution will not lead to an allocation with higher total transmission cost.

Finally, by changing the optimal partition every time the algorithm wants it to, the optimal partition is the same as the solution obtained by the algorithm. We have thus seen that under the conditions stated, the optimal solution can be changed step by step into the solution of the algorithm. □

6.3 Comparison Between Optimal and Heuristic Allocations Using Static Schedules

Now that we have seen that *total_data_allocation* computes data allocations that minimize the total transmission cost for processing-schedules graphs that belong to a special class, we are interested in how it works in "practice." To get an idea, we compute the optimal allocation of randomly generated processing-schedules graphs and compare it with the cost of the allocations generated by *total_data_allocation*. We also compare the number of sites over which the data are distributed per transaction. This means that for a transaction the number of sites are counted that contain fragments that are used in the transaction, except copies of fragments that are updated. Note that if the result site does not contain any fragments used in the transaction, it is not counted.

The transactions are generated as follows. A processing schedule of a transaction will have one of the basic forms shown in Figure 14, with its probability that it is generated below it; c is the complexity parameter, which indicates the probability that a particular branch ($R_j \rightarrow R_i$, $R_k \rightarrow R_i$, or $R_l \rightarrow R_j$) is included in a processing schedule. To complete a processing schedule a branch from R_i to the result site is included. So, for small c the simplest schedules are generated with a higher probability than the more complex ones. For larger c it is the other way around.

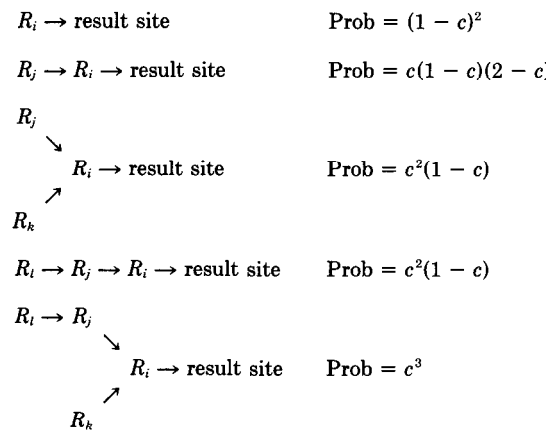


Fig. 14. Five different processing schedules with the probability that they are generated; c is the complexity parameter.

Because the relations are used by different transactions they are split into fragments; we assume that they are split into three. When generating the processing schedule for a transaction, for each relation it is decided which fragments are in fact used. Each of the three fragments of a relation is accessed in a transaction with probability *frag*, with a minimum of one fragment. While processing an update, the tuples that have to be changed are computed. We assume here that they are computed at the result site and that that site notifies all fragments of the changes. Below we will display the average total transmission cost (*TTC*) and the average number of sites over which the data for *one* transaction is distributed (*sites*), both for the optimal allocation and the allocation obtained by algorithm *total_data_allocation*.

The parameters that will vary are:

—The total number of transactions, queries, and updates is 4; the parameters q and u indicate the number of queries and updates, respectively, which vary from 0 to 4.

—The complexity parameter c, which varies from 0 to 1 with steps of 0.25.

—The fragmentation parameter *frag*, which varies from 0 to 1 with steps of 0.25.

When one of the parameters is varied, the others are kept fixed at the following values:

$$u = 2 \quad c = 0.5 \quad frag = 0.5.$$

Also, the number of queries q plus the number of updates u equals 4. The results are shown in Table I. The table is divided into three subtables, where the results are displayed for varying one of the parameters mentioned above.

To still be able to compute the optimal allocations, the processing schedules and the parameters were chosen rather small. For the processing-schedules graphs generated it took about five times longer to compute the optimal allocations compared to the heuristic ones. This may not seem too bad; however, further

Table I. Comparison Results of *Total Data Allocation* and Optimal Solution

	Optimal		Heuristic	
	TTC	*Sites*	*TTC*	*Sites*
q u				
4 0	0	1	0	1
3 1	282.6	1.075	291.6	1.05
2 2	1,093.5	1.225	1,133.5	1.25
1 3	1,336.6	1.25	1,380.2	1.225
0 4	1,708.8	1.275	1,763.6	1.325
c				
0	162.6	1.05	162.6	1.05
0.25	231.1	1.125	233.1	1.125
0.5	1,093.5	1.225	1,133.4	1.25
0.75	—	—	1,249.9	1.175
1	1,474.0	1.2	1,491.5	1.125
frag				
0	667.5	1.025	706.8	1.1
0.25	731.1	1.075	761.7	1.225
0.5	1,093.5	1.225	1,133.4	1.25
0.75	764.4	1.125	802.6	1.15
1	990.8	1.275	990.8	1.275
Overall	830.7	1.16	856.0	1.17

increasing the size of the processing-schedules graph will rapidly increase the time required to compute the optimal allocations.

Varying the number of update transactions, u, does not seem to influence the quality of the allocations obtained by *total_data_allocation*. For the whole range the total transmission costs are slightly more than 3 percent above the optimal values.

If c equals 0, the way the queries are processed is the same as in the file allocation problem. The corresponding processing-schedules graph belongs to the special class for which the algorithm can compute the optimal solution. For c equal to 0.75, the algorithm for computing the optimal solution ran out of memory.

For high values of *frag*, groups of virtual sites are tightly coupled, so it is easy for *total_data_allocation* to compute the optimal solution. For smaller values the structure of the processing-schedules graph becomes more important, increasing the chance that the processing-schedule graph falls outside the special class.

We may conclude that for the small processing-schedule graphs investigated, *total_data_allocation* computes allocations that have, on the average, a 3 percent higher total transmission cost than the optimal one. So the simulation results support the theoretical results obtained in the previous section. Also, the number of sites over which the data are distributed per transaction is just a bit more than in the optimal solution.

7. DATA ALLOCATION USING DYNAMIC SCHEDULES

The cost of an allocation computed using static schedules will, in general, be higher than the cost of an allocation as defined in Section 4. The latter requires recomputation of schedules; this will be called computing the cost using *dynamic*

processing schedules. A consequence is that the allocations obtained using dynamic schedules will have a lower cost than the ones obtained using static schedules. In this section we will, therefore, investigate ways of determining optimal and heuristic allocations.

7.1 Optimal Data Allocations Using Dynamic Schedules

Having considered static schedules, we now examine dynamic ones. The one advantage of using dynamic schedules is that the processing-schedules graph belonging to a completely specified allocation contains schedules that are identical to the schedules produced by the distributed query processing algorithm, given the completely specified allocation.

The other advantage is that, in general, given a completely specified allocation, the processing schedules produced by a distributed query processing algorithm have a lower cost than the ones obtained from the processing-schedules graph belonging to the initial allocation using static schedules.

The main disadvantage is the computational effort required, compared to the usage of static schedules. Subsection 5.2 shows an estimator, which can easily be computed, for static schedules. This is not necessarily the case for dynamic schedules.

For example, assume that in the decision tree of the Heuristic Path Algorithm decisions have been taken to unite VS_i with PhS_v and VS_j with PhS_w, and that about two other virtual sites, VS_k and VS_l, that are all accessed in one query, no decision has been taken so far. Without knowing anything about the final allocation of the fragments, the processing schedule of the query and its cost cannot be computed. To obtain an underestimate of its cost all possible allocations have to be considered, and the one with the least cost could be used as heuristic estimator.

So, in general, the computation of a cost-estimator of a partially specified allocation cannot be done in polynomial time. However, under the realistic assumption that each query only accesses a relatively small number of fragments, an estimator can be constructed that runs efficiently. This can be achieved by doing some initial processing. The estimator will be called *psa_dynamic_cost*. A *one-query-allocation* is a partially specified allocation of all fragments accessed in one query. A one-query-allocation *satisfies* a partially specified allocation if the fragments in the fragment-sets of the nucleus-sites in the one-query-allocation occur together in the same fragment-sets in the partially specified allocation. Before the search starts, all these one-query-allocations are given to the query processing algorithm used by the distributed database system to compute the corresponding schedules and their cost. Updates are treated exactly the same as queries. The cost of their schedules does not include transmissions to keep copies consistent. During the search, a lower bound on the cost of a partially specified allocation given by a path in the decision tree is computed as follows. For each query, we consider all the one-query-allocations that satisfy the partially specified allocation and take the one with the least cost. The sum of all these costs, plus the cost to keep copies consistent if more than one copy of a fragment is allocated, is the cost-estimator of the partially specified allocation.

The procedural form of *psa_dynamic_cost* is shown in Figure 15. An example is given to show how it works.

```
proc psa_dynamic_cost = (allocation psa)real:
begin
  real sum := 0;
  for each query Q
  do
    take the one-query-allocation of Q with the least cost that satisfies psa;
    sum := sum + cost of this one-query-allocation
  od;
  psa_dynamic_cost := sum
end
```

Fig. 15. Algorithm *psa_dynamic_cost*.

Example 9. Let us assume that we are given a query stated by a user at the site corresponding to PhS_1. This query computes the join between the two relations *WINE* and *WEATHER*. There are five one-query-allocations (only the fragment sets are shown, because the operation sets are empty):

(1) $PhS_1 = (\{\})$
    $VS_1 = (\{WINE\})$
    $VS_2 = (\{WEATHER\})$,
(2) $PhS_1 = (\{\})$
    $VS_1 = (\{WINE, WEATHER\})$,
(3) $PhS_1 = (\{WINE\})$
    $VS_1 = (\{WEATHER\})$,
(4) $PhS_1 = (\{WEATHER\})$
    $VS_1 = (\{WINE\})$,
(5) $PhS_1 = (\{WINE, WEATHER\})$.

For each of these one-query-allocations, a processing schedule for the query and its cost can be computed.

The cost of a partially specified allocation is underestimated by *psa_dynamic_cost* as follows. Assume that a decision has already been taken to allocate fragment *WINE* to PhS_2, and that no decision has been taken yet about *WEATHER*. The one-query-allocations that satisfy this partially specified allocation are 1, 2, and 4. The one with the least cost is taken.

THEOREM 6. *The heuristic estimator psa_dynamic_cost is admissible.*

PROOF. The cost of one query is underestimated because all possible one-query-allocations are investigated. Also, the cost to keep copies consistent is underestimated because only transmissions between copies of fragments that are already allocated to physical sites are counted. □

COROLLARY 3. *If the Heuristic Path Algorithm uses psa_dynamic_cost, then the completely specified allocations obtained have minimum total transmission cost.*

7.2 Heuristic Allocations Using Dynamic Schedules

Incorporation of dynamic schedules in the heuristic algorithm *total_data_allocation* can be done in different ways. Remember that, in the algorithm, when

using static schedules the changes in the processing-schedules graph when two virtual sites were united were rather simple. The union of the virtual sites inherited all the incoming and outgoing edges of the virtual sites, and only the edges between them disappeared.

A simple way of dealing with such changes using dynamic schedules is to recompute the schedules of all transactions that might be affected by the change in the allocation. This means that the decision to change the allocation is taken based on the cost of schedules corresponding to the current allocation; only after the change, the schedules corresponding to the new allocation are computed.

This approach deals with the disadvantage of static schedules, that is, that the schedules in the final allocation might differ from the ones obtained from the query processing algorithm given this final allocation. However, there is one problem: The total transmission cost of an allocation by algorithm *total_data_allocation* using dynamic schedules is not necessarily less than when using static schedules. The reason is that virtual sites are united based on transmissions that also depend on the rest of the allocation. A change in the allocation of other virtual sites might completely change processing schedules, making a previously taken decision to unite two virtual sites obsolete. Therefore a different approach is taken.

A processing-schedules graph is no longer the basis to decide about changes in the allocation; instead, a *LINK-graph* is used. The structure of such a graph is the same as a processing-schedules graph; it contains PhS- and VS-nodes and edges. The difference can be found in the edges and their labels. Between every pair of nodes there is an edge, and its label is the change in the cost function if the two adjacent nodes are united or if one is assigned to the other. To compute a label of an edge between two nucleus-sites, the query processing algorithm is applied twice: once when the two nucleus-sites are united and once when they are not. The difference between the two costs is the label.

The way a completely specified allocation is computed is basically the same as by algorithm *total_data_allocation*. First the virtual sites are individually assigned to physical sites such that the total transmission cost is minimized. Then pairs of virtual sites are considered for uniting in descending order of the labels of the edges between them. The cost of removing the two virtual sites from the physical sites to which they are assigned is the sum of the labels of the edges between the two virtual sites and the virtual sites that have already been assigned to the physical sites involved. Uniting them will decrease the cost function by an amount denoted by the label of the edge between the virtual sites. However, the decrease in the cost function when the union is assigned to a physical site is not yet known. Therefore, the schedules of the queries involved have to be recomputed and an assignment of the union to each physical site must be considered.

If the difference between the increase and decrease of the change is nonpositive the two virtual sites are united. Taking the union of virtual sites is continued until no further improvement of the total transmission cost is possible. Finally, the remaining virtual sites are united with the physical sites to which they are assigned.

The advantage of the dynamic versus the static approach is that in the process of changing an allocation towards the final allocation, the processing schedules

of the transactions are adjusted to the new allocation and are therefore more efficient. Because the interaction between the data allocation algorithm and the query optimizer is very tight, the allocations obtained will be more efficient than in the static approach. The main disadvantage is that recomputing all or part of the processing schedules of the transactions may be prohibitively expensive.

8. ALLOCATION MANAGEMENT PROBLEM

In the previous sections a model and algorithms were introduced to determine allocations. In this section we will provide a framework in which this model and these algorithms can be used as tools by either one or a group of database administrators.

8.1 Centralized Data Allocation

We speak of a *centralized data allocation* if the allocation of all the data is considered at the same time and if either one database administrator or one central database management system is allowed to change the existing allocation.

All queries and updates will be used to determine the fragments, as discussed in Subsection 4.2, and to compute a completely specified allocation that minimizes a particular cost function. Algorithms presented in Sections 5–7 can be used to do so for minimizing total transmission cost. The allocation obtained will then be implemented by the database administrator, who can dictate an allocation to the local database management systems.

One may object that all queries and updates have to be known in advance to compute the fragments and to compute their allocation. In case they are not known, we may determine the fragments based on the global external views of the users, which can be considered as queries themselves. The flow of data between the fragments can no longer be computed with a query processing algorithm and should be estimated with statistical information based on an existing allocation. Changes in this flow owing to changes in the allocation should be estimated, based on the queries and updates that are known.

Another problem is caused by changes in the access patterns of the users. This would require a complete recomputation of an allocation based on the new queries and updates and the already existing allocation. In general, determining and actually implementing a new allocation is rather expensive; the former because the allocation of all fragments have to be reconsidered again, and the latter because of interaction between fragments many more of them may have to be reallocated than accessed by the new queries.

8.2 Decentralized Data Allocation

Quite a different approach, called the *decentralized data allocation*, assumes that the data is owned by different database administrators or that the distributed database is a collection of databases owned by different parties. Both cases have in common that there does not exist a central organization that can dictate the allocation of the data. Therefore, the database management systems of the sites should, in cooperation with each other, try to determine an optimal allocation of the data required by the users of their own sites.

This approach assumes that an allocation already exists and cannot be changed; for example, the already existing databases, which together form the distributed database, will already have an allocation. This distributed database will either be accessed by users of the already existing database or by other users in the computer network.

Users at a site who share the same view of the distributed database can request their local database management system to change the allocation such that a certain cost function is minimized by introducing copies of the fragments of the relations in which the group is interested. The goal is now to compute an allocation of these copies. If a copy is allocated to a different site than its original, it is a real copy; otherwise, it might vanish, depending on whether the group of users wants to access periodically updated copies. Introducing and allocating copies can be done for users having a different view or working at a different site. Therefore, these copies will be called *private copies*.

One advantage of the decentralized approach is the natural partition of the general data allocation problem into a number of smaller problems, which probably can be solved more easily.

Another advantage is that the data allocation can change more or less continuously through time. If a group of users starts using the database or changes its access pattern, their database administrator simply determines a new allocation for them without changing other users' allocations.

Because a group may access only its own private copies, it may be possible to *periodically update* these copies, depending on how up to date these copies have to be. In reality, quite often a user is not interested in the latest version of the database, especially when this is very costly. Many times, a user is happy with a consistent version of the database that may be a couple of hours or days out of date. The group of users may themselves decide how up to date their copies should be, and thereby decide the update cost [2, 3]. Decreasing this cost will make it more likely that an allocation is chosen such that processing retrievals becomes cheaper. In the centralized data allocation these periodically updated copies are not possible because many users will make use of the same copies. Therefore, these copies have to be kept up to date at all cost.

One disadvantage of the decentralized approach is that the overall cost of the allocation might be higher compared to the centralized approach. The reason is that in the centralized case the whole processing-schedules graph is considered, and in the decentralized case a collection of smaller processing-schedules graphs.

SUMMARY

A model has been introduced to compute the cost of a completely or partially specified allocation for various cost functions. The model is suitable to be used in both branch-and-bound and heuristic algorithms. For minimizing total transmission cost, we have shown that the problem of determining a nonredundant allocation is NP-hard. A method for determining the unit of allocation by means of splitting a relation in the conceptual schema based on the queries and updates is presented. Under restrictive conditions regarding the behavior of query processing algorithms under splitting fragments, we have shown that the fragments obtained by this method can be regarded as the unit of allocation. Both optimal

and heuristic algorithms for minimizing total transmission cost using static and dynamic processing schedules for computing the cost of an allocation have been investigated and compared. Finally, a framework was discussed for managing allocations in a distributed database consisting of one database or consisting of a collection of already existing databases, each having their own database administrator.

APPENDIX: Proof of Theorem 3

The problem is NP, because for a "guess" allocation we can, in polynomial time, determine whether its total transmission cost is less than or equal to T.

To show the NP-completeness of this problem we transform three-dimensional matching [22], a known NP-complete problem, to it.

Three-dimensional matching. Set $M \subset W \times X \times Y$, where W, X, and Y are disjoint sets having the same finite number q of elements. Does M contain a matching, i.e., a subset $M' \subset M$ such that $|M'| = q$ and no two elements of M' agree in any coordinate?

The construction of a processing-schedules graph from a three-dimensional matching problem is done as follows:

—The elements of the sets W, X, and Y are the virtual sites.

—For every triple $(w, x, y) \in M$, create a physical site and connect the virtual sites corresponding to w, x, and y to this physical site; label these edges with the number $d = 2|M| + 1$.

—Create an edge between the virtual sites corresponding to w and x if there exists a triple $(w, x, y) \in M$. Do the same for the pairs (x, y); label all these edges with the number 1. Count the number of edges with label 1, say this is equal to l ($l \leq 2|M|$).

The question of whether there exists a matching M' is transformed to: Does there exist an allocation with total transmission cost less than or equal to

$$3(|M| - q)d + (l - 2q).$$

Now we have to prove that this is a polynomial transformation. Obviously, it is polynomial. The rest of the proof is an outline.

Assume there exists an allocation with total transmission cost less than or equal to $3(|M| - q)d + (l - 2q)$. We can show that in this allocation each virtual site is united with a physical site with which it is connected by an edge in the processing-schedules graph. Hence, the number of virtual sites united with the same physical site is less than or equal to three. Also, a physical site will not be united with less than three virtual sites.

So either there are no virtual sites united with a physical site or there are exactly three. It can also be shown that the three virtual sites united with the same physical site are interconnected by two edges. Thus the three virtual sites, united with the same physical site, correspond to a triple in a matching M'. (All virtual sites are used exactly once.)

Assume we have a matching $M' \subset M$ and assume that there is no allocation with a total transmission cost less than $3(|M| - q)d + (l - 2q)$. Construct the

corresponding processing-schedules graph as was done above and unite the virtual sites corresponding with the triples of M' to the site to which they triple-wise are connected. The total transmission cost of this allocation is computed as follows. In this allocation there are $3(|M| - q)$ edges with label d; $2q$ edges with label 1 disappear because three virtual sites corresponding to a triple of M' are united with the same site. Thus the total transmission cost is

$$3(|M| - q)d + (l - 2q). \qquad \square$$

ACKNOWLEDGMENTS

I would like to thank Reind van de Riet, who not only made this research possible but who also put a lot of effort in reading the many drafts of my thesis. I am much indebted to him. To Patricia G. Selinger I owe many thanks for her comments on my research and its presentation. Furthermore, I would like to thank the referees for their comments, which improved the quality of this paper.

REFERENCES

1. ADIBA, M., CHUPIN, J. C., DEMOLOMBE, R., GARDARIN, G., AND BIHAN, J. L. Issues in distributed data base management systems: A technical overview. In *Proceedings of the 4th International Conference on Very Large Data Bases* (West Berlin, Sept. 1978), pp. 89–110.
2. ADIBA, M. E., AND LINDSAY, B. G. Database snapshots. In *Proceedings of Conference on Very Large Data Bases* (Montreal, Oct. 1980). pp. 86–91.
3. ADIBA, M. E. Derived relations: A unified mechanism for views, snapshots and distributed data. In *Proceedings of 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981). pp. 293–305.
4. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
5. APERS, P. M. G. Data allocation and distributed query processing. In *Proceedings of ACM Pacific '80* (San Francisco, Nov. 1980). ACM, New York, 1980, pp. 48–54.
6. APERS, P. M. G. Redundant allocation of relations in a communication network. In *Proceedings of 5th Berkeley Workshop on Distributed Data Management and Computer Networks* (San Francisco, Feb. 1981). pp. 245–258.
7. APERS, P. M. G. Centralized or decentralized data allocation. In *Proceedings of 2nd Seminar on Distributed Data Sharing Systems* (Amsterdam, June 1981). R. P. van de Riet and W. Litwin Eds. North-Holland, Amsterdam, 1981, pp. 101–116.
8. APERS, P. M. G. Query processing and data allocation in distributed database systems. Ph.D. dissertation, Computer Science Dept., Vrije Univ., Amsterdam, Sept. 1982.
9. APERS, P. M. G., HEVNER, A. R., AND YAO, S. B. Optimization algorithms for distributed queries. *IEEE Trans. Softw. Eng. SE-9*, 1 (Jan. 1983), 57–68.
10. BALDISSERA, C., BRACCHI, G., AND CERI, S. A query processing strategy for distributed data bases. In *Proceedings of the European Conference on Applied Information Technology, EURO-IFIP 1979* (London, Sept. 1979). North-Holland, Amsterdam, 1979, pp. 667–677.
11. BERNSTEIN, P. A., GOODMAN, N., WONG, E., REEVE, C. L., AND ROTHNIE, J. B. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 602–625.
12. CASEY, R. G. Allocation of copies of files in an information network. In *Proceedings of AFIPS 1972 SJCC*, vol. 40. AFIPS Press, 1972, pp. 617–625.
13. CERI, S., MARTELLA, G., AND PELAGATTI, G. Optimal file allocation for a distributed database on a network of minicomputers. In *Proceedings of International Conference on Databases* (Aberdeen, July 1980). Deen and Hammerlsey, Eds., Hayden, 1980.
14. CERI, S., NAVATHE, S., AND WIEDERHOLD, G. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng. SE-9*, 4 (July 1983), 487–563.

15. CERI, S., NEGRI, M., AND PELAGATTI, G. Horizontal data partitioning in database design. In *Proceedings of ACM-SIGMOD Conference* (Orlando, Fla., June 1982). ACM, New York, 1982.
16. CERI, S., AND PELAGATTI, G. *Distributed Database—Principles and Systems*. McGraw-Hill, New York, 1984.
17. CHU, W. W. Optimal file allocation in a multiple-computer information system. *IEEE Trans. Comput. C-18* (1969), 885–889.
18. CHU, W. W. Optimal file allocation in a computer network. In *Computer-Communication Networks*. N. Abramson and F. F. Kuo, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1973, pp. 83–94.
19. COOK, S. A. The complexity of theorem-proving procedures. In *Proceedings of 3rd Annual ACM Symposium on Theory of Computing*. ACM, New York, 1971, pp. 151–158.
20. EPSTEIN, R., STONEBRAKER, M. R., AND WONG, E. Distributed query processing in a relational data base system. In *Proceedings of ACM-SIGMOD* (Boston, May 1979). ACM, New York, 1979, pp. 169–180.
21. ESWARAN, K. P. Placement of records in a file and file allocation in a computer network. In *Proceedings of the IFIP Congress on Information Processing 1974*. North-Holland, Amsterdam, 1974, pp. 304–307.
22. GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
23. HEVNER, A. R., AND YAO, S. B. Query processing in distributed database systems. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 177–187.
24. HEVNER, A. R. Data allocation and retrieval in distributed systems. In *Advances in Data Management, vol. II*. Wiley, New York, 1983.
25. HOROWITZ, E., AND SAHNI, S. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, Md., 1978.
26. IN-SUP PAIK, AND DELOBEL, C. A strategy for optimizing the distributed query processing. In *Proceedings of First International Conference on Distributed Computing Systems* (Huntsville, Ala., Oct. 1979). IEEE, New York, 1979, pp. 686–698.
27. Lawler, E. L., and Wood, D. E. Branch-and-bound methods: A survey. *Oper. Res. 14*, 4 (July 1966), 699–719.
28. LEVIN, K. D. Organizing distributed data bases in computer networks. Ph.D. dissertation, The Wharton School, Univ. of Pennsylvania, Philadelphia, Sept. 1974.
29. LEVIN, K. D., AND MORGAN, H. L. Optimizing distributed databases—A framework for research. In *Proceedings of 1975 AFIPS NCC, vol. 44*. AFIPS Press, 1975, pp. 473–478.
30. MAHMOUD, S., AND RIORDON, J. S. Optimal allocation of resources in distributed information networks. *ACM Trans. Database Syst. 1*, 1 (March 1976), 66–78.
31. NAVATHE, S., CERI, S., WIEDERHOLD, G., AND DOU, J. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 680–710.
32. NGUYEN, GIA TOAN. Decentralized dynamic query decomposition for distributed database systems. In *Proceedings of ACM Pacific '80* (San Francisco, Nov. 1980). ACM, New York, 1980, pp. 55–60.
33. NILSSON, N. J. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971.
34. PELAGATTI, G., AND SCHREIBER, F. A. A model of an access strategy in a distributed database. In *Proceedings of the Working Conference on Data Base Architecture, IFIP-TC2, Data Base Architecture* (Venice, June 1979). Bracchi and Nijssen, Eds., North-Holland, Amsterdam, 1979.
35. POHL, I. Is heuristic search really branch-and-bound? In *Proceedings of 6th Princeton IEEE Symposium on Information Sciences and Systems* (March 1972), pp. 370–373.
36. RAMAMOORTHY, C. V., AND WAH, B. W. The placement of relations on a distributed relational database. In *Proceedings of the 1st International Conference on Distributed Computing Systems* (Huntsville, Ala., Oct. 1979). IEEE, New York, 1979, pp. 642–650.
37. ROTHNIE, J. B., AND GOODMAN, N. A survey of research and development in distributed database management. In *Proceedings of 3rd International Conference on Very Large Data Bases* (Tokyo, Oct. 1977). pp. 48–62.
38. SACCA, D., AND WIEDERHOLD, G. Database partitioning in a cluster of processors. *ACM Trans. Database Syst. 10*, 1 (March 1985), 29–56.

39. SELINGER, P. G., AND ADIBA, M. E. Access path selections in distributed data base management systems. In *Proceedings of International Conference on Databases* (Aberdeen, July 1980), pp. 204–215.
40. TOTH, K. C., MAHMOUD, S. A., AND RIORDON, J. S. Query processing strategies in a distributed database architecture. In *Proceedings of 2nd Seminar on Distributed Data Sharing Systems* (Amsterdam, June 1981). R. P. van de Riet and W. Litwin, Eds. North-Holland, Amsterdam, 1981.
41. WAH, B. W., AND LIEN, Y.-N. Design of distributed databases on local computer systems with a multiaccess network. *IEEE Trans. Softw. Eng. SE-11*, 7 (July 1985), 606–619.
42. WONG, E. Retrieving dispersed data from SDD-1: A system for distributed data bases. In *Proceedings of 2nd Berkeley Workshop on Distributed Data Management and Computer Networks* (San Francisco, May 1977), pp. 217–235.
43. YU, C. T., AND CHANG, C. C. Distributed query processing. *ACM Comput. Surv. 16*, 4 (Dec. 1984), 399–433.
44. YU, C. T., LAM, K., CHANG, C. C., AND CHANG, S. K. A promising approach to distributed query processing. In *Proceedings of Berkeley Conference on Distributed Data Bases* (San Francisco, Feb. 1982), pp. 363–390.

Received April 1983; revised April 1984, September 1987; accepted October 1987