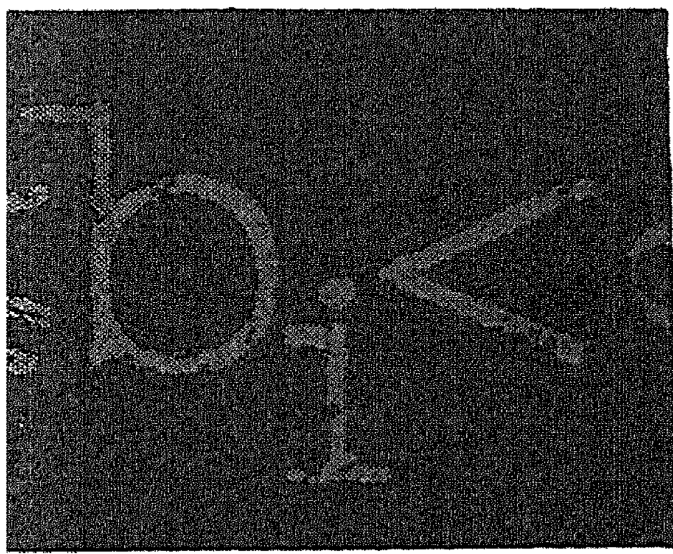
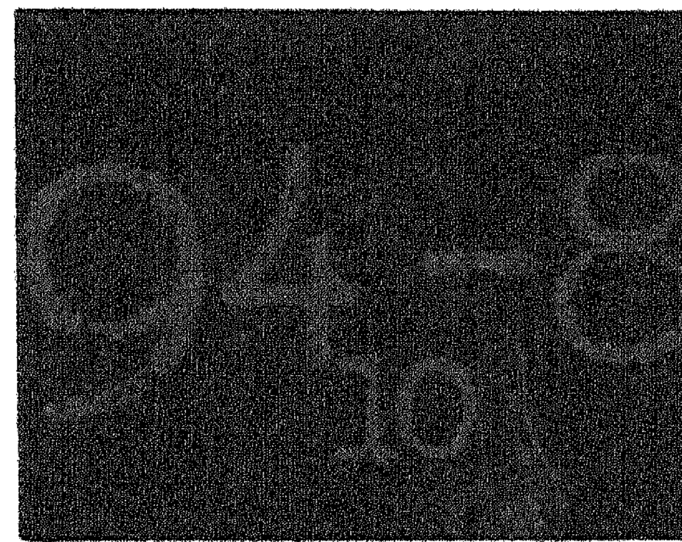
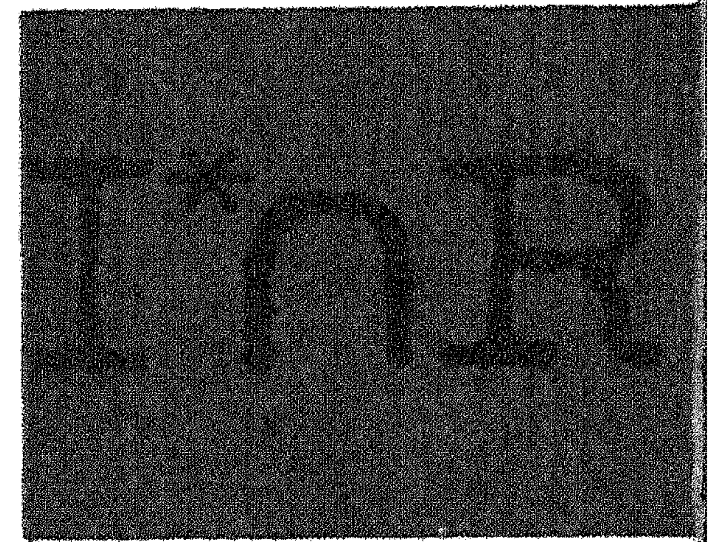
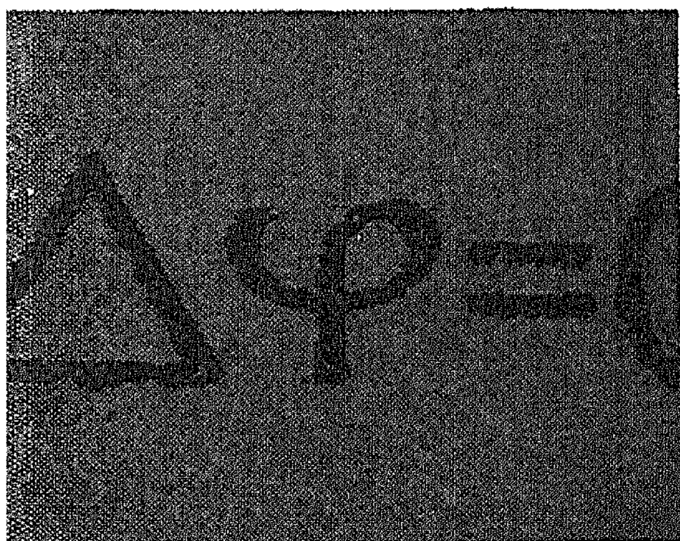
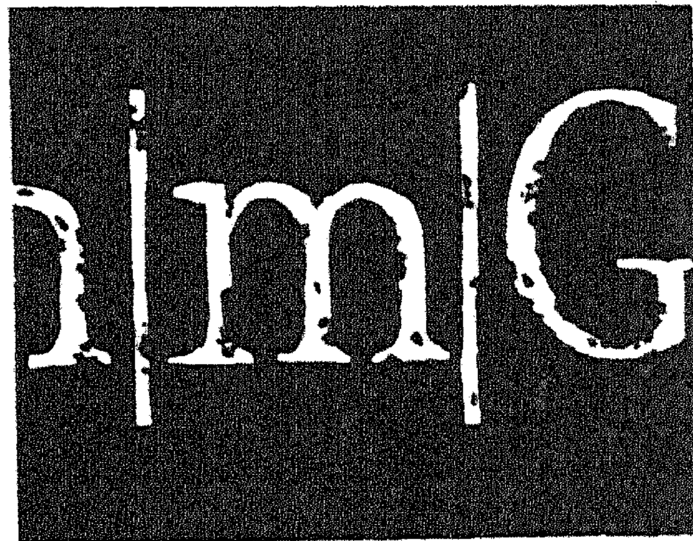
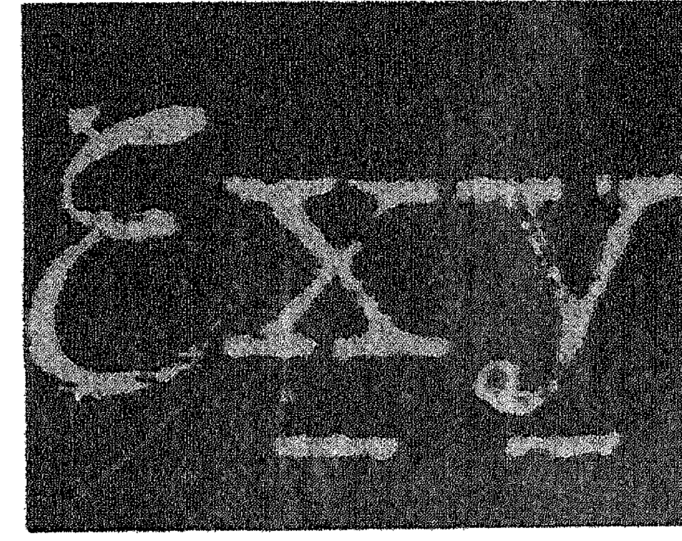
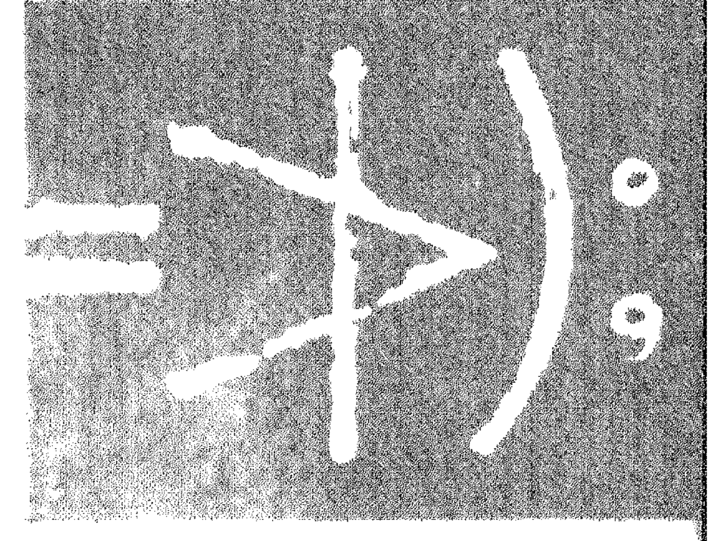
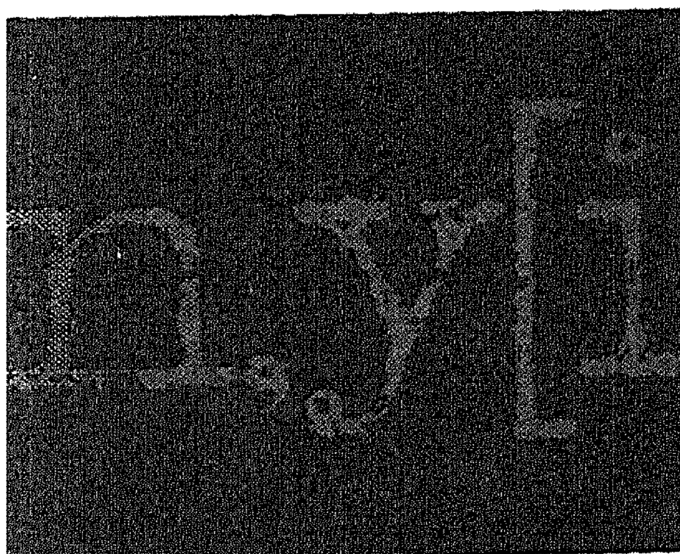
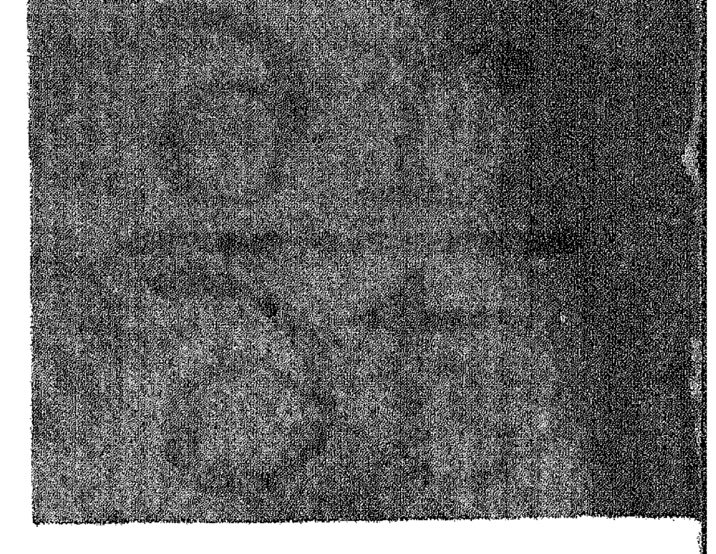
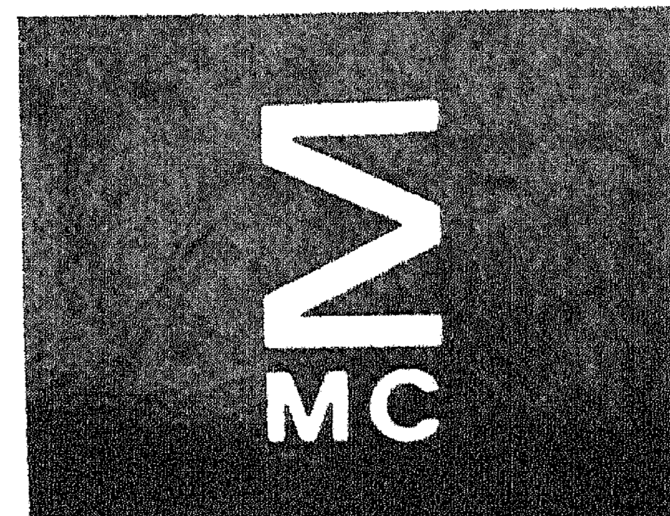
QUERY PROCESSING AND DATA ALLOCATION IN DISTRIBUTED DATABASE SYSTEMS

P.M.G. APERS



MATHEMATICAL CENTRE TRACTS



156

MATHEMATICAL CENTRE TRACTS 156

QUERY PROCESSING AND DATA ALLOCATION IN DISTRIBUTED DATABASE SYSTEMS

P.M.G. APERS

MATHEMATISCH CENTRUM AMSTERDAM 1983

1980 Mathematics subject classification: 68-02, 68B20, 68H05
1982 CR. Categories: C.2.4, H.2.2, H.2.4

ISBN 90 6196 251 X

Copyright © 1983, Mathematisch Centrum, Amsterdam.

CONTENTS

CONTENTS *i*
SUMMARY *v*
ACKNOWLEDGEMENTS *vii*

1 INTRODUCTION **1**

1.1 Subject of Thesis in Perspective *1*
1.2 Query Processing *2*
1.3 Data Allocation *3*

2 DISTRIBUTED DATABASE MANAGEMENT SYSTEM **5**

2.1 Database Architecture *5*
2.1.1 Computer Network *8*
2.2 The Relational Data Model *8*
2.2.1 Relational Database Management Systems *11*
2.3 Fundamental Problems *12*
2.3.1 Integrity and Consistency *12*
2.3.2 Recovery *13*
2.3.3 Data Allocation *14*
2.3.4 Query Processing *15*
2.3.5 Privacy and Security *16*

3 QUERY PROCESSING IN A DISTRIBUTED DATABASE **17**

3.1 Distributed Query Processing *17*
3.1.1 Three Phases in Query Processing *17*

3.1.2 Parsing a Query *17*
3.1.3 Determining a Schedule *19*
3.1.4 Execution of the Schedule *22*
3.1.5 Response Time of a Query *22*
3.1.6 Serializing Parallel Schedules Using the Same Resource *25*
3.1.7 Basic Operations *27*
3.1.8 Summary *28*

3.2 Overview of Distributed Query Processing Algorithms *28*
3.3 Query Processing with Inverted File Organization *38*

3.3.1 Inverted Lists as Storage Structure and Unit of Allocation *39*
3.3.2 Minimizing Response Transmission Time *43*
3.3.3 Disjunctive Normal Form *44*
3.3.4 Breaking a Term *48*
3.3.5 Equal Transmission Cost Model *53*
3.3.6 Arbitrary Transmission Cost Model *58*
3.3.7 Response Transmission Times Before and After Serialization *61*
3.3.8 Minimizing Response Time *62*
3.3.9 Minimizing Total Time *66*
3.3.10 Summary *73*

3.4 Distributed Query Processing Using Semi-Join *73*

3.4.1 Estimating Technique and Schedules *74*
3.4.2 Simple and General Queries *76*
3.4.3 Minimizing Response Transmission Time *79*
3.4.4 Minimizing Total Transmission Time *84*
3.4.5 Summary *91*

3.5 Comparison Between Distributed Query Processing Algorithms *92*

4 DATA AND OPERATION ALLOCATION IN A DISTRIBUTED DATABASE **98**

4.1 Data and Operation Allocation and File Allocation Problem *98*
4.2 Overview of File Allocation Problem *99*
4.3 Data and Operation Allocations and Their Costs *100*

4.3.1 Forking Points and Forking Graphs *104*

4.3.2 Unit of Allocation and Processing Schedules *105*

4.4 Centralized Data Allocation *110*

4.4.1 Construction of Processing-Schedules Graph *110*
4.4.2 Static versus Dynamic Schedules *112*

4.5 Decentralized Data Allocation *114*

4.5.1 Private Copies and Processing-Schedules Graph *114*

4.6 Comparison Centralized and Decentralized Approach *116*
4.7 Minimizing Total Transmission Cost *122*

4.7.1 Optimal Allocations Using Static Schedules *123*
4.7.2 Heuristic Allocations Using Static Schedules *127*
4.7.3 Theoretical Results Concerning Algorithm *total data allocation* *132*
4.7.4 Comparison Optimal and Heuristic Allocations Using Static Schedules *134*
4.7.5 Semi-Dynamic Schedules *135*
4.7.6 Optimal Allocations Using Dynamic Schedules *136*
4.7.7 Heuristic Allocations Using Dynamic Schedules *138*
4.7.8 Comparison Static and Dynamic Schedules *139*
4.7.9 Extensions of Algorithm *total data allocation* *141*
4.7.10 Constituents of Total Transmission Cost *141*
4.7.11 Primary Copies *144*
4.7.12 Summary *146*

4.8 Minimizing Average Response Time *147*

4.8.1 Queueing Model and Response Time *148*
4.8.2 Simple Processing Schedules *149*
4.8.3 Optimal Allocations to Minimize Average Response Transmission Time *150*
4.8.4 Heuristic Allocations to Minimize Average Response Transmission Time *153*
4.8.5 Comparison Heuristic and Optimal *ARTT* Allocations *155*
4.8.6 Minimizing Average Response Processing Time *156*
4.8.7 Optimal Allocations to Minimize Average Response Processing Time *158*
4.8.8 Heuristic Allocations to Minimize Average Response Processing Time *159*

4.8.9 Comparison Heuristic and Optimal *ARTT* Allocations *161*
4.8.10 Minimizing Average Response Time *163*
4.8.11 Arbitrary Processing Schedules *163*
4.8.12 Optimal Operation and Data Allocations to Minimize *ARTT* *163*
4.8.13 Operation Allocation Given a Data Allocation *165*
4.8.14 Heuristic Operation and Data Allocation to Minimize *ARTT* *169*
4.8.15 Summary *172*

5 SUMMARY AND CONCLUSIONS **174**

5.1 Query Processing *174*
5.2 Data Allocation *176*
5.3 Future Research *180*

REFERENCES **181**

INDEX **188**

SUMMARY

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 a network, making it possible to communicate with each other. On top of such a network a distributed database management system can be built so that the distribution of logical and physical components of the databases is kept hidden from the users.

The advantages of a distributed database compared with a centralized one are: increased availability, decreased access time, and easy expansion and possible integration of existing databases. The acceptance and wide-spread 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 stated by users and to determine allocations of the data such that the availability is increased and the access time is decreased. This monograph deals with the two dual problems query processing and data allocation. A query processing algorithm determines, given a data allocation, schedules for processing queries. A data allocation algorithm determines — given the queries and updates, how frequently they are stated, and a query processing algorithm — an allocation of the data of a database.

This monograph is organized as follows.

In chapter 1 query processing and data allocation are placed in perspective and an impression is given of the topics that will be investigated.

In chapter 2 a tutorial on distributed databases will be given. Besides the advantages some of the fundamental problems, which still require research, are touched on. For example, there is still no generally agreed on way of controlling concurrent accesses. Also, crash recovery techniques have hardly been studied in relationship with concurrency control mechanisms. Furthermore, query processing and data allocation problems are discussed.

In chapter 3 the problem of query processing is treated. Different cost functions are defined to measure the efficiency of a processing schedule. Especially, the computation of the response time of a query is complex, because the execution order of operations, competing for the same CPU, is not fixed. The sort of basic operations used in a schedule has a strong influence on the way queries are processed. On the one hand, low level operations that manipulate storage structures are considered, and, on the other hand, operations that merely decompose a query into subqueries at the logical level. The main part of this chapter is concentrated on the construction of processing schedules that minimize either the response time or the total time. A qualitative and quantitative comparison is made between the algorithms proposed and already existing algorithms.

In chapter 4 the problem of allocating the data of a database is investigated. The data allocation problem is essentially more difficult than the well-known file allocation problem, because the objects to be allocated have to be determined and also

more complex processing schedules are allowed. A model is introduced, wh
it possible to compute the cost of allocations under construction, taking int
the processing schedules.

A centralized approach to the data allocation problem requires the ex
a central organization that may decide about the allocation of all the data in
base. A decentralized approach, in which this is only possible for portio
database, is especially applicable to databases that are the integration of exis
bases. These two approaches are both qualitatively and quantitatively c
The main part of this chapter is concentrated on determining optimal alloc
means of the Heuristic Path Algorithm, developing heuristic algorithms th
polynomial time, and comparing the allocations obtained by the heuristic al
with the optimal ones. This is done for minimizing both the total transmis
and the average response time.

In chapter 5 the results obtained are discussed and we express our exp
about future research related to databases.

ACKNOWLEDGEMENTS

At the Vrije Universiteit, where this research was carried out, I would like to thank my advisor, 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 this monograph. I am much indebted to him. To Patricia G. Selinger I am very grateful for her comments concerning the presentation of the results.

Finally, I would like to thank the Mathematical Centre for the opportunity to publish this monograph.

1. INTRODUCTION

1.1. Subject of Thesis in Perspective

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 a computer network, making it possible to communicate with other computers. On top of such a network a distributed database management system can be built so that the distribution of logical and physical components of the databases is kept hidden from the users.

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.

In chapter 2 we will introduce the main ideas about distributed databases and we will deal with the above notions in more detail. To achieve these advantages, there are still many fundamental problems to be solved.

Maintaining the **integrity** of a database, while the data are simultaneously accessed by several transactions, with as much **concurrency** as possible to increase the throughput, has been investigated for both centralized and distributed database management systems. So far, no agreed on concurrency control mechanism has been developed. A closely related problem, which received little attention in relationship with concurrency control, is the **recovery** of a database if transactions, for whatever reason, fail to commit. Current concurrency control and crash recovery mechanisms put a heavy burden on the efficiency of database management systems.

Two other problems about efficiency can be viewed as dual problems: **query processing** and **data allocation.** Query processing is the problem of determining a materialization of the database and deciding at which sites to execute certain operations, such that the required result can be presented at the site where the query was issued, or, on request, at another site. This implies that a possibly redundant data allocation is given. The dual of this is the data allocation problem, where the access patterns are given, and the objects to be allocated and their allocation have to be determined.

The integration of existing databases requires the construction of a **global conceptual schema** on top of the conceptual schemas of the individual databases. In general, each of these local conceptual schemas has its own data language. So, the integration will require conversions between local and global conceptual schemas.

The social impact of databases and, especially, the integration of existing databases has culminated in **privacy legislation**. To carry this out, **security** tools will have to be developed. Unauthorized access to confidential data should be prohibited. Also, the storage and transmission of data should be made secure by, for example, encryption. The access to different databases, which has become easy if stored in a

computer network, must be regulated by integration of the access methods of these databases. Otherwise, unbridled access of the databases can not be stopped.

This monograph deals with the two dual problems, query processing and data allocation. It shows how they are related, and that, especially, the data allocation problem can not be studied without detailed knowledge about query processing. Because the acceptance and wide-spread usage of distributed databases highly depends on how efficient they will be, it is of importance to provide the distributed database management system with algorithms that compute efficient schedules to process queries, and with tools that both measure the efficiency of data allocations and decide about changes to improve it.

1.2. Query Processing

The first part of the monograph (chapter 3) deals with the problem of query processing in a distributed database. Cost functions that are used to measure efficiency, are the **total cost** and **response time**. The total cost can be viewed as the sum of the times spent in the different components of the computer network. Most of the research on distributed query processing has been on minimizing total cost. Here, ample attention will be given to minimizing response time of schedules for processing queries, as well. To achieve maximum parallelism the query is split into subqueries of which the results are processed at the site where the final result is to be delivered. To compute schedules for these subqueries we assume that none of the operations or transmissions share the same resource (**Parallelism Assumption**). Because of physical constraints the schedules obtained may not be executed completely in parallel. Therefore, they are **serialized**, such that parallelism that can not be achieved is removed.

Normally, a query is first decomposed into subqueries at the logical level and then efficient schedules for these subqueries are determined at the physical level. Obviously, the decomposition may lead to non-optimal solutions. Ideally, a query stated by a user is translated directly to basic operations without losing optimality. These **basic operations** are transmissions of data from one site to another, and operations that manipulate storage structures. This approach is discussed for the Boolean expressions of index terms with inverted lists as storage structure and the set operations to manipulate these inverted lists. An expression of set operations on inverted lists corresponding to a Boolean expression is rewritten into its disjunctive normal form such that the intersection operations are executed before the union operations. Furthermore, the union operations are executed at the result site. Also, before transmitting a large list it is, whenever possible, intersected with another smaller one to decrease its size.

It is interesting to note that, although the set operations differ from the relational operations, in the research on query processing in the relational data model a similar, however intuitive, approach can be observed: joins are computed at the result site and the relations are made as small as possible by applying semi-joins.

Backed by the theoretical results for the set operations on inverted lists, and the results obtained elsewhere for simple queries in the relational data model, we investigate whether the same approach can be used for more general queries. For each relation referenced in a query a schedule is determined to decrease the relation in size and to transmit its target and joining attributes to the result site, where the joins are computed. Intermediate result are estimated using selectivities.

For minimizing total transmission time, first the same approach is taken as for minimizing response time. The schedules for the subqueries are integrated to form the schedule for the query, such that identical transmissions are removed. Hence, parts of the schedules for the subqueries may be used for different subqueries. For this reason optimal results can not be obtained. The second approach tries to emphasize the collective use of schedules for subqueries. Finally, the two approaches are compared.

In most of the research on query processing it is assumed that transmission cost does not depend on the topology of the computer network. Here, the effect of an arbitrary topology on the optimality of the schedules and the complexity of the query processing algorithms is investigated.

A qualitative and quantitative comparison is made with other distributed query processing algorithms. Issues of comparison are: does the database management system have to supply a materialization beforehand or is it determined during optimization, which data is needed to estimate sizes of intermediate results, whether schedules are determined before or during execution, whether joins are computed solely at the result site or not, etc.. The quantitative comparison is only done with respect to the cost of the schedules obtained and the time required to compute them.

1.3. Data Allocation

The second part of the monograph (chapter 4) treats the problem of determining a, possibly redundant, allocation of the data of a distributed database. Again the two cost functions, total cost and response time, are used to measure the efficiency of allocations. Originally, the allocation problem was known as the **file allocation problem**: given the query and update transactions, and the frequencies of their execution, determine the allocation of copies of a file in a computer network. However, in a distributed database the problem is far more complicated. First, the objects to be allocated are unknown. A method is developed to partition the relations of the global conceptual schema into fragments. This partitioning is done by splitting the relation both horizontally and vertically. Secondly, the processing schedules produced by distributed query processing algorithms are far more complex than the ones used in the file allocation problem, in which only transmissions from the sites, where the files reside to the result site are allowed. In a distributed database, however, schedules also contain transmissions between sites where fragments are located. A consequence is that the allocation of the fragments can not be considered independently. The problem of finding an allocation of the fragments such that the total time is minimum, is shown to be NP-complete.

To obtain better insight in the **data allocation problem**, different lines of research are followed. Where possible the complexity of the problem is established. Admissible heuristic estimators are determined to compute optimal allocations with the Heuristic Path Algorithm (or branch-and-bound techniques). Also, heuristic algorithms that run in polynomial time are presented, and using simulations their results are compared with the optimal ones.

A model is presented which makes it possible to discuss redundant allocations where not all fragments have been allocated to fixed sites. The notions of **physical** and **virtual site** are introduced. The former represents a site in the computer network, including the fragments located there and the operations executed at it, and the latter represents the fragments and operations that have not found a final allocation yet. A

graph, called **processing-schedules graph**, is used to compute the cost of such an allocation.

Both for minimizing total cost and average response time heuristic algorithms are developed, which locally try to change allocations to decrease the cost. For minimizing total cost the effect of changing the query/update ratio on the degree of redundancy and the percentage of the transmissions that is required to keep copies consistent, is investigated.

Two different approaches to data allocation are compared. For a distributed database that is managed centrally, information about the access patterns can be obtained. This implies that a possibly redundant allocation can be computed for the database as a whole. This approach is called **centralized**. Opposite to this is the **decentralized** approach. If a database is the integration of already existing databases, it might not be possible to change the existing allocation. To still be able to improve the efficiency of the integrated database, the database administrator of this database is given the opportunity of creating private copies. A similar approach can also be taken if finding an allocation for a database as a whole is computationally not feasible. The data allocation problem is then partitioned into smaller problems, which can be solved more efficiently. The usage of private copies also enables the groups of users to determine how up-to-date their copies should be.

2. DISTRIBUTED DATABASE MANAGEMENT SYSTEM

What makes a database distributed and what are the advantages over a centralized one? Enslow [Enslow1978] has given us a definition of a **distributed data processing system.** For a distributed database management system this can be interpreted as a system where the logical and physical components are distributed and the local database management systems running at the sites of a computer network cooperate with each other, such that the users are unaware of the distribution.

The advantages of a distributed database over a centralized one are:

- **increased availability**
  The failure of a site or a communication channel does not necessarily imply the inaccessibility of the whole database. By storing the data redundantly the system can be made resilient to certain failures.
- **decreased access time**
  If the access pattern in a distributed database shows a high locality of reference, it is worthwhile to partition the database based on this and store the portions of the database at these localities. The access time will be significantly less than when all data are stored at a single site.
- **easy expansion and possible integration of existing databases**
  It is to be expected that the hunger for information both by industry and government will increase rapidly. Consequently, databases will continue to grow and, eventually, reach certain physical limits. The upgrading of capacity will be considerably easier in a distributed system than in a centralized one, as far as continuation of service is concerned. Also, data bases can be integrated to combine data from existing databases.

2.1. Database Architecture

A commonly agreed on architecture of a database is shown in fig. 2.1(a). It consists of three levels: the **external schema** (ES), the **conceptual schema** (CS), and the **internal schema** (IS). The idea behind this is to create independence both upwards and downwards. The conceptual schema contains all the available and required knowledge about, for example, a company to model it as a database. Because not all the users are interested, or are allowed to see the information contained in the conceptual schema, an extra level on top of it is created for each user or group of users. For example, an employee in the personnel department may access confidential information about other employees, while an employee at the sales department may only see inventory figures.

To improve the efficiency of a database, the database administrator is likely to change the storage structure every now and then. To avoid having to change the conceptual schema every time, the internal schema was created which contains all information required to access the data.

What happens to this three level representation if the database is distributed? Let us start with the bottom level. If only the data is distributed we can merely create a **local internal schema** (LIS_i) for every site S_i in the computer network and place them below the internal schema, which is now called **global internal schema** (GIS). If both the data and the description of them are distributed, each site S_i is given its own **local internal schema** (LIS_i) and **local conceptual schema** (LCS_i), which are placed below the conceptual schema, now called **global conceptual schema** (GCS).

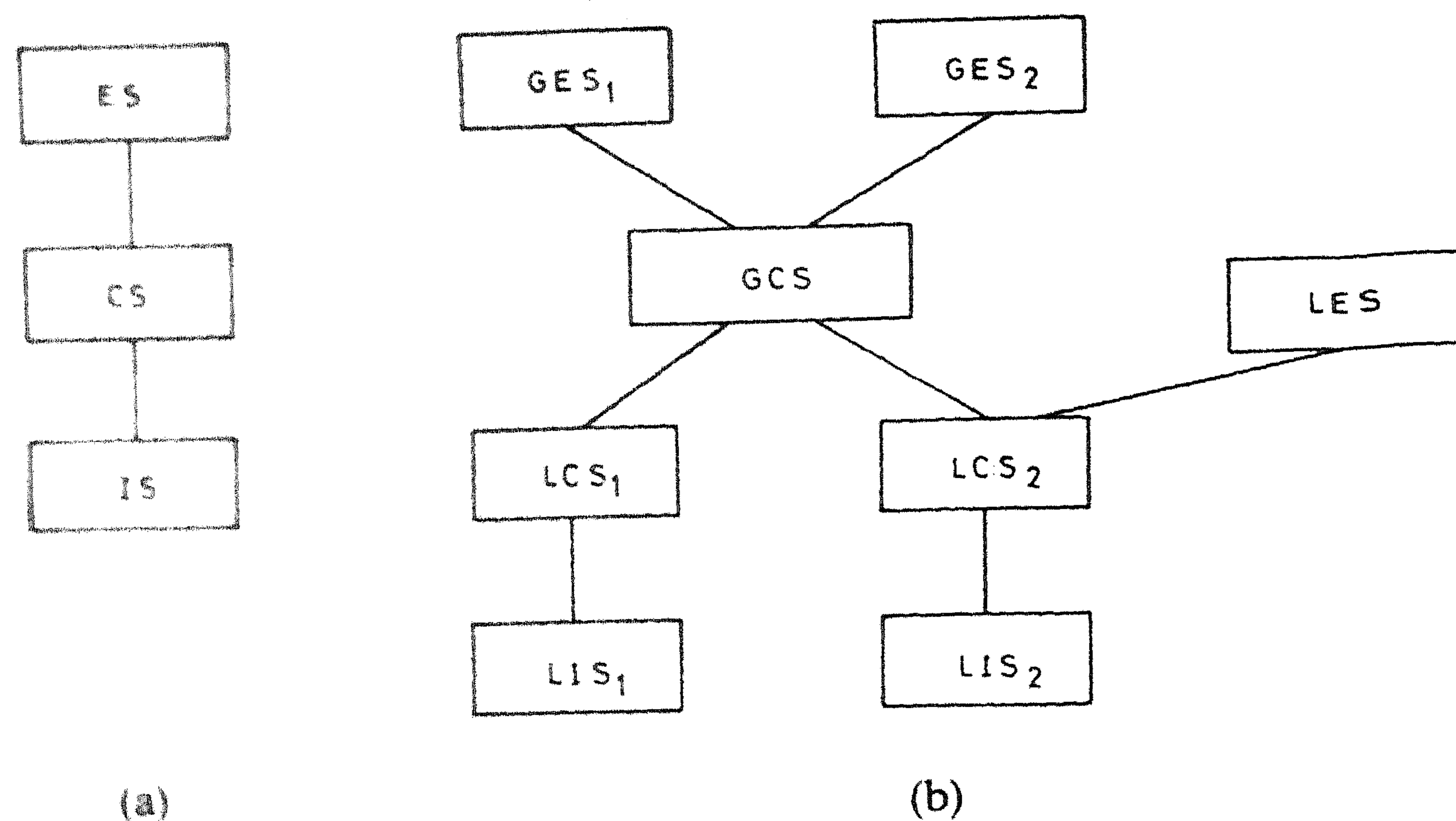


Figure 2.1. (a) Centralized and (b) distributed database architecture.

Sometimes the individual sites may even support their own local database and then a **local external schema** (LES) is placed on top of local conceptual schema. For each group of users of the distributed database a global external schema (GES_i) is placed on top of the global conceptual schema (see fig. 2.1(b)).

The conceptual schema was created with the idea of an integrated database, contrary to every one having its own files. For the same reason the global conceptual schema can be viewed as the union of all local conceptual schemas. If the data language both at the local and global level are the same, this database is called **homogeneous**. This type of database is, in general, obtained by a top-down design. For example, the database of one company which is distributed over several offices, which may be geographically dispersed.

Another type of databases, called **heterogeneous**, is, in general, obtained by a bottom-up design. In this case the distributed database is an integration of already existing, centralized databases. It is to be expected that many distributed databases will be of this type. Besides the problems occurring in a homogeneous database, of which there are still some that are not solved satisfactorily, there is also a tremendous translation problem from one data model to another. One would expect that a person is a person. Wrong! In databases one can have employees, taxpayers, subscribers, etc., and in each database the same person will have a different identification. Integrating the conceptual schemas into one global conceptual schema is far less than trivial, so, one may ask whether a doubly layered conceptual schema will suffice.

In [Litwin1980] a different approach is taken. It does not consider the physical distribution of the data, but the way the conceptual schema can be structured. The architecture of the conceptual schema is tree- or graph-structured. For example, there is a database for hotels, and another one for airlines. On top of these one may build third database for holidays that contains these two. It is just like abstract datatypes. Putting together procedures and data that logically belong to each other and letting other procedures that make use of them know as little as possible about the implementation details.

A graphical representation of such an architecture of a distributed database is shown in fig. 2.2. It shows a graph-structure where a box above another box with a line between them means that the schema corresponding to the upper box is built on top of the schema corresponding to the other box. An *E*-box stands for an external schema, a *C*-box for a conceptual schema and an *I*-box for an internal schema.

An internal schema on top of other internal schemas shows a database which is only physically distributed. A conceptual schema on top of other conceptual schemas shows a logical distribution of databases, or an integration of existing databases. Such an integration can be done in one step, as was shown in fig. 2.1(b), or in more steps with a graph or tree. It is also possible to construct a conceptual schema on top of a conceptual schema and an internal schema. This means the extension of a database with data for which no logical description is given in a separate conceptual schema. Also, an external schema may be constructed on top of two other external schemas, meaning accesses to different databases that are not integrated. Note, that in this case no security can be provided. If this is required an additional conceptual schema should be constructed on top of the conceptual schemas of the databases accessed. Finally, the external schema is then constructed on top of the newly constructed conceptual schema. So, this structured approach to the architecture may also provide a model to implement security tools. This seems an attractive approach, however, more practical experience will be required to see whether more complex problems can be solved with it.

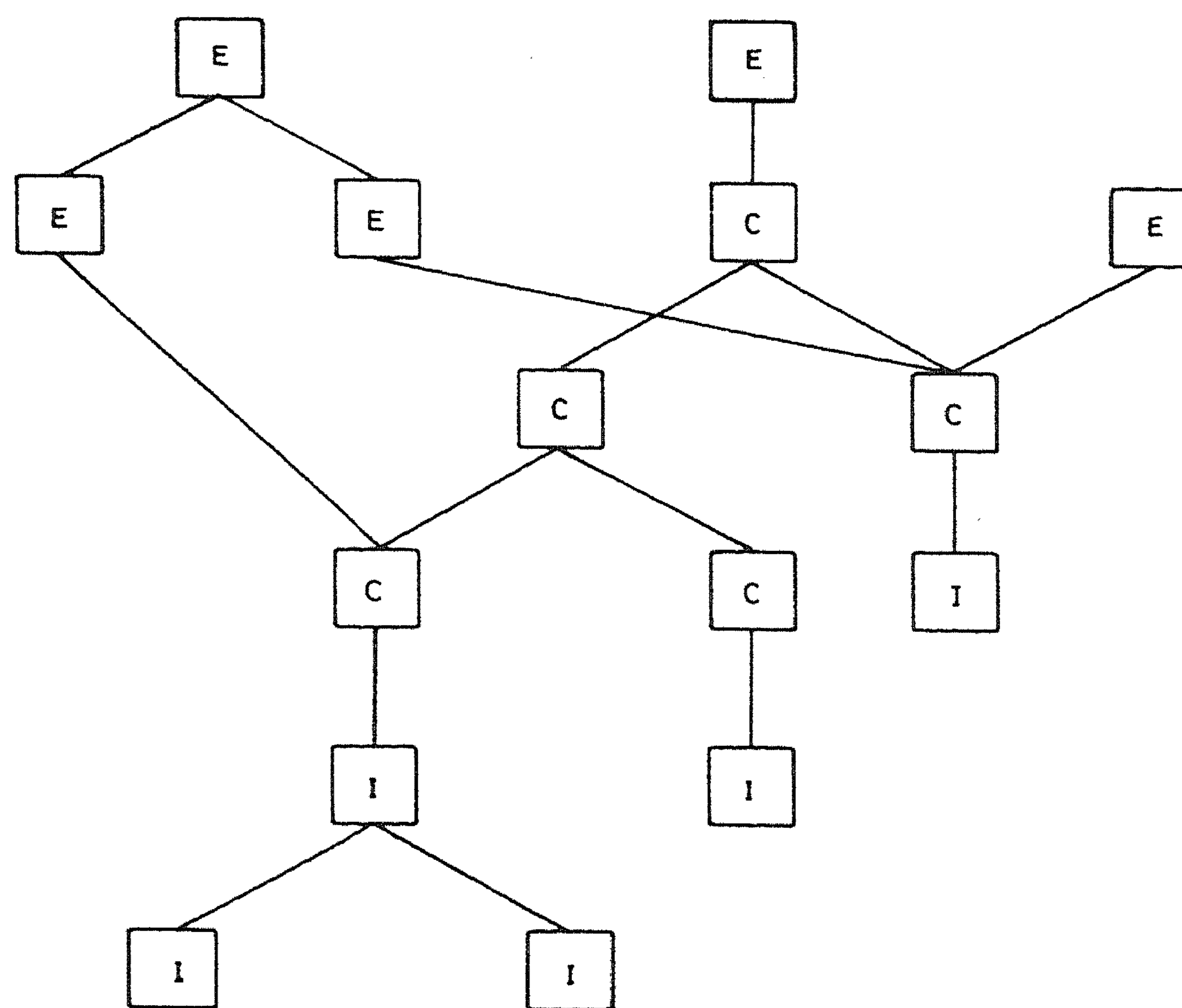


Figure 2.2. Structured architecture of a distributed database.

2.1.1. Computer Network

The relationship between a distributed database management system and a computer network is easily explained by using the OSI model. In the ISO proposal [Zimmerman1980] a network consists of 7 layers. We will briefly discuss the function of each layer; a more detailed description is given in [Tanenbaum1981]. The layers are numbered from 1 to 7, where 1 is the lowest level. Layer 1 is called the **physical layer** and is merely concerned with the transmission of bits. The second layer, called **data link layer**, deals with the transmission of the frames, containing data or acknowledgements, between IMPs. The **network layer**, being the third layer, transmits packets along routes which may be determined dynamically. The fourth layer, called the **transport layer**, is the first layer that provides communication between hosts. Its task is to deliver messages error-free and in the order in which they were offered. The **session layer**, the fifth layer, is the user's interface to the network. It makes an orderly dialogue with another machine possible. The sixth layer, called the **presentation layer**, can be viewed as a library containing useful programs to make communication a bit more friendly. Finally, in the **application layer** we find the **distributed database management system.** In a general-purpose computer network it will run on top of a **network operating system** or a **distributed operating system.**

2.2. The Relational Data Model

Among the three well-known data models, namely the **relational**, the **hierarchical** and the **network** [Date1977, Martin1975, Tsichritzis1977, Ulmann1980], the relational [Codd1970] is, in the scientific world, the most widely accepted data model for modeling the data of a distributed database.

A **domain** is a set of values, which is denoted by D. The **Cartesian product** of domains $D_1, D_2, \ldots, D_n$, denoted by $D_1 \times D_2 \times \cdots \times D_n$, consists of all n-tuples $(d_1, d_2, \ldots, d_n)$, where $d_i \in D_i$ for $i = 1, 2, \ldots, n$. A **relation** R is a finite subset of a Cartesian product and is represented by a table. Each row is called a **tuple**, and each column an **attribute.**

Example 2.1

Let *YEAR DMN* be the set of natural numbers, *NAME DMN* the set of names of wines, *PRODUCER DMN* the set of names of producers, *AREA DMN* the set of names of areas, and *COUNTRY DMN* is the set of names of countries. Fig. 2.3(a) shows the relation *WINE* with attributes *YEAR*, *NAME*, *PRODUCER*, *AREA* and *COUNTRY*, which is a subset of the Cartesian product

$$YEAR\ DMN \times NAME\ DMN \times PRODUCER\ DMN \times AREA\ DMN \times COUNTRY\ DMN.$$

Each tuple represents a wine of which the grapes were grown in a certain area, picked in a certain year, and which was bottled by a certain producer.

Fig. 2.3(b) shows a 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.

□

Two data languages can be used to describe queries and updates, the relational algebra and the relational calculus. The main operations in the **relational algebra** are

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)

Figure 2.3. (a) relation *WINE* and (b) relation *WEATHER*.

selection, projection, join and the normal set operations.

- **selection**
  The result of a selection is a subset of the tuples of a relation R, which satisfy a certain condition. It is denoted by $R\{C\}$.
- **projection**
  The result of a projection over attributes A of a relation R, is a relation with attributes A, and from which duplicate tuples are removed. It is denoted by $R[A]$.
- **join**
  A join is an operation that works on two relations, R_1 and R_2. A tuple from one relation is concatenated with a tuple from the other relation if certain conditions $C_1, C_2, \ldots, C_n$ are met. Such conditions concern the comparison of the value of an attribute from a tuple in one relation with the value of the corresponding attribute from a tuple in the other relation. In general, this comparison may be complex, and in that case the join is called a **natural join**. If, however, the comparison is a simple test for equality, it is called an **equi-join**. The join is denoted by $R_1(C_1,C_2, \ldots, C_n)R_2$.
- **set operations**
  By viewing a relation as a set of tuples, the operations **union**, **intersection** and **difference** are defined as in set theory.

These operations are relationally complete [Codd1970].

A query stated in the relational algebra can be represented by a tree, where the leaves represent the relations in the database and the internal nodes the results obtained by a relational operation.

Example 2.2

Assume the following query is posed by a user:

> *Give the names of the wines and the year in which the grapes were picked in which the area got more than 1800 hours of sun and got less than 1200 mm. of rain.*

In the relational algebra this looks like

$$((WEATHER\{SUN > 1800\ AND\ RAIN < 1200\}) \\ (YEAR = YEAR, AREA = AREA)WINE)[YEAR, NAMES]$$

The result is the Margaux wine (1970) and the Cabernet Sauvignon from California (1976).

□

To evaluate queries efficiently a new operation, called **semi-join**, was introduced [Palermo1974, Bernstein1981a, Hevner1979b]. Although this operation is just as independent of the implementation of the relations as the relational operations, it is considered to be a low level operation, whose only purpose is to process queries in a more efficient way. The result of a semi-join on relation R_1, based on a particular join between R_1 and R_2, can be defined as the result of the join and projection over the attributes of R_1. The semi-join is computed by projecting R_2 over the attributes involved in the join and selecting those tuples of R_1 that satisfy the joining condition with any of the tuples in the projected R_2. The semi-join is a useful operation to reduce the sizes of the operands, which makes the computation of the join more efficient. Furthermore, if the underlying join is an equi-join, the result can easily be obtained from the two relations, R_1 and R_2, after the semi-joins have been applied, by concatenating the tuples of R_1 with matching tuples of R_2.

The other data language is the **relational calculus**. The main difference between the two languages is the way the result is described. In the algebra the query describes the way the result can be obtained and in the calculus the result is described by giving the conditions which the result tuples have to meet.

A query in the relational calculus can be represented by a graph. The nodes are the relations, and the edges are labeled with the condition that has to be satisfied by the tuples in the adjacent relations. In case a condition concerns only one relation the two ends of the corresponding edge are connected with the node that corresponds to that relation. To express which attributes are part of the resulting relation an extra node is added, called the **target**. This target can be viewed as a relation, which is the Cartesian product of the domains of the attributes in the result relation.

Example 2.3

We will give the query of example 2.2 in QUEL [Held1975] the data language of INGRES [Stonebraker1976]. For each of the relations accessed in the query, a

range variable is introduced. The values that such a variable can obtain are the tuples of the relation mentioned in the RANGE-statement. In QUEL the query looks like

RANGE OF *WI* IS *WINE*
RANGE OF *WE* IS *WEATHER*
RETRIEVE INTO *W*(*WI.NAME*,*WI.YEAR*)
WHERE *WI.YEAR* = *WE.YEAR* AND *WI.AREA* = *WE.AREA*
AND *WE.SUN* > 1800 AND *WE.RAIN* < 1200

The graphical representation of this query is given in fig. 2.4.

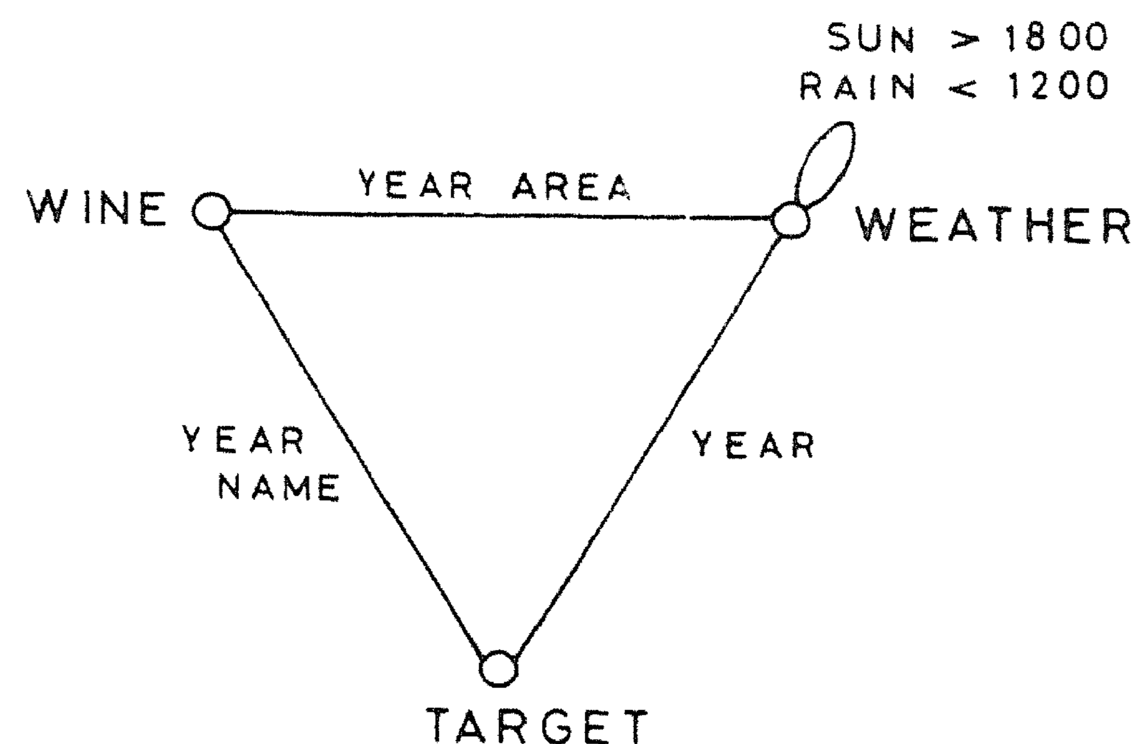


Figure 2.4. Graphical representation.

□

2.2.1. Relational Database Management Systems

So far, few database management systems have appeared that support the relational data model. In the academic environment INGRES (Interactive Graphics and Retrieval System) has been developed at the University of California in Berkeley [Stonebraker1976]. It supports the data language QUEL [Held1975] which is based on the relational calculus. System R, developed at the research laboratory of IBM in San Jose [Astrahan1976] has a data language, called SQL [Astrahan1975] which is based on the relational calculus. More recently IBM has offered a relational database system called SQL/Data System for the DOS operating system. At the Thomas J. Watson Laboratory the Query By Example system, which supports the QBE data language, was developed. Two systems were developed at the University of Toronto, the ZETA and OMEGA system [Czarnik1975, Schmid1975]. At the University of California San Francisco and the Vrije Universiteit a programming language called PLAIN [Wasserman1981], which embeds the relational algebra, has been developed, and a database management system [Kersten1981] to support it is under construction. For a more complete list of relational systems we refer to [Kim1979, Chamberlin1976].

The development of general-purpose distributed database management systems that support the relational data model, has started. For both INGRES [Stone-

braker1975] and System R [Williams1981] distributed versions are on their way. In Canada we have the ADD system [Toth1978] in France SIRIUS-DELTA [LeBihan1980] and POLYPHEME [Adiba1980a] and in Germany the POREL system [Neuhold1977].

2.3. Fundamental Problems

To avoid the impression that distributed databases have only advantages and that they can be obtained at no cost, we discuss a few fundamental problems that still require extensive research.

2.3.1. Integrity and Consistency

To make a database work, its **internal integrity** should be safeguarded. To start with, all update transactions have to change the database such that after their termination the data are consistent. A simple way of implementing this is to check integrity constraints prior to committing a transaction. If a constraint fails the transaction must be undone, for example via a rollback [Eswaran1976]. Assume that one of the integrity constraints of a database is that the sum of the two data items, A and B, must remain the same, for example zero, after each transaction. A transaction that updates only A or B, or both of them but with $A = B \neq 0$ as result, will violate the constraint and must be undone, such that the values of A and B before the execution of the transaction are restored. Only transactions that modify both A and B such that the result of their sum is again zero, may be committed.

The next problem arises when **concurrent** execution of transactions is allowed. If these transactions can freely access the data, they may read **inconsistent data**, and the following problems may arise. For example, if two transactions try to update the same data item without being controlled when accessing the data item, it might happen that the update of only one transaction becomes effective, because the other is overwritten. Hence, one update is lost. Another example is that during the execution of a transaction the database will be in an inconsistent state, and if other transactions are not prohibited from reading that particular part of the database that is changed, they will show inconsistent data to the users.

Most concurrency control mechanisms order the execution in such a way that it is equivalent to executing the transactions one after another. It has been shown [Eswaran1976] that if the execution of the transactions can be made **serial** the resulting database is consistent. This property of serializability is obtained by two-phase locking or by timestamp ordering. Transactions that run under a **two-phase locking** regime must obtain locks for every data item they want to access and no more locks may be acquired after one or more locks have been released. A simple way of implementing this regime is to designate one of the sites as a central site, to which all lock requests must be sent [Garcia-Molina1979a, Garcia-Molina1979b]. To detect deadlock it constructs **wait-for graphs**. Based on these graphs locks are granted or not.

Another way to serialize the transactions is to have a **total ordering** based on time. This is done by giving transactions a timestamp. A decentralized approach is to let each site have its **local clock** [Thomas1979]. To prevent local clocks from getting out of phase, local times are exchanged on a regular basis [Lamport1978]. Each transaction receives a **time stamp** which is the maximum of the local time or the highest time stamp of the local data item plus one. This implies that each data item

receives the time stamp of the transaction that last changed it. In the algorithm of Ellis this universal time is constructed by maintaining a local counter at each site, containing the last number issued for a transaction initiated at that site. A global identification is obtained for a transaction by adding the site number. A similar approach can be found in the algorithm of Rosenkrantz et al. [Rosenkrantz1978], where the time of the day the transaction first started, the initiating site and some priority compose the identification. In the algorithm of LeLann [LeLann1978] a token circulates around some virtual ring of sites and each site that wants to initiate a transaction can get a ticket from the token. In this way each transaction receives its own number, determining the order in which the transactions are executed. So, the ticket number can be viewed as a kind of time.

For a more complete overview we refer to [Wilms1980, Bernstein1981b].

2.3.2. Recovery

The execution of a transaction may be ended for several reasons. A user may abort his transaction, a system crash may occur, if the concurrency control mechanism detects a deadlock it will abort a transaction involved, etc. In all these cases the changes caused by the aborted transaction should be undone. The state of every transaction is either that it is **committed**, or that it still can be **undone**. If it is committed the values of the items in the database should reflect the changes caused by it. Otherwise, the database may be in an inconsistent state and the concurrency control mechanism should prohibit other transactions from reading the inconsistent data.

If a system crash occurs the database should be brought back into a consistent state. This can be obtained by undoing all transactions that are not committed. To avoid the undoing of other already committed transactions (**domino effect**), no transaction may read data that is changed by a transaction that is not committed yet. If **check points** are used it is not always necessary to undo a complete transaction, only that part after the latest checkpoint. However, here the relevant part of the transaction should be redone to obtain a consistent state. Obviously, the changes of an already committed transaction should be reflected in the database.

To make sure that the database that uses locking can recover from a system crash the transaction should lock and unlock the accessed data in a **two-phase** manner. This means no lock may be obtained after one or more items are unlocked. Furthermore, entries in a log, stored on **stable storage**, should be created, containing information about undoing and redoing transactions. Finally, a **two-phase commit** protocol is needed for distributed transactions. In the first phase the transaction is executed and a master waits until all its slaves have done their duty and recorded on stable storage all information needed to undo the transaction. If one slave was unable to do its task, the complete transaction is undone. If, on the other hand, all slaves reply positively, the second phase is entered. The coordinator decides to commit the transaction by writing a commit entry in the log and after that all slaves are notified. They, in their turn, release the locks and reply with an acknowledgement. In some protocols the master needs a positive acknowledgement from all its slaves.

The requirement that a database should be recoverable from a system crash makes the interleaved execution of transactions more difficult, and hence the throughput will be less than without a recovery mechanism. The integration of the mechanism that controls the concurrency and the one that controls the recovery, such

that a high concurrency can be obtained, is a problem that still needs much research.

For a more complete overview we refer to [Kohler1981].

2.3.3. Data Allocation

Tuples occur in the same relation because they give information about the same attributes of somebody or something. So, they are grouped together because there is a logical relationship between them. In reality, one group of tuples is used most of the time in New York and another group in Amsterdam. Obviously, splitting the relation and locating one fragment on one side of the ocean and the other on the other side will tremendously decrease intercontinental traffic. This is called **horizontal splitting**. For the same reason a relation may be **split vertically**, because one location is more interested in only certain attributes and another in other attributes.

If the relations are split horizontally and/or vertically and each fragment is placed at exactly one location, the database is called **partitioned**. If none of the relations is split and a copy is placed at each location the database is called **fully replicated**. Combinations are called **partitioned** and **replicated**.

Besides the data in the relations, a database also contains in its data dictionary data necessary to parse queries, to check authorization, to compute processing schedules, etc.. Some of the entries will be volatile, while others hardly ever change. Therefore, the data dictionary should be treated just as a relation. For each entry an allocation can be determined based on the frequency with which it is accessed and how frequently it is updated.

In chapter 4 the problems involved in determining the allocation of all the data in the database such that query processing can be done efficiently, are discussed.

Example 2.4

One way to partition the relation *WINE* is to split it based on the countries that produce wines, for example

$$\begin{aligned} WINE\,F &= WINE\{COUNTRY = \text{FRANCE}\} \\ WINE\,I &= WINE\{COUNTRY = \text{ITALY}\} \\ WINE\,U &= WINE\{COUNTRY = \text{USA}\}. \end{aligned}$$

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

$$\begin{aligned} WEATHER\,R &= WEATHER[YEAR, AREA, COUNTRY, RAIN] \\ WEATHER\,S &= WEATHER[YEAR, AREA, COUNTRY, SUN]. \end{aligned}$$

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

□

One advantage of a distributed database over a centralized one is the increased **reliability**. This means that certain components, such as sites or communication channels, may fail without causing a total failure. What is really meant is the increase of the **availability** of the database. By storing data redundantly at different sites the

failure or inaccessibility of a site does not necessarily mean that the users of the database can no longer access the required data.

In [Mahmoud1976] the file allocation problem was studied. For each file a lower bound on the availability is specified and an allocation is feasible if the availability of each file is higher than the specified one. There is, however, one problem, the availability can not easily be expressed analytically for general network topologies [Hansler1972], because of the routes to different copies of one file are not necessarily disjoint. In [Mahmoud1976] the availability was, therefore, estimated.

In a distributed database the availability problem is even more complex. A user is not interested in the availability of a particular fragment of a relation, he is only interested in the availability of all the fragments he wants to access in his query. So, it is impossible to compute the availability of one fragment, the availability of the whole database should be considered.

2.3.4. Query Processing

To get an idea of what distributed query processing looks like we will discuss some of the problems involved. Assume we want to process the query

> *Give the wines, the years in which the grapes were picked and the hours of sun in the years that the area in which the grapes were grown got more than 1800 mm. of rain*

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. Other algorithms take full advantage of the redundancy and will select copies during optimization.

The query can be processed in many ways, we will discuss only two.

Schedule 1

Transmit $(WEATHER\{RAIN > 1800\})[YEAR,SUN,AREA]$ 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 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 the union and $WEATHER$. If the size of the result is 1400 bytes the total number of bytes transmitted is $12{,}000 + 15{,}000 + 20{,}000 + 1400 = 48{,}400$.

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, causing a smaller response time.

In chapter 3 many of the problems involved in distributed query processing are discussed in detail.

2.3.5. Privacy and Security

In the past the privacy of people was maintained simply because the document files were kept in separate offices or departments. To obtain confidential data was a difficult and time-consuming task. Later, the files were put on tapes or disks making it possible to retrieve data from a remote terminal by dialing up. In the near future, when distributed databases will be more common, combining data from several databases will become relatively easy. Because of the technological advances in the area of computer networks and databases, legislation is needed to ensure the privacy of people.

To maintain privacy, tools are required to make the system secure. We will discuss some of the safeguards. When a person wants to log-on he/she needs to identify his/herself (**authentication**), for example, by supplying a password. During a session a user may want to access certain data or run a particular application program. It should be checked whether a user has been granted that permission in the past (**authorization**). Furthermore, a log file should be kept that contains entries about users that wanted to execute application programs. Data kept on secondary storage should be **encrypted**. For a distributed database there is the additional problem of transmitting data over insecure communication channels. The same techniques can be used, namely encryption. U.S. National Security Agency has tried to prohibit open publication of research papers concerning cryptology [Davida1981, Denning1982]. So, it is to be expected that communication channels will be the one of the weakest links in distributed databases.

The mere fact that a database is distributed does not necessarily pose more difficult problems as far as authorization is concerned. However, the database may be controlled in a decentralized way, meaning that there is not just one database administrator. For System R a decentralized authorization scheme has been developed that gives owners of certain rights the opportunity to grant other users the same rights [Griffiths1976]. A right may be the right to retrieve certain data, to insert data in a certain relation, the right to grant rights, etc.. Such a scheme will be useful in the environment of a distributed database.

3. QUERY PROCESSING IN A DISTRIBUTED DATABASE

Query processing in a distributed database means retrieving data from several, possibly geographically dispersed, sites in a computer network. One of the advantages of a distributed database is that the data can be located at the sites where they are most heavily used. But even if the data is allocated in such a way, it is still important in query processing to efficiently combine the required data from the involved sites to compute the desired result. Different functions can be used to measure this efficiency: one can favor parallelism, another the minimum use of resources, etc. This efficiency is not only important for the user, who may have to wait a long time before he gets his answer, but also for the system, which may get congested because of inefficient use of its resources.

In section 3.1 we discuss the way queries are processed and how the efficiency can be measured. An overview of current research on distributed query processing models and algorithms is given in section 3.2. Both section 3.3 and 3.4 deal with minimizing response time and total time. In the first one the allowed operations manipulate inverted lists and in the latter the relational operations together with the semi-join are used. Some of the currently known algorithms are compared and suggestions for improvements are given in section 3.5.

3.1. Distributed Query Processing

3.1.1. Three Phases in Query Processing

Query processing in either a centralized or a distributed database consists of three phases:

- ☐ parsing a query,
- ☐ determining a schedule,
- ☐ execution of the schedule.

These three phases and the interfaces between them and other parts of the distributed database management system are shown in fig. 3.1. All communication with other sites goes through the distributed operating system (DOS).

As far as query processing is concerned an update and a query are alike. We can consider an update as consisting of a query-part, to determine the tuples to be updated, followed by the actual change of the tuples. In the following we will therefore use the word query if we mean query or update.

3.1.2. Parsing a Query

In the first phase the query is parsed, by a **parser,** just like a computer program, to check its syntax and semantics. With the aid of **data dictionaries** the parser checks the use of attributes of relations, such as whether this relation has an attribute called so and so, or, if a comparison is made between attributes, whether the domains are compatible, etc.. For an update it will also check whether integrity constraints are violated. If the query is correct it is translated from the global external schema to the global conceptual schema.

During this phase we might also check whether the user that stated the query

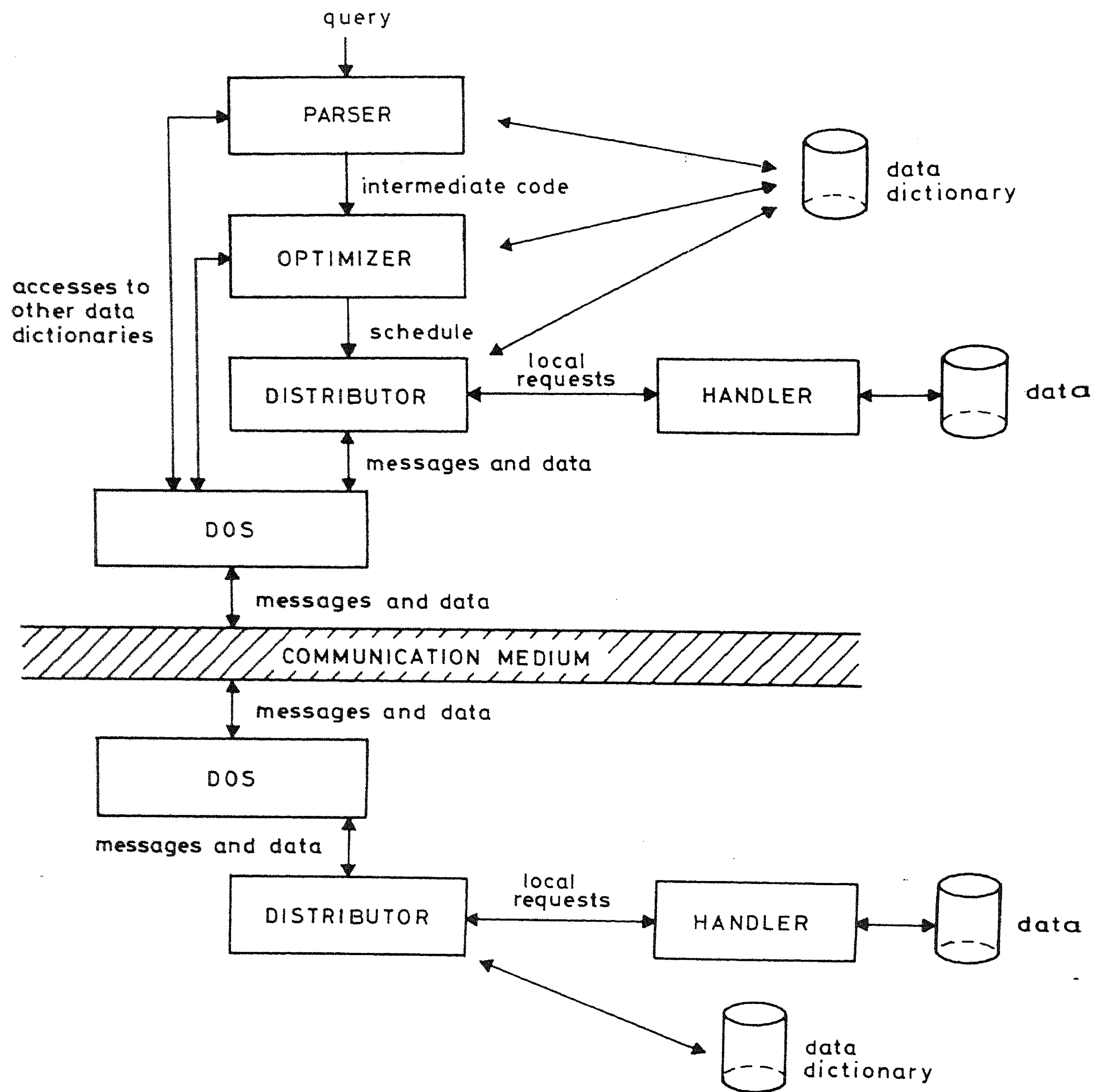


Figure 3.1. Three phases in query processing.

has the right to access the referenced relations. Or, another approach is to add clauses concerning integrity constraints to the query such that this check can be done at runtime [Stonebraker1976].

3.1.3. Determining a Schedule

If the query is correct and the user is allowed to state the query the second phase is entered. An **(query) optimizer,** also called query processing algorithm, will determine a schedule to process the query. A **(processing) schedule** contains the tasks, called **basic operations,** of the involved sites, which together deliver the final result to the user. So, the optimizer decomposes the query into basic operations. What these basic operations look like depends on whether the query is mapped to the local external or conceptual schema, or to the local internal schema. In the first case the basic operations will be relational operations on local fragments and transmissions of data. See for a more detailed discussion on basic operations subsection 3.1.7.

Assume that our example distributed database consists of the following two relations:

> *PARTS*(*P#* ,*PNAME* ,*QOH*) located at site S_1
> *PROJECT*(*P#* ,*S#* ,*QTY*) located at site S_2

and the query

> RANGE OF *P* IS *PARTS*
> RANGE OF *J* IS *PROJECT*
> RETRIEVE *P#* ,*PNAME* ,*QTY*
> FROM *P* ,*J*
> WHERE *P.QOH* $>$ 1000 AND *J.QTY* $<$ 500 AND *P.P#* = *J.P#*

originates at S_3. There are many ways to process this query. We will discuss only one. The basic operations are the relational operations and data transmissions. In fig. 3.2 two representations of the corresponding schedule are given. The first one merely indicates the data transmissions from one relation to another. The arrows may be labeled with the amount of data transmitted. The other representation that is used, is more or less the same as the one in [Cellary1980]. For each site a horizontal line is drawn that denotes a time scale. The time spent in executing an operation is measured along the scale of the site that does the computation. The longer an operation takes the longer its portion of the scale is. Let the **processing time** of an operation be denoted by $PT(X) = P_0 + PC(X)$, where P_0 is the queueing time and $PC(X)$ is the processing cost, expressed in units of time. X stands for the operation, sometimes in the form of the result after the operation. The transmissions from one site to another are drawn by a slope, connecting the starting and end point of the transmission on the two time scales. The **transmission time** of data is denoted by $TT(X) = T_0 + TC(X)$, where T_0 is the queueing time and $TC(X)$ is the transmission cost, expressed in units of time. X stands for the data to be transmitted. Here, we assume that whenever a job is offered to a server, the job will run until completion before the server starts with the next job.

First the duties of the involved sites are transmitted by S_3. Then the two res-

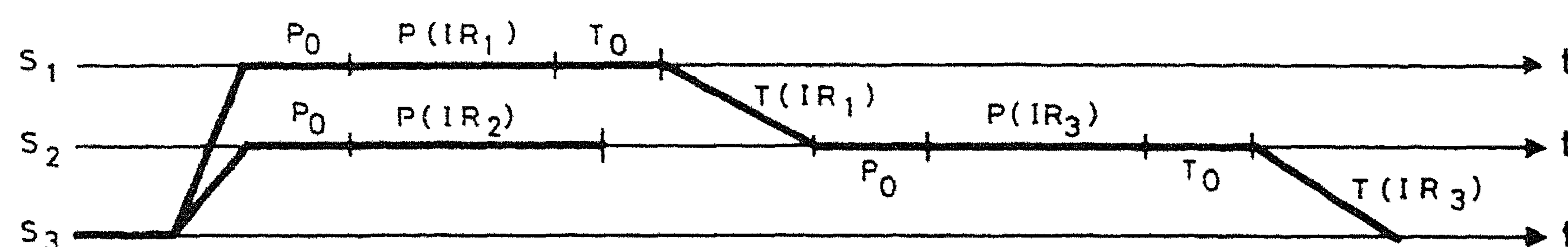
trictions, $P.QOH > 1000$ and $J.QTY < 500$, and the projections are applied to both relations. The results are denoted by IR_1 and IR_2, where

$$IR_1 = PARTS\{QOH > 1000\}[P\#, PNAME], \text{ and}$$
$$IR_2 = PROJECT\{QTY < 500\}[P\#, QTY].$$

The processing times are $PT(IR_1)$ and $PT(IR_2)$. Under the assumption that IR_1 is the smallest one of the two results it is transmitted to S_2; transmission time $TT(IR_1)$. As soon as IR_1 arrives at site S_2 the computation of the join can commence; $IR_3 = IR_1(P\# = P\#)IR_2$. Finally, the result is sent to the user's site, S_3.

$$PARTS \rightarrow PROJECT \rightarrow \text{result site}$$

(a)



(b)

Figure 3.2. Two representations of a schedule.

To compare schedules we have to know their costs. The **cost of a schedule** is the value of a particular cost function given this schedule. This cost function may measure the response time, the total time, the total network traffic, the total CPU time, etc. Having several cost functions is useful. Some users work interactively and are therefore only interested in the response time. The optimizer should in that case allow for as much parallelism as possible. On the other hand, a user might only be interested in the cost of processing his query without regard to response time and then the use of resources should be minimized. Intuitively, we can define the **response time of a schedule** as the time elapsed between the start of the first operation and the moment the required data is presented to the user. In the next subsection we will give another definition, which is more useful for optimization purposes. The response time of the schedule of the example query is

$$PT(IR_1) + TT(IR_1) + PT(IR_3) + TT(IR_3),$$

where IR_1, IR_2 and IR_3 are defined above. The **total time of a schedule** is defined as the sum of the times required for all operations and transmissions, involved in the schedule, including the queueing times. And the **total cost of a schedule** is defined as

the sum of the cost of all operations and transmissions. The total time of the example schedule is

$$PT(IR_1) + PT(IR_2) + TT(IR_1) + PT(IR_3) + TT(IR_3),$$

and the total cost

$$PC(IR_1) + PC(IR_2) + TC(IR_1) + PC(IR_3) + TC(IR_3).$$

In a distributed environment often the assumption is made that local processing takes negligible little time compared to the transmission time of data. The line segments in the schedule that represent local processing shrink to length zero. In that case we use the **response transmission time** (RTT) instead of response time, **total transmission time** (TTT) instead of total time and **total transmission cost** (TTC) instead of total cost. The value of a cost function of a query is the value of that cost function of a schedule for that query that minimizes the cost function.

The data of the database may be stored redundantly. A **materialization** is a non-redundant version of the database. Some systems provide the optimizer with a predetermined materialization. The advantage is that not all copies have to be mutually consistent. With **mutual consistency** we mean that the copies are exactly the same all the time. If, on the other hand, we know that the copies are mutually consistent we can take full advantage of the redundancy by letting the optimizer itself determine which copies to use.

The optimization objective may depend on the type of network on which the distributed database is placed. For example, for an ARPA-type network with low bandwidth communication channels, the transmission cost will dominate the local processing cost. In that case it suffice to minimize cost functions such as response transmission time or total transmission time. If, on the other hand, the data is distributed over a local network consisting of micro-computers that are interconnected with a fast bus then both the transmission cost and the local processing cost will be comparable [Selinger1980]. Hence, cost function such as response time and total time should be minimized. Here, we will assume that the cost to transmit data is an order of magnitude more expensive than local processing (**Transmission Assumption**).

The time required to transmit data from one site to another depends on the network topology and the bandwidth of the communication channels. If all sites are directly connected with each other, for example when they communicate via a satellite, and when the queueing delays before transmission are, on the average, the same we will speak of the **Equal Transmission Cost Model**. If, on the other hand, the network has an arbitrary topology or queueing delays vary too much we will speak of the **Arbitrary Transmission Cost Model**.

Because the main objective of this research is to minimize cost functions that include only transmission cost, just a simple model is used for local processing. We assume that the time required to process data locally is proportional to the amount of data (**Local Processing Assumption**). Much research is being done to develop optimizers that produce efficient schedules for queries stated in the relational algebra or calculus. In section 3.2 a brief overview is given and in section 3.5 a comparison is made between several of the distributed query processing algorithms. In this chapter we will discuss the optimization for two types of basic operations, on the one hand

relational operations on fragments, and, on the other hand operations on storage structures. In subsection 3.1.7 the merits of both optimizations are discussed.

3.1.4. Execution of the Schedule

In the last phase the schedule is executed. To ensure that none of the data that are accessed by the query or update are changed by another update, a concurrency control mechanism is used.

The **distributor** will disentangle the schedule and determine the duties of the involved sites and transmit them. It will also communicate with the concurrency control mechanism to get a shared or exclusive "lock" on the required data. When the sites receive their duties they add **synchronization** and **forking points.** The synchronization points are needed to let an operation wait until its inputs or part of them are available. The distributor of a site will monitor the synchronization points of that site. The forking points are needed if the result of an operation is needed as input by several other operations. An example of a forking point is the notification of the duties by the distributor (see fig. 3.2 (b)). Between forking and synchronization points parts of the schedule may be processed in parallel. These parts will be called **parallel schedules.**

The description of distributed query processing we gave, is called **static,** because a processing schedule is determined prior to execution. To do this, intermediate results are estimated to evaluate the cost of schedules. During execution of the schedule it may happen that intermediate results are much larger than expected. The schedule is, however, fixed and, although changing the schedule would be advantageous, processing must proceed along the dictated lines. Another interesting approach is to determine a schedule during processing, so it can be based on the correct sizes of intermediate results. This is a **dynamic** approach. In this chapter, however, we will confine ourselves to the static approach.

3.1.5. Response Time of a Query

What exactly do we mean by response time? After the query has been parsed and a suitable processing schedule has been decided on, the involved sites are notified of their duties. These duties may be to send data to another site or to wait for incoming data and to perform some operation on them and to further transmit the result. The elapsed time between the first starting operation (i.e., after the first forking point) until the desired result has been presented to the user at the result site (last synchronization point) will be called the **response time.** To compute the response time of a schedule a model for processing operations and transmitting data is required. For both an operation and a transmission we assume that all the input data on which it operates should be locally available, before the operation or transmission is put in the queue of its respective server. This may seem rather restrictive; for example, the merge-join in System R [Selinger1979] only requires part of the input data before its execution may start. Such operations can be split into smaller operations of which each will wait for only a fraction of the input data.

So, the response time of a schedule at a synchronization point is determined by the last "arriving" input. Hence, an obvious way of defining the response time at a synchronization point is to say that it is equal to the maximum of the response times of the different parallel schedules ending at that synchronization point **(Parallelism**

Assumption). This is something to be careful about because of the following two reasons.

First of all, the transmission time of a package is an expectation, which means that the actual time maybe smaller or larger than this expectation. Taking the maximum of two expectations is of course not the same as the expected maximum of the response times of the two packages. This would only be true if the distributions of both response times do not overlap.

Secondly, during processing, two parallel schedules ending at a synchronization point might have "shared" a resource such as a CPU or a communication channel. By "sharing" we mean that for example, one package is already waiting in a queue to be transmitted and then the second package enters the same queue. In that case the second package will always have to wait longer in the queue than the first one, of course under the assumption that nothing else has significant influence on the queueing times.

The difference between the expected and the observed response times has been investigated by computing the response time of schedules for queries in two simple systems. In the experiment roughly 32,000 queries were executed. The response times of the first thousand schedules were discarded, to avoid the effects of the empty system at the beginning.

To study the difference between the maximum of the two expected response times and the observed maximum of two response times, we need two independent servers, say S_1 and S_2. These servers stand for one-way communication channels from site A to C and B to C. The schedule for a query consists of the integration of two parallel schedules. So, after these parallel schedules there is a synchronization point. One parallel schedule contains the transmission of data from site A to C and the other from B to C. The interarrival time between queries is drawn from an exponential distribution with expectation equal to λ. The service time (transmission cost) is drawn from a negative exponential distribution with expectation equal to $1/\mu_i$ for server S_i $(i = 1, 2)$.

The average number of queries entering the system per unit of time (λ) is set to 1. μ_1 is set to 2 and μ_2 will vary from 2 to 6. This means that the expected response transmission time of the parallel schedule in which server S_1 is used, is equal to $1/(\mu_1 - \lambda) = 1$, and the other one less than or equal to 1. Hence, the maximum of the two expectations is 1. This value will be compared with the observed response transmission time for varying μ_2.

μ_2	2	3	4	5	6
RTT	1.43	1.14	1.06	1.02	1.01

Table 3.3. Average response transmission times of schedules.

For large values of μ_2 the response transmission time of the integrated schedule is almost completely determined by the response transmission time of the parallel schedule in which S_1 is used, and is therefore almost equal to 1. For smaller values

of μ_2 the response transmission time will be determined by either parallel schedule, resulting in a value larger than the maximum of the two expectations.

In general, we expect that the distributions of the response transmission times of the parallel schedules at synchronization points hardly overlap because of the wide range of RTTs of the parallel schedules. Therefore, we make the assumption that the effect of this phenomenon can be neglected. However, we have to keep this in mind when discussing the validity of the obtained results.

To investigate the effect of resource sharing on the response time we require a slightly more complicated queueing system. There are three servers, S_1, S_2 and S_3, that stand for one-way communication channels. S_1 is a channel from A to C, S_2 from B to C and S_3 from C to D. This system is used to simulate a schedule that is the integration of two parallel schedules. In one of these parallel schedules some local processing is done at site A, the obtained result is transmitted to site C, where possibly some more local processing is done, and, finally, the result is sent to site D, the result site. In the other parallel schedule the same is done only processing starts at site B, visits C, and ends at D. So, although these schedules can be processed in parallel they share S_3, the communication channel from C to D.

The inputs of this queueing system are two transmissions, one for S_1 and one for S_2. The input distribution is an exponential function with λ as the expected number of queries stated per unit of time. Because nothing is known about the amount of data to be transmitted, the service time has been drawn from a negative exponential distribution with μ_i as the expected number of transmissions served per unit of time for S_i ($i = 1, 2, 3$). The observed response transmission time of the integrated schedule is equal to the largest of the response transmission times of the parallel schedules. So again, after these parallel schedules, there is a synchronization point.

The influence of the speed of S_2 (μ_2) on the RTT of the query is investigated. Both μ_1 and μ_3 are set to 2. In addition to the transmissions of the queries there are other transmissions as well, such that the average number of transmissions entering the queue of each server is equal to 1.

Under the assumption that the parallel schedules can be executed fully in parallel and that they do not influence each other's response transmission time, we can compute the RTT of both parallel schedules independently. The RTT of the one that uses S_1 equals $1/(\mu_1 - 1) + 1/(\mu_3 - 1) = 1 + 1 = 2$, and the RTT of the other one equals $1/(\mu_2 - 1) + 1/(\mu_3 - 1) \leqslant 2$ if $\mu_2 \geqslant 2$. Hence, the maximum of the RTTs equals 2.

μ_2	2	3	4	5	6
RTT	2.79	2.48	2.41	2.40	2.39

Table 3.4. RTT for varying μ_2, and $\lambda = 0.1$.

The average RTT was observed for $\lambda = 0.1$ and for varying μ_2 (see table 3.4). λ stands for the frequency with which the query is stated. The difference between the observed and expected RTT can partly be explained by the additional delay of the

second entering transmission in S_3's queue, on top of its normal queueing time, caused by the not yet completed transmission of the first one.

Another reason is that because of the forking and synchronization points the **Independence Theorem** of Jackson [Jackson1957] can not straightforwardly be applied to determine the queueing delays of the servers.

So, our conclusion is that parallel schedules that share a mutual resource can not be treated independently. In the next subsection a tool is proposed for computing the "real" response time of a schedule and for determining the order in which the jobs are served.

3.1.6. Serializing Parallel Schedules Using the Same Resource

In the previous subsection a problem met in minimizing the response time of a query was encountered. A processing schedule will, in general, consist of the integration of many parallel schedules. Some of these parallel schedules may somewhere, during their processing, share a mutual resource, such as a CPU or a communication channel. In this subsection we will propose a tool for the optimizer that can be used to compute the response time of a schedule that has been computed under the Parallelism Assumption and to determine the order in which jobs that share the same resource have to be served. The proposed tool is explained in the context of data transmissions.

Let us go into detail now. In two parallel schedules two packages, say V and W, are transmitted from S_x to S_y. In the response transmission time of these parallel schedules under the Parallelism Assumption their transmission times, $TT_{xy}(V)$ and $TT_{xy}(W)$, are accounted for. If V is the first package entering the queue for the channel and its transmission has been completed before W enters the queue there is no problem. However, if V is still waiting in the queue when W enters it, W will on the average have a longer queueing time than V, because it has to wait for the completion of the transmission of V as well.

Serializing parallel schedules that use the same communication channel means ordering the packages that enter the queue and adapt their expected time in the system if other packages of the same schedule are ahead of them and still in the system. If we write $TT(V) = T_0 + TC(V)$, where T_0 is the expected queueing time and $TC(V)$ is the time to transmit V over the channel, then the transmission time of W is

$$TT(W) = T_0 + TC(V) + TC(W),$$

where $T_0 + TC(V)$ is the queueing time of W. There may be other packages as well in the system, but their effect on the queueing time of V and W can expressed by T_0, because they enter the queue independently of V and W.

Note that serializing may imply changing the order in which the packages are served. The reason we may want to change this order is that for example, the response transmission time of the query discussed above is determined by the parallel schedule in which W is transmitted from S_x to S_y. So, adapting the transmission time of W because V is ahead of W and still in the queue, will probably increase the response transmission time of the integrated schedule. In that case it might be better to postpone the moment that V enters the queue until W enters it. This will increase the transmission time of V and, therefore, the response transmission time of the

parallel schedule in which V participates, but does not necessarily increase the response transmission time of the integrated schedule.

The problem of serializing the parallel schedules of an integrated schedule such that the response time is minimized, is known as **precedence constraint scheduling** with the restriction that jobs have to be executed by specific computers. In [Garey1979] this problem is listed under the known NP-complete problems [Garey1979, Aho1974].

Now we will discuss some ways of serializing an integrated schedule that consists of parallel schedules. The different strategies for serialization will be explained by simulating the schedules. The time a package enters a queue can be computed from the schedule. First, we look at just one channel from S_x to S_y.

A straightforward way of serializing a schedule is to keep the order of transmissions the same as the order in which they enter the queue. If two packages enter the queue at exactly the same time they can be ordered arbitrarily. This strategy favors the one that has a lead. Something like: it makes wealthy people richer and poor people poorer. This strategy would be all right if we were only interested in the first arriving package by our previous definition. However, the response transmission time of an integrated schedule is determined by the last arriving package. Therefore, we will propose another strategy.

Again we consider the packages in the order in which they enter the system. We will say that a queue is **schedule-empty** if no other package that participates in the schedule of the query, is present in the queue. At the time a package enters the queue it might be schedule-empty. In that case we just put it in the queue. In general, say P_{n+1} is the newly arriving package and $P_1, P_2, \ldots, P_n$ are the ones waiting in the queue. P_{n+1} is put at the end of the queue. The response transmission times of the following alternatives are compared. There are $n + 1$ alternatives one for each $j = 1, 2, \ldots, n$, and one which makes no changes. Each alternative takes all P_i $i = j, j+1, \ldots, n$ from the queue and put them right behind P_{n+1}. This will change both the transmission time of P_{n+1} as well as the ones of the P_i's that are moved. The final effect on the response transmission time of the integrated schedule is computed for each alternative and the one with minimum response transmission time is chosen.

Now all channels used in the processing schedule are to be serialized and the effect of different orders on the response transmission time of the integrated schedule is computed. We will travel in time; we go from left to right on the time scale of all the involved sites. Every time a package enters one of the queues we consider the alternatives discussed above. The response transmission time of the query is computed based on the partial serialization of the parallel schedules that has already been decided on and for other parallel schedules the Parallelism Assumption is used. Based on this response transmission time the alternatives are compared and the one with minimum response transmission time is chosen.

In subsection 3.3.7 a comparison will be made between the solutions obtained by the heuristic approach described above for serializing schedules and the optimal serializations.

How effective changing the order of transmissions is, will much depend on the underlying distributed operating system. If the decision about the routing of the transmissions is made by the system, changing the order may become useless, and

hence making the minimization of the response time extremely difficult. However, in computer networks where routing is fixed we expect it to be useful. For example, in some networks, such as the ETHERNET [Metcalfe1976], we might even have to serialize all the transmissions in a query to compute its response time.

3.1.7. Basic Operations

The available information in the data dictionaries to parse a query and to compute a processing schedule will determine how detailed such a schedule is. For the computation, it may contain the locations and the sizes of relations, the sizes and the selectivity of attributes, whether indices on certain attributes are available, etc.. For two kinds of distributed databases we will discuss the **basic operations** of the schedules produced by their respective optimizers.

Consider a distributed database that merely consists of a collection of centralized databases, and assume also that the only available information in the globally accessible data dictionaries consists of the parsing information and the locations of these databases. The unit of allocation is thus a complete database. Because sizes of intermediate results can not be estimated, the query processing algorithm will probably decompose the query into subqueries that can be processed at each of the sites independently. The results are then gathered at the result site, where the final processing is done. So, these subqueries are the basic operations. Every local database management system will determine an efficient way for processing them. A comparison between different decompositions is not possible because no information is available to compare the cost of schedules.

Another kind of distributed database is the one whose unit of allocation is a fragment of a relation, and whose globally accessible data dictionaries contain all information necessary to determine efficient schedules. Just like before, the basic operations can be relational operations on the fragments, but now their cost can be determined from the available information. But they can even be low level operations that make use of (secondary) indices. Also, if the fragments are stored redundantly, the query processing algorithm can decide which copy to access, based, for example, on the indices on it.

We have seen two extremes of distributed databases and their information about logical and physical structures that is available to the query processing algorithm. There are of course many variations between these two. The advantage of having complete information is that optimization can be done at a high level (decomposition into subqueries) as well as at a low level (which index and copy to use). In the ideal case a query stated in the relational calculus is translated into operations on the storage structures, and that further optimization can be done using this translated query without the loss of optimality caused by decisions taken at a higher level. In section 3.3 we will investigate this by using inverted lists as storage structure. A similar approach, namely bypassing the high level optimization, is discussed in section 3.4; only, there relational operations are allowed as basic operations. Also, a more or less low level operation, called the semi-join, is used as a means to decrease the cost of transmitting fragments. Finally, in section 3.5 the integration of the use of semi-joins with a decomposition process is investigated, based on existing algorithms.

3.1.8. Summary

In this section the three phases of query processing, namely parsing, determining a schedule and executing it, were discussed. Minimizing the response time of a query was treated in detail. Because the problem of ordering jobs in parallel schedules that compete for the same resource is NP-complete a heuristic approach for minimizing response time is adopted. First, a schedule is determined under the Parallelism Assumption, and then it is serialized such that parallelism that can not be achieved, is removed. By serializing a schedule the order in which operations have to be executed or in which data have to be transmitted is fixed.

The basic operations in a schedule will depend on the data available to the optimizer. In one case the basic operations are subqueries in the relational data model, which are to be processed locally. Then each site involved must determine a local schedule for its subquery. The advantage is that site autonomy can be realized. In the other case the optimizer has detailed knowledge about the data stored at other sites as well, and incorporates this knowledge in the schedules. The advantage is that more efficient schedules can be obtained.

To conclude this section we summarize in table 3.5 the assumptions discussed in this section; it also includes a few that were not mentioned explicitly.

At each site a distributed database management system is available, which can execute all the required operations.

All sites can communicate with each other, either directly or indirectly.

Transmission Assumption: transmitting data is an order of magnitude more expensive than local processing.

Equal Transmission Cost Model: the time to transmit a particular amount of data between any pair of sites is the same.

Arbitrary Transmission Cost Model: no assumptions are made about the time required to transmit data from one site to another.

Local Processing Assumption: the cost to process data locally is proportional to the amount of data.

Parallelism Assumption: the response time at a synchronization point in a schedule is equal to the maximum of the response times of the different parallel schedules ending at that synchronization point.

Table 3.5. List of assumptions.

3.2. Overview of Distributed Query Processing Algorithms

Since the late seventies all over the world a lot of research is going on in the area of distributed databases and especially in the area of distributed query processing. We may well say that E. Wong made a start with his paper [Wong1977] and its influence on current research is still noticeable. Most researchers have developed their own model of query processing which makes a qualitative and, even more, a quantitative comparison difficult. In [Apers1981c] an attempt has been made to compare most

of the models and algorithms. Such a model is characterized by its assumptions about the way the query is stated, the data allocation, whether a materialization is determined before or during optimization, which cost function is minimized, how much information is available to the optimizer, whether estimators are used, whether the schedules are determined statically or dynamically, etc..

A crude classification can be given based on (see also [Hevner1981]):

- **materialization**
  Some algorithms assume that the materialization is determined before the computation of the schedule (Wong, Hevner and Yao, Epstein et al., Chu and Hurley, Toth et al., and Nguyen Gia Toan). The advantage of this is that not all copies of all relations have to be consistent. Only a particular choice of the copies for a query (materialization) must give a consistent view of the database. The disadvantage is that this choice is not necessarily optimal for query processing. To overcome this problem other algorithms let the optimizer decide about the materialization (Baldissera et al., In-Sup Paik and Delobel, Pelagatti and Schreiber, Selinger et al.).
- **schedule**
  Almost all algorithms determine the schedules before executing them. This approach is called **static,** and the cost of the schedules produced is computed based on estimates of intermediate results. Only two algorithm compute the schedules during processing (Epstein et al. and Nguyen Gia Toan) of which one is integrated with the concurrency control mechanism to synchronize decisions about the schedule (Nguyen Gia Toan); this approach is called **dynamic.**
- **computation of the join**
  The size of the result of a join can be small but can also be as large as the product of the sizes of the operands. If a join is computed at a non-result site, eventually, the result will have to be transmitted to the result site. Some algorithms let arbitrary sites compute joins (Epstein et al., Chu and Hurley, Pelagatti and Schreiber, Nguyen Gia Toan, Selinger et al.), others do so only if it concerns joins that produce small results (Baldissera et al., and Toth et al.). The remaining algorithms only let the result site compute the joins (Wong, and Hevner and Yao).

In this section we will briefly discuss the models and algorithms in turn. A quantitative comparison between the algorithms can be found in section 3.5.

Wong

Wong's algorithm has been developed for the distributed DBMS SDD-1 [Rothnie1977a], which runs on the ARPANET [McQuillan1977]. Because of this, the total transmission cost was taken as cost function. Although other costs, such as processing time, are not excluded explicitly, the original paper [Wong1977] does not consider them. Redundant data allocation is not allowed; also the optimizer expects a predetermined materialization. The result of a query is produced at an arbitrary site in the network.

Before computing the schedule, as much local processing as possible is done. With this we mean the computation of restrictions and projections. The initial schedule consists of the transmissions of all the reduced fragments to the largest one.

This set of transmissions will be denoted by M. At the site that receives all the fragments, the result site, the joins between them are computed to obtain the final result. The algorithm now tries to replace M by two sets M_1 and M_2, that are executed sequentially with local processing between them. What these M_i's look like will be discussed in a moment. Consider for M_1 and M_2 transmission sets such that their sequential processing produces the same result as M. Take the pair with the minimum total transmission time. After that, apply the algorithm recursively to both M_1 and M_2. If no replacement has a smaller total transmission time nothing changes. The schedule is represented by a tree. The nodes stand for local processing and the leaves for the transmissions. The replacement of M by M_1 and M_2 means that the leaf M is substituted by a subtree consisting of M_1 as left leaf, a node for the local processing and M_2 as right leaf. The processing tree is executed inorder.

Now we discuss the way M_1 and M_2 are obtained. M consists of transmissions of reduced relations. A way to further reduce a relation in size is to apply semi-joins to it on its joining attributes. M_1 contains all transmissions to compute **semi-joins** on fragments that are transmitted in M, and that cost less than the reduction in cost caused by the reduced size of the fragments (cost-effective). The set M_2 then consists of the transmissions of the further reduced fragments and the transmissions in M for which no cost-effective transmissions could be found.

The algorithm is **greedy** [Horowitz1978]; that means that it locally tries to minimize the cost function, here the total transmission cost, as much as possible.

Hevner and Yao

The work of Hevner and Yao [Hevner1979a, Hevner1979b] is based on the same fundamental idea as Wong's algorithm: apply **semi-joins** to the relations to reduce them in size, transmit them in their reduced form to the result site where the joins between them are computed. These authors developed a quantitative model to compare the cost of schedules. This model makes it possible to estimate the effect of the application of semi-joins on the sizes of the relations. This effect, called **selectivity**, is defined as a number between 0 and 1 and denotes the portion of the relation that remains. For a special class of queries, called **simple queries**, they presented an algorithm that produces optimal response transmission time schedules and one that produces optimal total transmission time schedules.

We explain their model in more detail in section 3.4, because our own work is partly based on it.

Epstein, Stonebraker and Wong

This research was done in the context of the distributed INGRES [Stonebraker1977]. The query processing algorithm will minimize a weighted function of the total transmission cost and total processing cost, given a predetermined non-redundant materialization. It can handle fragmentation. It does not necessarily need an estimating technique.

The query processing algorithm for centralized INGRES [Stonebraker1976] is based on **decomposing** the query into irreducible components that overlap on one variable (relation) only [Wong1976]. Fig. 3.6 shows an arbitrary query and its decomposition. The result of component C_1 forms a restriction on the tuples of relation R_1,

and, therefore, fewer tuples are used in further processing. The same is true for C_2 but now on the tuples of R_2. So, after both C_1 and C_2 are processed C_3 will have two reduced relations, resulting in cheaper processing of C_3.

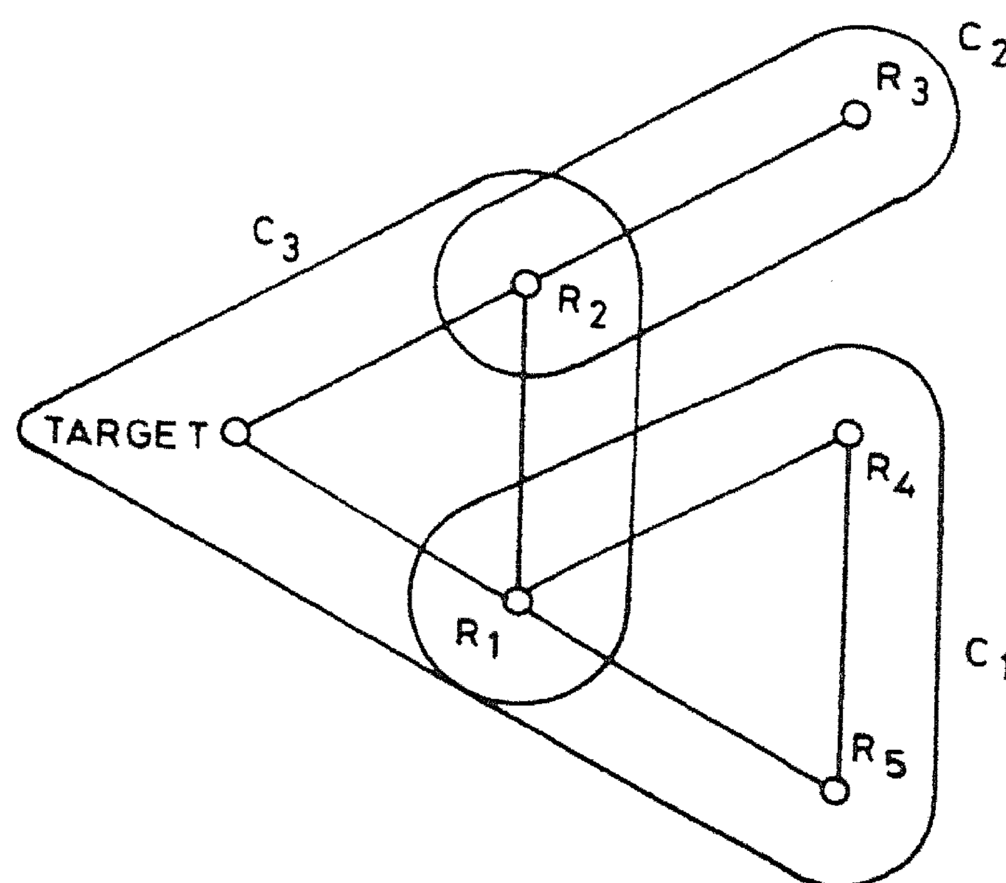


Figure 3.6. Decomposition into irreducible components.

These components are substituted by components containing fewer relations by **tuple substitution**; This substitution continues until only one-variable queries remain. And these can be processed in a straightforward way.

A similar approach [Epstein1979] was taken in the distributed version of INGRES. Again the query is decomposed into irreducible components. Tuple substitution is abandoned and replaced by a more general technique. Based on some heuristic a **piece** is taken from the component. Such a piece is a subgraph of the component and contains two or more relations and the joins between them. How these joins are processed depends on the fragmentation and the type of network. The resulting relation will be substituted in the component in place of the processed piece. Again a piece of the remaining component is taken until nothing is left over. If a component is processed it is substituted in the query by the resulting relation.

A form of backtracking is used which makes it possible to throw away intermediate results that are unexpectedly large and to make a different decision about what to do next. So, in a sense it may be called **dynamic**.

When minimizing total transmission cost for a site-to-site network the joins in a piece are processed at the site where the largest fragment of the largest relation in that piece resides. This implies that all other relations and the other fragments of the largest relation must be gathered at that site. For a broadcast network the largest relation may remain fragmented and at each of the sites containing such a fragment part of the joins are computed. For the site-to-site network only in one special case the largest relation is split, namely if the network consists of two sites and there is only one other relation in the piece to be processed.

To get rid of the heuristic flavor of the algorithm described above exhaustive search is used [Epstein1980a, Epstein 1980b]. The decomposition of the query is

removed and a search for the piece to be processed next is started right away. This piece may range from two to all relations in the query. The use of a perfect estimator ensures that this algorithm finds an optimal solution within the capabilities of INGRES simply because it investigates every solution.

Baldissera, Bracchi and Ceri

The distributed query processing algorithm presented in [Baldissera1979] can handle only tree structured queries. This means that the only kind of joins that are allowed, are **semi-joins**. The optimizer minimizes total transmission cost taking full advantage of the redundancy. To compare the cost of the schedules, the sizes of the relations and the selectivities of attributes have to be known.

Although not explicitly stated in their article, the proposed algorithm belongs to the family that uses semi-joins. The target list may contain only attributes from one relation and the graphical representation of a query is a tree, which consists of combinations of two basic substructures, the **branch structure** and the **vertical (linear) structure**. Both are shown in fig. 3.7.

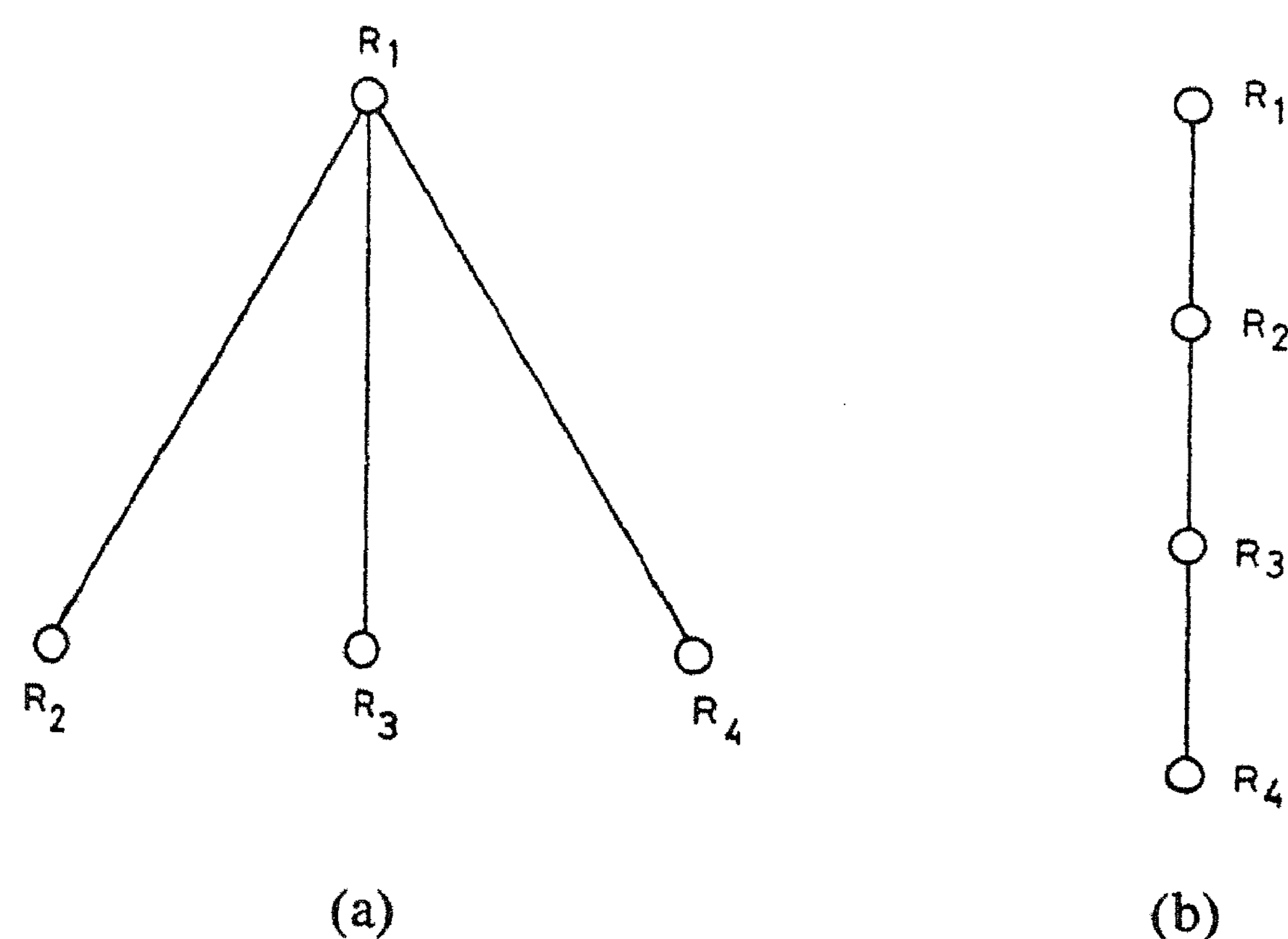


Figure 3.7. The branch and vertical structure.

The complete algorithm consists of three modules. The first one computes the selectivities resulting from the restrictions on relations. This is done for all copies. Later, when it is known which copies are used, superfluous restrictions and projections are removed from the schedule. The second module, the heart of the algorithm, decides how to tackle the query. Finally, the third one tries to achieve as much **parallelism** as possible.

For the second module we first explain the branch structure (see fig. 3.7(a)). Let us call R_2, R_3 and R_4 the sons. These sons may form restrictions on the tuples of R_1 based on several attributes. If so, each attribute is handled in turn. Let us therefore assume that there is only one attribute in the branch structure. The basic idea is now to send the smallest son, say R_2, to another son. The best choice for the receiving son is the one that is smallest after the selectivity of R_2 has been applied. To

compute this the selectivity of R_2 is applied to all other sons and the smallest one is chosen as the receiving son. After that, R_2 is removed from the structure. This is repeated until a vertical structure remains.

In a vertical structure several attributes can be used (fig. 3.7(b)). For example, between R_2 and R_3 there may be a semi-join on attribute A and between R_3 and R_4 on B. Therefore, if R_3 is to be transmitted to R_4 its attribute values of both A and B have to be sent. If, however, R_3 is transmitted to R_4 only the attribute values of A are sent.

The initial schedule consists of the transmissions of all relations in a vertical structure to the site containing most of the data; call this site the central site. Then a contiguity analysis is applied to see whether this can be improved. This analysis starts with the lowest two relations in the vertical structure and computes whether sending one to another and then to the central site is cheaper than the initial schedule (this is a form of decomposition). If so, this schedule is adopted. Then it goes one level up in the vertical structure and the same analysis is done. The analysis continues all the way up.

In general, a query will be a combination of branch and vertical structures. If the structure is vertical the algorithm explained above is applied. Otherwise, search for the first branch structure. If a son is a root of subtrees, the algorithm is applied recursively to compute its size and selectivity. After this is known for all sons, the algorithm for branch structures is applied, leaving that part of the query as a vertical structure.

The interesting thing about the algorithm is that the query is decomposed into subqueries and that it uses the semi-join operation. Its main drawback, however, is that it can not handle all types of queries.

Chu and Hurley

The approach of Chu and Hurley consider **operation trees** [Chu1979]. Such a tree corresponds with a particular way of processing a query. Data reduction functions, which tell us what portion of a relation is left over after an operation, help the optimizer to decide where to allocate the operations in the trees. The cost function is a combination of total transmission cost and total processing cost. The optimizer assumes that it gets a predetermined materialization. No fragmentation is considered.

For queries that do not contain cycles, all possible operation trees are computed. Such a tree contains three types of nodes: a **file node** (β_i^0), which is a leaf of the tree and represents the restriction and projection on one relation on which the query operates, an **operation node** (β_i), which represents an operation that is computed between two relations, such as a join, and an **end node** (β_e), which represents a final operation, such as displaying the data on a terminal, and can only be executed at the result site. Operation nodes may contain more than one operations, meaning that all these operations are to be executed at one site.

The algorithm considers all possible operation trees. For every operation a data reduction function is known that computes the fraction that remains. The total transmission cost is computed by assigning the operation nodes to different sites (we call this operation allocation, see also chapter 4). The processing cost of a site is computed by determining the optimal order of the execution of operations.

To limit the search for the optimal schedule, each tree is put in a Sequence Group. Such a group contains all trees that execute the operations in a certain order. This order fully determines the total processing cost and, therefore, we only need to find the trees with minimum total transmission cost of each Sequence Group. Given these trees the total processing cost for each of them is computed, resulting in the complete optimal solution.

The use of data reduction functions is convenient in accurately estimating the result of an operation without making any assumptions. However, these functions, which are probably based on statistical data, may require a database of their own to be stored, if there is a large collection of operations. Another drawback is that the model does not allow for all types of queries. A tree with a fixed set of operations β_i, is not sufficient to represent queries containing cycles.

In-Sup Paik and Delobel

The contribution of In-Sup Paik and Delobel [Paik1979] is not so much concerned with the way a query is processed, but more with the choice of **materialization**. If the fragments are stored redundantly and all copies are identical, meaning that a query can access any of the copies, independently of the copy used for another fragment, a materialization is computed to minimize total transmission cost. Fragmentation of a relation is allowed and information about the fragmentation criterion, cardinalities, etc. are available in the data dictionary. Also, no assumption is made about the network topology, although a distance table should be available.

The materialization process consists of three phases, called clustering, filtering and centroiding.

In the **clustering phase** every possible materialization for a query is investigated. Each materialization consists of a duplicate for each relation. A **duplicate** is the collection of copies of fragments such that the complete relation can be reconstructed. A **cluster** corresponding to a materialization is the set of sites containing the data of the materialization. For each cluster C the value of the clustering function

$$f_C = \sum_{i,j \in C} distance(i,j)$$

is computed. The materialization with a cluster whose f_C has the smallest value is chosen. Intuitively, this means that the most concentrated one is used.

The **filtering phase** corresponds to what is normally called the initial local processing. To all fragments the restrictions and projections are applied. Some restrictions may have a high correlation with the fragmentation criterion leaving an empty result and making the corresponding copy in the cluster obsolete. So, after the filtering phase the cluster may have been reduced in size. The resulting query will only contain operations whose operands are located at different sites.

In the **centroiding phase** several alternatives for processing the query are distinguished. One alternative is to transmit all data to one central site, called **centroid**. Such a site is searched among all sites of the network and is not necessarily restricted to the cluster. The one that minimizes the total transmission cost is chosen. The other alternative is to use the centroid only to control the processing. How this is done is not discussed here. The controlling site is the one that is situated most central with respect to the sites of the cluster.

Clearly, the described algorithm should be followed by an algorithm that, given a materialization, will compute a processing schedule. As such it is useful especially when we realize that many query processing algorithms expect a predetermined materialization.

Pelagatti and Schreiber

The algorithm proposed by Pelagatti and Schreiber [Pelagatti1979] illustrates a different area in distributed query processing. It is not designed for ad hoc queries but more for queries that are stated quite frequently on a slowly changing database. For these queries it is important that they are executed efficiently and, therefore, much processing time may be invested in finding appropriate schedules. Pelagatti and Schreiber provide the system programmer with tools to determine such schedules with the aid of the computer.

The design consists of three levels: Logical Strategy, Distribution Strategy and Transmission Execution Strategy. The first one merely translates a relational calculus query to a set of operation trees (**Logical Strategy**). Each of these trees is equivalent to the original query, they only describe how the result is obtained. The next two levels are entered for each such tree. At the second level the site where the operations are executed, is determined. Because fragmentation is allowed, a closer look is taken at the relational operations to lay down the conditions on the allocation before the execution of an operation. For a restriction no limitations are imposed on the allocation. And the allocation of the result is the same, except that some of the fragments may be empty. The projection operation is split into two operations. To each of the fragments a local projection, called distributed projection, is applied and the results are gathered at one site where again a projection, called a global projection, is computed. This second projection is necessary to delete duplicates that occurred in different fragments. One of the operands of a join has to be completely duplicated at each of the sites where fragments of the other operand reside. The allocation of the result is the same as the relation that stayed distributed. For other operations, such as aggregates, similar conditions can be determined. If relations do not satisfy these conditions, transmission operations are included in the operation tree.

To evaluate the cost of a **Distribution Strategy**, a tool is required to estimate intermediate results. One problem involved with fragmentation is the **correlation** between a clause in a query and the fragmentation criterion of a relation. Pelagatti and Schreiber propose methods to test for correlation and its consequence for the allocation of a result. The third level is entered for each Distribution Strategy. Such a strategy can not directly be translated to a **Transmission Strategy** (schedule) because of redundancy. Also the cost function to be minimized is more complicated than in other models. An example cost function is: minimize response transmission time subject to the minimization of total transmission cost. All the problems at this level are translated to integer programming problems, for which optimal solutions can be found using standard integer programming techniques.

A general technique has been developed by Pelagatti and Schreiber which searches for a schedule in a large space and is therefore useful for **precompiled** queries, that are used often.

Toth, Mahmoud and Riordon

In his thesis [Toth1980] Toth developed an algorithm for query processing in the ADD system [Toth1978]. This algorithm expects a predetermined materialization. Fragmentation of relations can easily be incorporated. The cost function to be minimized is total network traffic subject to a specified response transmission time constraint.

The query is expressed in relational algebra. A user query will look like

$$r \leftarrow ((R_1 \times R_2 \times \cdots \times R_m) \mid C)[Z],$$

where $\times$ stands for the Cartesian product, the C for a restriction on the tuples and the Z for a projection. The R_i's are the relations of a distributed database. Both the C and the Z can be transformed such that operations on one relation result, which are the subqueries. Every site will have its own subquery. They can all be processed in parallel. The results of the subqueries can be combined in any order to produce the desired result. Estimators are used to compute the expected sizes of the subqueries. In [Toth1980] an extensive study on this can be found.

Determining the order in which the results of the subqueries are taken together is done heuristically based on a special type of query, called **Class A query**. For this Class A query an optimal solution can be found for minimizing total network traffic subject to a specified response transmission time constraint (expressed in the number of hops). Intuitively, one can imagine such a query as having no joins, between subqueries, whose result is no larger than its largest operand. Obviously, a query consisting of only semi-joins (simple queries, see [Hevner1979b] and subsection 3.4.1, and the tree structured queries of Baldissera et al. [Baldissera1979]) belongs to this Class A.

Based on the result sizes of the subqueries all Class A queries are identified. One such Class A query is selected and an optimal schedule for it is computed. The first time this is done, the last transmission in the schedule goes to the result site. For later computed schedules the destination may be any site from which a path of transmissions already goes to the result site. This iteration, searching for Class A queries and computing a schedule for it, continues until no Class A queries remain. Subqueries that have not been considered yet will transmit their result to the result site.

Although the user query is stated in relational algebra, he has no influence on the order in which the operations are executed. The applicability of the algorithm depends on the resulting sizes of the join operations. If they are small many Class A queries can be found for which an optimal schedule can be computed, resulting in near optimal schedules for the user queries.

Selinger et al.

The research discussed in this paragraph is part of the System R* project of IBM [Williams1981]. In [Selinger1980] the extension of the Access Path Selector of System R [Selinger1979] is explained. As in the centralized version an N-way join is replaced by a sequence of 2-way joins. For each such join the location, the inner and outer relation, and the join method is determined. The join methods considered are the **nested loop join**, which goes through the entire outer relation and retrieves the

matching tuples from the inner relation, or the **merge join**, which goes through both relations in order simultaneously. The problem of replicated and redundant data is solved by extending the search tree such that a choice is made among non-redundant materializations of non-partitioned relations. Sizes of intermediate results are estimated by using statistical data on the selectivity of predicates. The cost function contains terms for the communication, for the disk accesses and the usage of the CPU. In [Selinger1980] it is shown that especially the disk accesses may not be neglected if a high bandwidth communication channel is used.

In System R queries are **compiled** to achieve better performance at execution time. The same approach is adopted in System R* [Daniels1982, Ng1982]. In [Daniels1982] first a global plan is determined by a master site using the above described method. This plan is then given to the sites involved, which, to ensure site autonomy, may discard the plan because it was based on outdated catalog data. If the plan is accepted the site identifies its duties from it and determines a local plan, which is stored locally in compiled form. Part of the global plan is the way the local plans communicate with each other.

Access paths provided by the database management system or privileges to access a given relation may change over time making a compiled plan invalid. In [Ng1982] the problem of automatic invalidation and recompilation is discussed. Furthermore, it is show that not always a global recompilation is necessary to maintain the optimality of the plan. If, for example, one of the indices is dropped it may be replaced by another one, requiring only local recompilation. Especially, the case of the 2-way join is treated extensively.

Nguyen Gia Toan

Most of the algorithms discussed so far determine a processing schedule prior to executing it. These schedules are called static because they are fixed. The optimizer will estimate intermediate results to compare the cost of the different schedules. Although, during execution time, results may be larger than expected, the processing continues as dictated by the schedule. Quite a different approach is proposed by Nguyen Gia Toan [Nguyen Gia Toan1979, Nguyen Gia Toan1980]. The processing schedule is constructed during execution, and is, therefore, called **dynamic.** Decisions about this construction are made in a **decentralized** way. Decisions taken by the sites are synchronized by a concurrency control mechanism. There a **token** travels around the network along a virtual ring [LeLann1978]. At any time only one site can have the token and is allowed to make a decision.

The query is represented by an **operation tree** that operates on a predetermined materialization. The goal is to determine where each of the operations is executed. The site where the query originates will search for maximum local subtrees. These subtrees will contain operations, that can be executed at one site, and operations whose operands already exist. This means that the sizes of the operands of these operations are known. Such operations are restrictions and projections, but may also be joins between existing relations, which are not necessarily located at the same site.

The operation tree, including the localized subtrees, is now broadcast to all sites. Each site processes its subtree. From now on decisions about data transfers are made in a decentralized way with the aid of a **threshold.** What the initial value is of this threshold and how it is updated depends on the cost function to be minimized and will not be discussed. With it, a data transfer of a not yet computed result may

be ordered. For example, a join has to be computed between intermediate results IR_1 and IR_2. IR_1 is computed by S_1 and IR_2 by S_2. A few situations will be discussed; in [Apers1981c] a complete table can be found covering all situations.

The computation of IR_2 is finished and S_1 is notified of that. If IR_2 is smaller than the threshold, then S_1 can, after it has obtained the token, decide to transfer IR_2 to S_1 if the computation of IR_1 has not finished yet or if IR_1 is larger than IR_2. Only if IR_1 is smaller than IR_2, S_1 decides to transfer IR_1 to S_2. After such a decision the threshold is updated; this updated threshold together with the decision is broadcast through the network. All other situations in which a decision can be taken require that both IR_1 and IR_2's computation has finished. Then the smallest one is transferred.

The fact that a site requires the token when it makes a decision ensures that no conflicting decisions can be made. Every decision is notified to other sites giving them up-to-date information about the way the query is processed.

The application of this dynamic algorithm, which takes decisions in a decentralized way, highly depends on the concurrency control mechanism with which it is integrated, and the way the threshold is updated. It seems a promising direction and a comparison with static algorithms in a real distributed data base will be interesting.

From the above overview of query processing algorithms in distributed databases we may conclude that query processing in a distributed database has been given much attention. Striking is the diversity of models that are investigated. Almost every model has its own repertoire of operations, and its own restrictions about the data in the data dictionary concerning relations. Besides that, some of the research is concerned with particular aspects, such as, for example, the choice of materialization, compilation and recompilation of schedules, and dynamically determining schedules.

Most of the research, except [Selinger1980], addresses only the problem of global optimization. The schedules produced consist of transmissions and subqueries to be processed at the sites involved; the way the data are stored is not take into account. Also, minimizing response time received little attention (except [Hevner1979a, Hevner1979b]).

In sections 3.3 and 3.4 minimizing response time and response transmission time will be discussed in the context of the inverted file organization and the relational calculus, respectively. Minimization of the response time is done by first considering only the transmissions, and then including the local operations. For minimizing total transmission time two approaches are considered. On the one hand, a query is split into subqueries for which, independent of each other, schedules are determined, and the schedules are integrated to obtain a schedule for the query. On the other hand, as few parallel schedules as possible are constructed, which are used in processing more than one subquery.

3.3. Query Processing with Inverted File Organization

The goal of query processing is to determine a schedule for a query stated by a user. Such a schedule consists of data transmissions and operations that have to be executed at the sites involved. Under the Transmission Assumption the transmissions will mainly determine the cost of a schedule. Therefore, query optimization is split

into two phases. First the optimizer considers only data transmissions. The resulting schedule is called a **macro-schedule**. Besides the fact that it contains only transmissions, it also fixes the tasks of the sites involved. If local processing cost can be neglected completely compared to transmission cost, this schedule will suffice. Each site involved will be notified of its duties, and the way they are handled has no influence on the macro-schedule and can, therefore, be determined locally. In section 3.4 we will just determine macro-schedules.

If local processing cost can not be neglected completely or if the optimizer has to compile the query [Daniels1982, Ng1982] local processing should be included in the schedule as well. A schedule for the processing done at one site is called a **micro-schedule**. After the macro-schedule has been determined these micro-schedules may be computed by each of the sites involved, and transmitted back to the site responsible for the overall optimization. Or, if sufficient information is available in the data dictionary, all the micro-schedules may be determined by the site that computed the macro-schedule. If response time is minimized this integrated schedules is serialized. The latter is still necessary to determine the order in which data is transmitted and in which operations are executed at each of the sites.

In this section we will discuss these ideas in the context of the inverted file organization [Apers1978]. Considering query processing as manipulating lists is attractive in the context of a distributed database. Compared to tree structured indices, not much processing is required to reconstruct the inverted lists after transmission. Although the ideas are explained for inverted lists, they are equally applicable to other storage structures.

3.3.1. Inverted Lists as Storage Structure and Unit of Allocation

The inverted file organization is useful for answering queries of the type

$$A \ relop \ V,$$

where A is some attribute, *relop* is an element of the set $\{=,<,\leq,>,\geq,\neq\}$ and V is a value of the domain of attribute A. In [Hsiao1970] we can find a formal description of this file organization. The use of the inverted file organization has been extensively studied for a centralized system. Both in [Cardenas1975] and in [Yao1977] an analysis can be found of the average access time, and in [Hill1978] this analysis also includes updates such as insertion and deletion. How to process a query that consists of a Boolean expression of index terms of the simple form above is presented in [Liu1976]. There, the total processing cost required to merge the inverted lists corresponding to the index terms in the Boolean expression is minimized.

Before discussing the use of an **inverted file organization** in a distributed database we will briefly describe it in a centralized system. Assume we have a set of elements. Each element has a key, which is a unique number, and the value of an attribute. The keys will be inverted on the value of this attribute. This means that for each value in the domain of the attribute we make a list of the keys of the elements, whose attribute equals that value. The pointers to the beginning of each of these **lists**, short for **inverted lists**, are put in a **directory**, that contains an entry for each value in the domain; the keys in a list are ordered.

So far, we only discussed inverted lists on one attribute, say A. Accessing the elements in the set on another attribute would only be possible by a sequential scan.

If the keys are inverted on all attributes it is called a **completely inverted system.** Then a directory exists for each attribute and a key will be included in a list in every directory.

In general, a query is a Boolean expression of conditions that have to be satisfied by the attributes, expressed by means of relational operations. Such an expression can be straightforwardly translated to set operations on subsets, represented by the lists. If the condition is expressed with the $=$ operation it is replaced by the subset of the corresponding list; for the $\neq$ operation the complement of the subset with respect to the complete set is taken. The **complement** of a list A is denoted by A'. For the other relational operations the union of the subsets of the elements that satisfy the condition can be taken. Finally, the **and** and **or** operations are replaced by the $\cap$ and $\cup$ operations, respectively.

Example 3.1

Assume that a university has a database about documents. This database will contain the following relations:

DESCRIPTION (*bookno* ,*title* ,*year* ,*publisher* ,*location*)
AUTHORS (*bookno* ,*author*)
KEYWORDS (*bookno* ,*keyword*).

Both on year and publisher secondary indices are created by means of inverted lists for relation *DESCRIPTION*. *AUTHORS* is inverted on author and *KEYWORDS* on keyword. A book may have several keywords. The notation *keyword* $=$ topic is short for: topic is one of the keywords; *keyword* $\neq$ topic means that none of the keywords is topic. A Boolean expression of index terms like:

(*keyword* $=$ database) **and** (*keyword* $=$ distributed) **and**
(*year* $\geq$ 1981) **and** (*publisher* $=$ North Holland)

is translated into set operations on lists:

$$A \cap B' \cap (C \cup D) \cap E$$

where

A is the list of *keyword* database
B is the list of *keyword* distributed
C is the list of *year* is 1981
D is the list of *year* is 1982
E is the list of *publisher* is North Holland.

□

The operations $A \cap B, A \cup B$ and $A \cap B'$ can be computed by merging the lists corresponding to the index terms A and B. For the last operation all elements of A are taken except those in B. Therefore, a B' is only allowed if it is intersected with another, not complemented list. The cost of merging two lists is proportional to the sum of their sizes. This means that the cost to compute $A \cap B$ is $P_1(|A| + |B|)$, where $|A|$ denotes the size of list A, and P_1 is the proportionality constant. To compare the cost of different processing schedules we need to know the

sizes of the intermediate results. Let us define the Prob($x \in A$) = $|A| / |X|$, where X is the complete set. Then the Prob($x \in A \cap B$) = $|A \cap B| / |X|$. Under the assumption that there is independence between lists, we may say

$$\text{Prob}(x \in A \cap B) = \text{Prob}(x \in A) \times \text{Prob}(x \in B) = |A|\,|B| / |X|^2.$$

So, $|A \cap B| = |A|\,|B| / |X|$. Along the same lines $|A \cup B|$ and $|A \cap B'|$ can be determined. Summarizing

1) $|A \cap B| = |A|\,|B| / |X|$,
2) $|A \cup B| = |A| + |B| - |A|\,|B| / |X|$, and
3) $|A \cap B'| = |A| - |A|\,|B| / |X|$.

An **operation tree** for a query is a tree, consisting of operations as internal nodes and lists as leaves, such that the result of the operations in the tree is the same as the result of the query. A schedule for an operation tree tells us at which sites the operations are computed, and thereby fixing the transmissions. A serialized schedule also dictates in which order operations and transmissions are executed.

Now we will go distributed. We assume that the lists are the **units of allocation**. To obtain an efficient data allocation, the lists of all directories may be assigned to one or more sites. The elements of the set themselves may, based on the access pattern, also be allocated according to some distribution criterion. Information concerning the allocation of the lists and the elements is stored in the data dictionary. This information is used to translate the user query to an expression of lists stored at the different sites. The result of a processing schedule is a list of keys satisfying the user query. Given these keys we still have to retrieve the attributes of the elements in which the user is interested.

Example 3.2

Assume that the database of our previous example is distributed over a network. Relations may be partitioned and more than one copy may be stored in the network. The lists are not necessarily stored at the same sites as the fragments of the relations; depending on how frequently they are accessed, possibly in combination with other lists, by the users they may be stored redundantly. The following user query is stated at site 1: give the booknumbers, locations and the year of publication of the books satisfying

(*keyword* = database) **and** (*keyword* = social impact) **and**
((*location* = Computer Science Dept.) **or** (*location* = History Dept.))

and the result is to be delivered to site 1. Let us assume that the relation *DESCRIPTION* is distributed over the network. For example, tuples of books physically located at the Computer Science Dept. are stored at the site of the same department. Furthermore, we assume that a materialization for the lists is determined before optimization:

site 1: the list of *keyword* = database
site 2: the list of *keyword* = social impact
site 3: the list of *location* = Computer Science Dept.
site 4: the list of *location* = History Dept.

The original query is replaced by two queries of which the results are united at the result site (S_1):

(*keyword* = database) **and** (*keyword* = social impact) **and**
(*location* = Computer Science Dept.)

(*keyword* = database) **and** (*keyword* = social impact) **and**
(*location* = History Dept.).

The schedule for these queries is shown in fig. 3.8. To take full advantage of the fact that the queries look very much alike the processing starts with the transmission of the list *keyword* = database to S_2 (this transmission is denoted by T_1). At S_2 the intersection is computed with the list *keyword* = social impact (denoted by P_1). So far, the schedules of both queries are the same. Because the lists concerning the locations are stored at different sites, S_3 and S_4, the result of the intersection computed at S_2 is sent to both of them in parallel (denoted by T_2 and T_3). At S_3 this intermediate result is intersected with the list *location* = Computer Science Dept. (denoted by P_2) and at S_3 with the list *location* = History Dept. (denoted by P_3). So, at both S_3 and S_4 we have a list of booknumbers satisfying the original query of which the tuples are stored at that same site. The relevant data transmitted from S_3 and S_4 to S_1 (denoted by T_4 and T_5). Finally, at S_1 the union of the data from S_3 and S_4 is presented to the user (denoted by P_4).

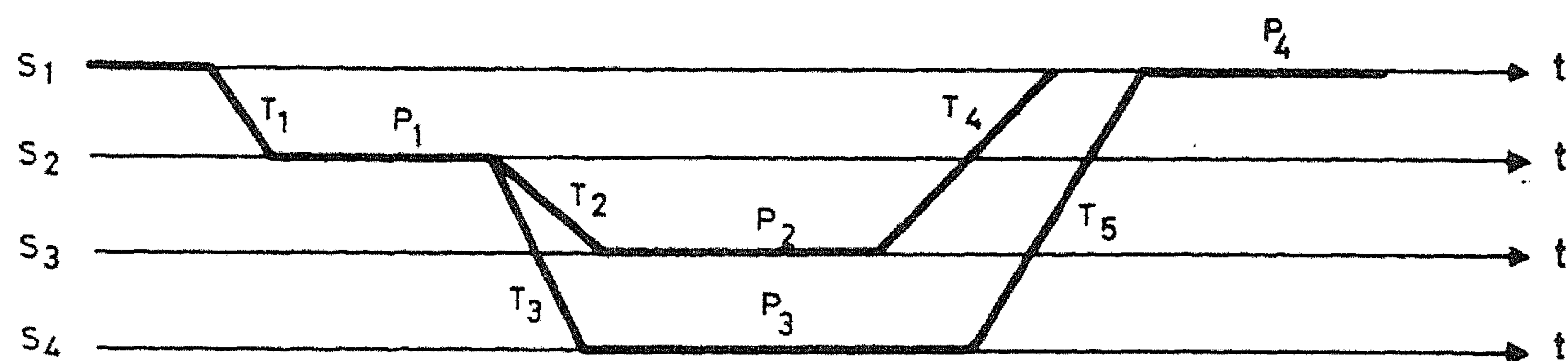


Figure 3.8. Schedule for example query.

□

The example shows only one of the many schedules that are possible for this query. In the next subsections we will give algorithms to process lists in a distributed environment, such that either the response time or the total time is minimized.

3.3.2. Minimizing Response Transmission Time

Based on the assumption that the response time of a query is mainly determined by the transmissions involved (Transmission Assumption), we will first minimize the response transmission time and then for each site minimize the response processing time.

In the following we assume that we can define the response transmission time of the integrated schedule as the maximum response transmission time of the parallel schedules. Based on this, an algorithm is presented that produces minimum response transmission time schedules. We will then change such a schedule such that no two lists ever share the same resource without paying for it. Experimental results are given in subsection 3.3.8. We expect that this way of producing schedules, first computing them under the Parallelism Assumption and then serializing them, is more convenient than considering the minimization of the real response time as one big problem. The optimizer would then have to deal with query decomposition and the use of resources all at once.

To formalize the above we give the next proposition.

Proposition Under the Parallelism Assumption every schedule, given in the form of a graph, can be converted to a tree with no forking points after an operation or transmission, with the same response transmission time.

Proof Assume we have a schedule with forking points. Let O_0 be the first operation or transmission of one of the sites with a forking point after it; the result of O_0 is input of $O_1, O_2, \ldots, O_n$. Then, for each O_i, $i = 2, 3, \ldots, n$ a copy is made of all operations and transmissions that are required to produce the same result as the result of O_0, which is now used as input of O_i. The connections in the schedule between O_0 and O_i ($i = 2, 3, \ldots, n$) are removed. Hence, the forking point after O_0 is removed. Furthermore, no new forking points are introduced, because it was the first forking point. Also, the response time of this new schedule is the same as the one of the original schedule under the Parallelism Assumption, because only copies are made of existing operations and transmissions.

This process can be continued until no forking points are left in the schedule, and the response time of the newly constructed schedule is the same as the response time of the original schedule.

□

Every operation in a schedule that is a tree with no forking points after an operation is a synchronization point. So, the schedules for the operands of an operation are parallel schedules. The operation subtrees of the parallel schedules are treated as a set of operation trees, with a synchronization point at the site where the operation on the subtrees is computed. Let OT be such a set of operation trees and the site with the synchronization point is S_i. The response transmission time of OT at site S_i is defined as

$$RTT_i(OT) = \max_{T \in OT} RTT_i(T).$$

3.3.3. Disjunctive Normal Form

In this subsection we will give an algorithm, which produces schedules for minimizing response transmission time. Normally, the optimality of the solutions produced by an algorithm is proven after the algorithm has been presented. Here, however, we will first give some theorems that limit the number of expression trees to be considered for processing a query. Before proving them we will summarize their results. In Theorem 3.1 and Corollary 3.2 we transform an arbitrary operation tree by applying the distributive law on the set operations, such that all union operations are at the top of the tree. Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption (see below) it can be shown that this transformation does not increase the response transmission time of the operation tree. In Theorem 3.3 and Corollary 3.4 it is shown that, under these same assumptions, executing all union operations at the result site, does not lead to a higher response transmission time. The results will be clarified by an example. To start with, we will introduce some notation.

A **literal** is either an inverted list or the complement of an inverted list. The **conjunction** of $Q_1, Q_2, ..., Q_n$ is

$$Q_1 \cap Q_2 \cap \cdots \cap Q_n,$$

where the Q_i's are literals. A query Q is said to be in **disjunctive normal form** if and only if Q has the form

$$Q = Q_1 \cup Q_2 \cup \cdots \cup Q_n,$$

where Q_i is a conjunction of literals. A Q_i will be called a **term.** Throughout this chapter we will assume that the size of the intersection of two lists A and B, $|A|\,|B|\,/\,|X|$, is neglectably small compared to the sizes of both A and B (**Intersection Assumption**).

Theorem 3.1 Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model applying the distributive law to an operation tree of a query will not increase its response transmission time.

Proof Assume that somewhere in the operation tree of a query Q we have the expression

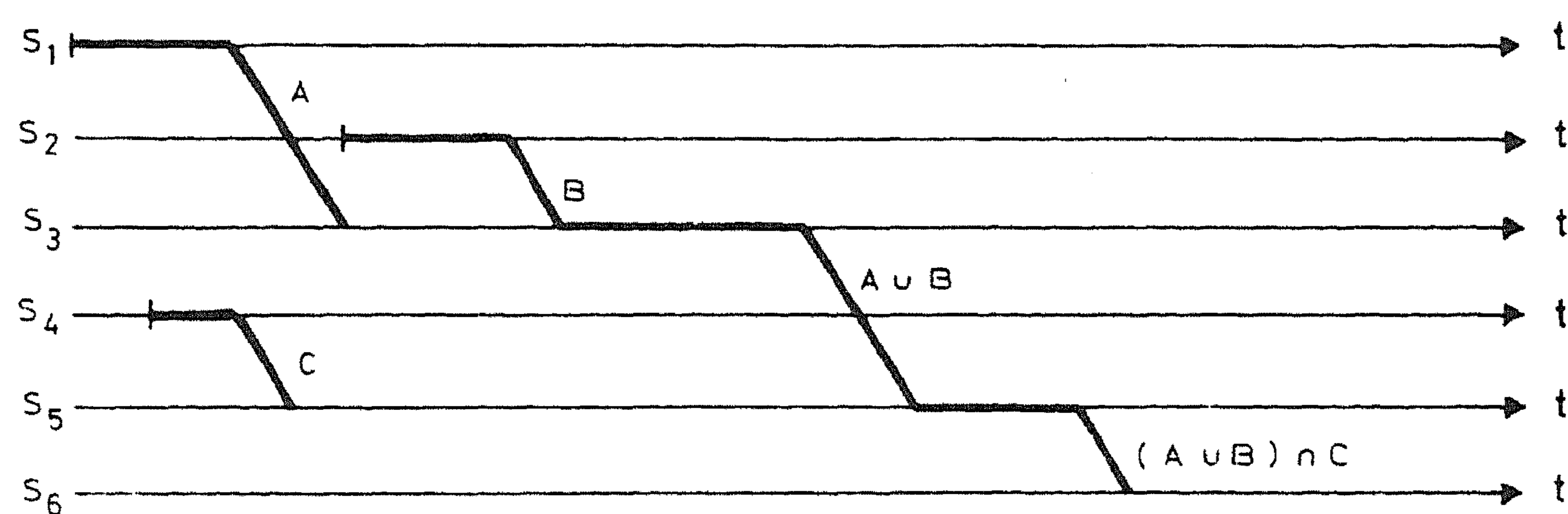
$$(A \cup B) \cap C,$$

where C may be an intermediate result or its complement. What we have to do is to change this to

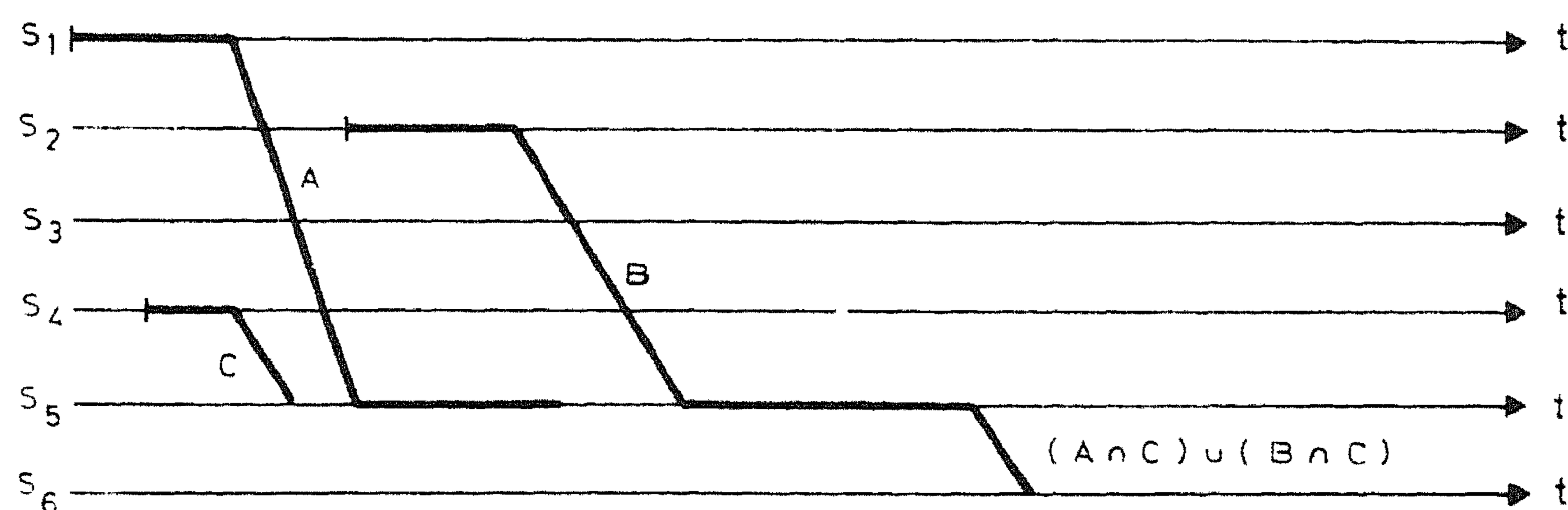
$$(A \cap C) \cup (B \cap C),$$

and prove that it does not increase the response transmission time of Q. It suffices to show that the response transmission time of the subtree does not increase (proposition

of subsection 3.3.2). In fig. 3.9(a) an arbitrary schedule is depicted; the sites 1 to 6 are not necessarily different. This means that some of the transmissions may have zero time, depending on whether the source and destination site are the same.



(a)



(b)

Figure 3.9. Schedule for (a) $(A \cup B) \cap C$ and (b) $(A \cap C) \cup (B \cap C)$.

The response transmission time of this processing schedule is:

$$RTT_6((A \cup B) \cap C) = \max(RTT_3(\{A, B\}) + TT_{35}(A \cup B), RTT_5(C)) + TT_{56}((A \cup B) \cap C).$$

We now use the same sites as in the above schedule to process the operation tree

$(A \cap C) \cup (B \cap C)$. Its schedule is shown in fig. 3.9(b) and the response transmission time is:

$$RTT_6((A \cap C) \cup (B \cap C)) = \max(\max(RTT_5(A), RTT_5(B)), RTT_5(C)) + TT_{56}((A \cap C) \cup (B \cap C)).$$

Because $RTT_5(A) \leq RTT_3(A) + TT_{35}(A)$, we can say

$$RTT_6((A \cap C) \cup (B \cap C)) \leq \max(\max(RTT_3(A) + TT_{35}(A), RTT_3(B) + TT_{35}(B)), RTT_5(C)) + TT_{56}((A \cup B) \cap C).$$

Letting A and B wait for each other at site 3 and transmitting their union is more expensive than transmitting them in parallel to site 5, because $|A \cup B|$ is larger than $|A|$ and $|B|$. So, the right hand side satisfies

$$\leq \max(RTT_3(\{A,B\}) + TT_{35}(A \cup B), RTT_5(C)) + TT_{56}((A \cup B) \cap C) = RTT_6((A \cup B) \cap C).$$

Hence, applying the distributive law to an operation tree does not increase its response transmission time.

□

Corollary 3.2 Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model an operation tree with optimal response transmission time for a query can be replaced by an equivalent tree in disjunctive normal form without sacrificing its optimality.

The top of an operation tree that is in disjunctive normal form consists of union operation(s). Now the site where to compute them will be discussed.

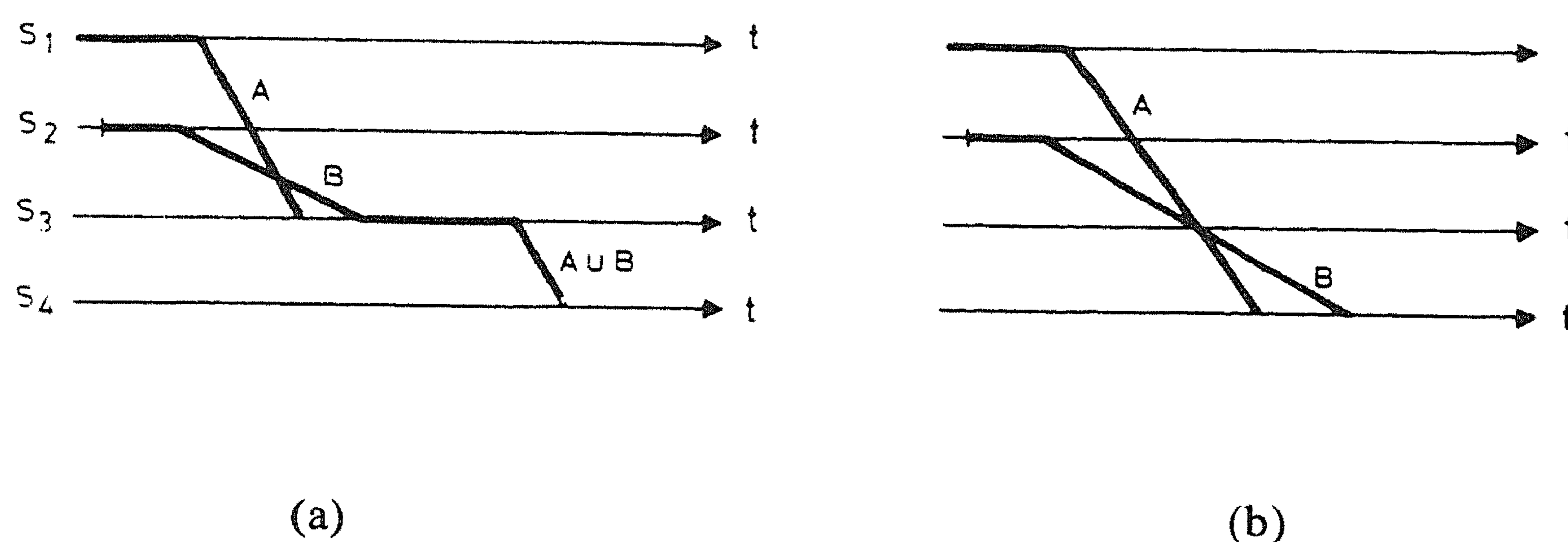
Theorem 3.3 Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model the union operations of an operation tree in disjunctive normal form can be executed at the result site without increasing its response transmission time.

Proof We will prove this by induction on the number of union operations in the tree. Assume that there is only one union operation. In that case the operation tree looks like

$$A \cup B,$$

where A and B may be intermediate results. The schedule for computing the union operation outside the result site is shown in fig. 3.10(a). Again the sites are not necessarily different. The response transmission time of this schedule is:

$$RTT_4(A \cup B) = RTT_3(\{A,B\}) + TT_{34}(A \cup B).$$



(a) (b)

Figure 3.10. Schedule for (a) $A \cup B$ and (b) $\{A, B\}$.

Transmitting both A and B in parallel to the result site, where the union is computed, gives us the schedule shown in fig. 3.10(b). The response transmission time of this schedule is:

$$\begin{aligned} RTT_4(\{A,B\}) &= \max(RTT_4(A), RTT_4(B)) \\ &\leqslant \max(RTT_3(A) + TT_{34}(A), RTT_3(B) + TT_{34}(B)) \\ &\leqslant RTT_3(\{A,B\}) + \max(TT_{34}(A), TT_{34}(B)) \\ &\leqslant RTT_3(\{A,B\}) + TT_{34}(A \cup B) \\ &= RTT_4(A \cup B). \end{aligned}$$

Assume that the induction hypothesis is true if the operation tree contains m union operations. Now consider an operation tree that contains $m + 1$ union operations. Then again it will look like $A \cup B$, where A and B are subtrees which together contain m union operations. The proof that the union between A and B can be computed at the result site without increasing the response transmission time, is exactly the same as for the case that their is only one union. Hence, all the union operations in A and B can be computed at the result site because of the induction hypothesis.

□

Corollary 3.4 Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model an operation tree with optimal response transmission time for a query can be replaced by an equivalent tree in disjunctive normal form and all union operations can be computed at the result site without sacrificing its optimality.

Based on this last corollary we can immediately give an algorithm to compute the minimum response transmission time of a query Q; see fig. 3.11. This algorithm requires the computation of the minimum response transmission time of a term, for which an algorithm will be given later.

```
proc MRTT query=(query Q,site S_r)schedule:
begin
   schedule sch := empty schedule;
   put Q in disjunctive normal form;
               {say, Q_1 ∪ Q_2 ∪ ··· ∪ Q_d }
   for t to d
   do
      sch := integrate(sch,MRTT term(Q_t,S_r))
   od;
   sch
end
```

Figure 3.11. Algorithm *MRTT query*.

Example 3.3

Throughout section 3.3 we will use the same example. At site S_1 the query

$$(A \cap B) \cap (C \cap D' \cup E')$$

is stated. At site S_2 the lists A (1000) and E (700) are located, at site S_3 D (2000), at site S_4 B (400) and site S_5 C (200). The sizes of the lists are given in parentheses. Part of the construction of the minimum RTT schedule is discussed in this example and will be continued in example 3.5.

From Corollary 3.2 we know that we can replace the example query by its disjunctive normal form:

$$(A \cap B \cap C \cap D') \cup (A \cap B \cap E')$$

without increasing its response transmission time. From corollary 3.4 we know that the union can be computed by S_1.

□

3.3.4. Breaking a Term

In the previous subsection we discussed the rewriting of an expression tree into its disjunctive normal form and where to compute the union operations. From this and the definition of the response time at a synchronization point we know that our next goal is the minimization of the response transmission times of the terms in the disjunctive normal form of a query. This subsection deals with breaking a term into smaller expressions that consists of the intersection of at most three A-lists and at most one B-list.

Theorem 3.5 Let Q_t be a term of the form

$$A_1 \cap A_2 \cap \cdots \cap A_n \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m .$$

Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model we only have to consider the minimum response transmission time of the following set S of seven types of operation trees:

$$A_i$$
$$A_i \cap A_j$$
$$A_i \cap A_j \cap A_k$$
$$B_p$$
$$A_i \cap B'_p$$
$$A_i \cap A_j \cap B'_p$$
$$A_i \cap A_j \cap A_k \cap B'_p$$

for $i, j, k = 1, 2, \ldots, n$ (i, j and k have different values) and $p = 1, 2, \ldots, m$, to compute the minimum response transmission time of Q_t. Let E be a subset of the set S then

$$RTT_r(Q_t) = \min_{E \subset S} (\max_{e \in E} RTT_r(e))$$

where the operation trees in E must include all A_i's and B_p's of Q_t.

For the sake of convenience we will call A_i ($i = 1, 2, \ldots, n$) an A-list and B_p ($p = 1, 2, \ldots, m$) a B-list. To prove Theorem 3.5 we must prove a series of lemmas. The first lemma provides a technique for replacing a query with m complemented lists by m queries with only one complemented list, without increasing the response transmission time. Once that is proven we will go on to prove lemma 3.7, which shows how the intersection of an arbitrary number of A-lists can be replaced by a set of intersections of at most three A-lists, again without increasing the response transmission time

Lemma 3.6 Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model an operation tree for

$$A \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m,$$

where $A = A_1 \cap A_2 \cap \cdots \cap A_n$ can be replaced by the set of operation trees

$$\{A \cap B'_1, A \cap B'_2, \ldots, A \cap B'_m\}$$

whose results are directly sent to the result site, where the intersections are computed, without increasing the response transmission time.

Proof We will prove this by induction on m. For $m = 1$ the set of operation trees consists of the original operation tree. Let us assume that the lemma is true for values less than or equal to m. Now consider the operation tree for

$$A \cap B'_1 \cap B'_2 \cap \cdots \cap B'_{m+1}.$$

We search for an intersection node such that the operation subtree rooted at that node contains all the B-lists and that both operand operation subtrees contain at least

one B-list. Such a node can be found by starting off at the root node and by asking how many B-lists are contained in both operand subtrees and going down that one that contains all the B-lists, until both operand subtrees contain at least one B-list. Assume that the intersection of the root of the subtree is computed at site S_x. The operation tree is shown in fig. 3.12.

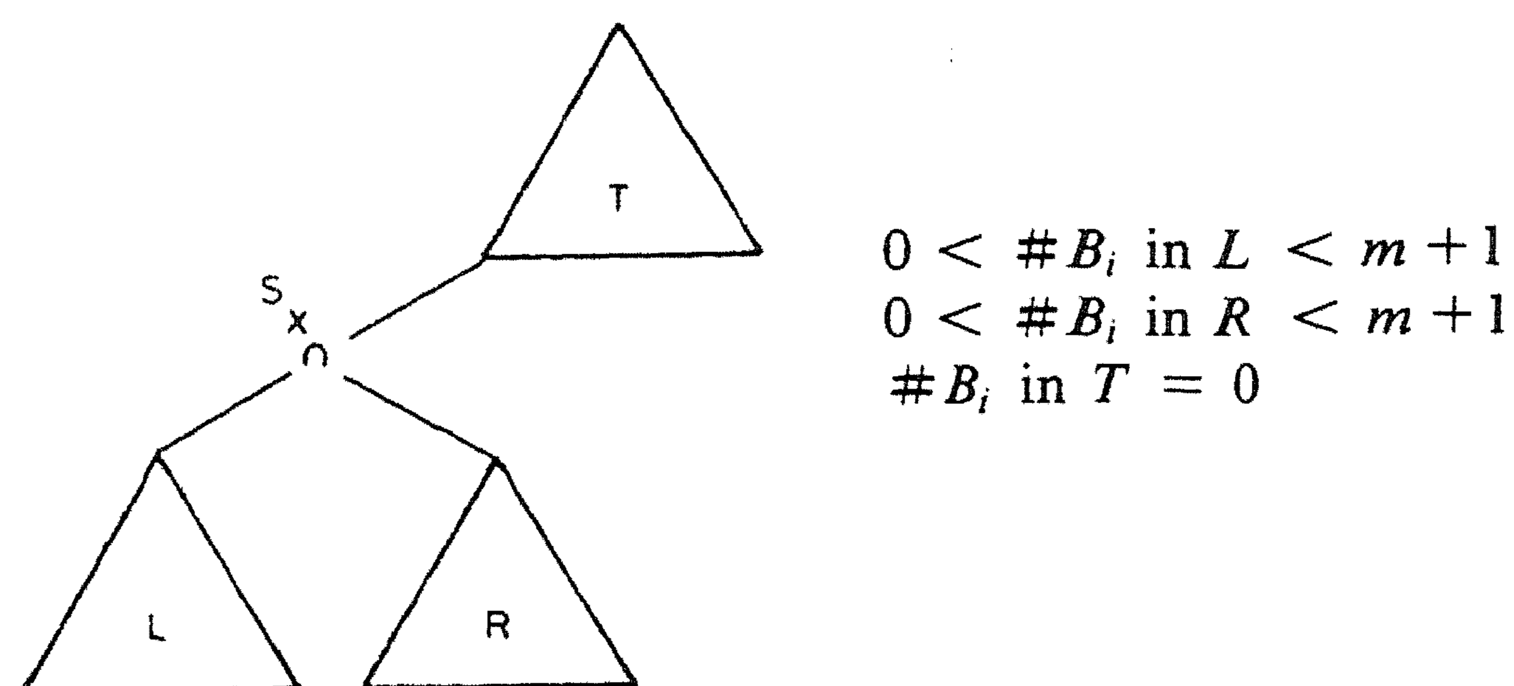


Figure 3.12. L and R contain all B-lists.

Let L^* (R^*) be the operation subtree L (R) with all B-lists removed together with all operations of which they are an operand. Because of the assumption about the resulting size after an intersection with a complemented list we know that the size of the result of L is neglectably smaller than that of L^*. Therefore, we regard them as being of the same size. Certain transmissions in the schedule for L^* (R^*) can be changed compared to the one for $L(R)$ because certain operations are deleted, and therefore,

$$RTT_x(L^*) \leq RTT_x(L)$$
$$RTT_x(R^*) \leq RTT_x(R).$$

Replacing $L \cap R$ by the expression $(L \cap R^*) \cap (L^* \cap R)$ where the top intersection is also computed at S_x will give the same result and does not increase the response time at S_x. Since,

$$\begin{aligned} RTT_x((L \cap R) \cap (L^* \cap R^*)) &= RTT_x(\{L \cap R^*, L^* \cap R\}) \\ &= RTT_x(\{L, R^*, L^*, R\}) \\ &= RTT_x(\{L, R\}) \\ &= RTT_x(L \cap R). \end{aligned}$$

The intersection of $L \cap R^*$ and $L^* \cap R$ is useless as far as reducing the size is concerned, because both $L \cap R^*$ and $L^* \cap R$ contain exactly the same A_i's. Therefore, this intersection will be computed at the result site. Hence, the original operation tree is replaced by two operation trees of which the result is intersected at the result site. Fig. 3.13 shows this. The response transmission time did not increase because the response transmission time of the subtrees rooted at S_x did not.

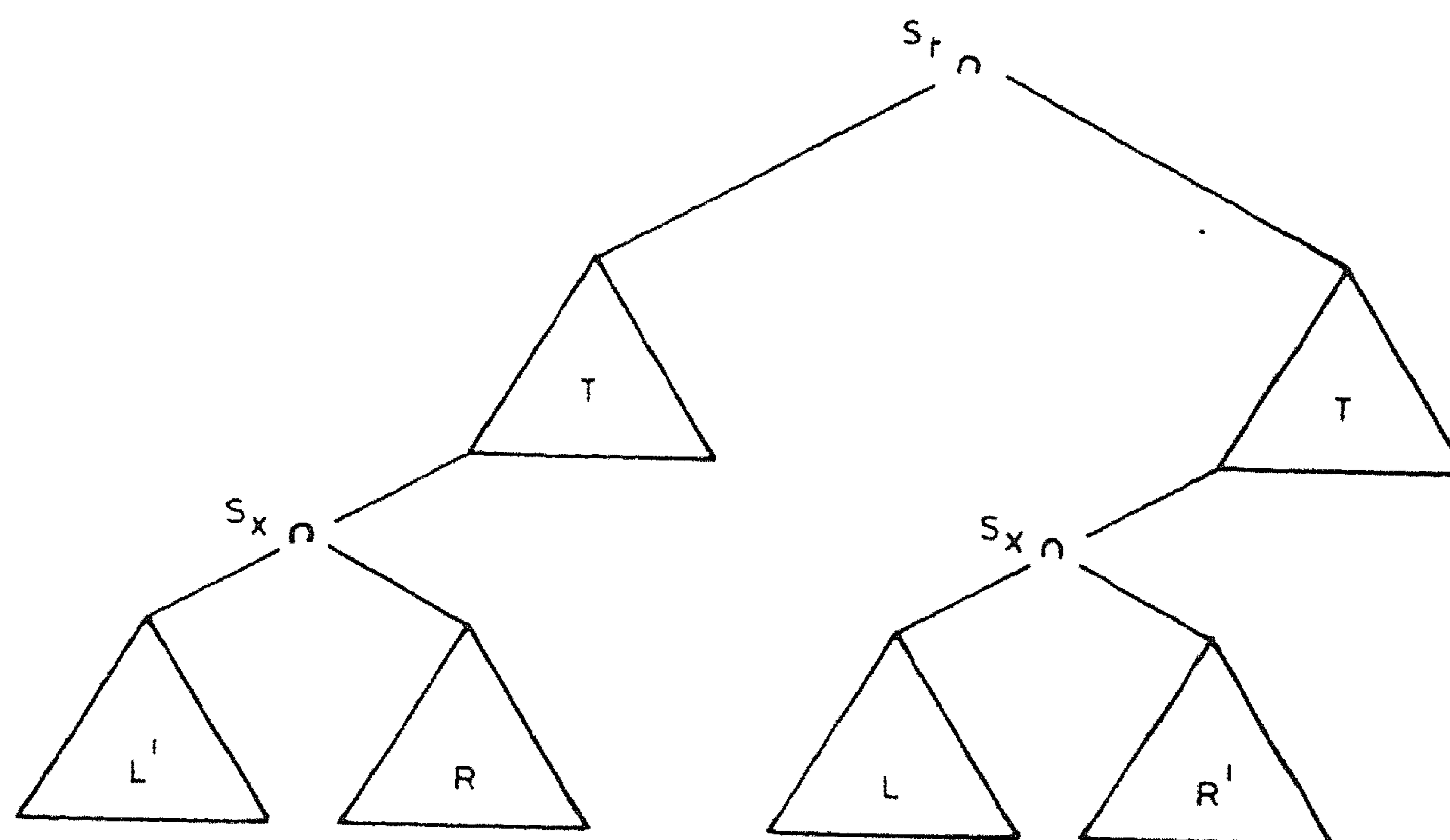


Figure 3.13. Intersection computed at result site.

Now we can apply the induction hypothesis to the two operand operation trees of the intersection at the root of the tree.

□

Lemma 3.7 An operation tree for $A = A_1 \cap A_2 \cap \cdots \cap A_n$ can be replaced by a set of operation trees of the form

A_i
$A_i \cap A_j$
$A_i \cap A_j \cap A_k$

whose results are directly sent to the result site, where the intersections are computed, without increasing the response transmission time.

Proof We will prove this by induction on n. For $n = 1$, 2 or 3 the set of operation trees consists of the original operation tree. Assume the lemma is true for all values less than or equal to n. Let us now consider the operation tree for an expression containing $n + 1$ A-lists. Such a tree can be written as $L \cap R$. Two cases have to be distinguished:

1) Both L and R contain at least two A-lists.
Say, L and R are intersected at S_x, and the result is sent to the result site, S_r. An alternative is to send both L and R to the result site and let it compute the intersection. The response transmission time of this schedule is no larger than the response transmission time of the original one:

$$\begin{aligned} RTT_r(\{L,R\}) &= \max(RTT_x(L) + TT_{xr}(L), RTT_x(R) + TT_{xr}(R)) \\ &\leq RTT_x(\{L,R\}) + \max(TT_{xr}(L), TT_{xr}(R)) \\ &= RTT_x(\{L,R\}) + TT_{xr}(L \cap R). \end{aligned}$$

Because both L and R are results of intersections we know that the results of

L, R and $L \cap R$ all have a size neglectably small compared to the A- and B-lists, and therefore the last equality holds.

2) Either L or R must contain exactly one A-list, say R does.
Because the total operation tree contains at least 4 A-lists we know that L contains at least 3 A-lists. Therefore, we will write L as $L^{*} \cap L^{**}$, where the intersection of L^{*} and L^{**} is computed at S_x and the one of L and R at S_y. The original operation tree will be replaced by an operation tree whose root node is an intersection computed at the result site with operands $L^{*} \cap L^{**}$ and $L^{*} \cap R$. The intersection of L^{*} and L^{**} is computed by S_x and that of L^{*} and R by S_y. The schedule of this new operation tree will have a response transmission time which is not larger than the original one:

$$\begin{aligned} RTT_r(\{L^{*} \cap L^{**}, L^{*} \cap R\}) \leq\ & \max(RTT_x(\{L^{*}, L^{**}\}) + TT_{xr}(L^{*} \cap L^{**}), \\ & \quad RTT_y(\{L^{*}, R\}) + TT_{xr}(L^{*} \cap R)) \\ \leq\ & \max(RTT_x(\{L^{*}, L^{**}\})) \\ & \quad + TT_{xy}(L^{*} \cap L^{**}) + TT_{yr}(L^{*} \cap L^{**}), \end{aligned}$$

$$\begin{aligned} & RTT_y(\{L^{*}, R\}) + TT_{yr}(L^{*} \cap R)) \\ & \quad \leq \max(RTT_x(\{L^{*}, L^{**}\}) + TT_{xy}(L^{*} \cap L^{**}), \\ & \qquad RTT_y(\{L^{*}, R\})) \\ & \qquad + \max(TT_{yr}(L^{*} \cap L^{**}), TT_{yr}(L^{*} \cap R)) \\ & \quad = RTT_y(\{L^{*} \cap L^{**}, R\}) + TT_{yr}((L^{*} \cap L^{**}) \cap R) \\ & \quad = RTT_r((L^{*} \cap L^{**}) \cap R). \end{aligned}$$

Because both $L^{*} \cap L^{**}$ and $L^{*} \cap R$ are results of intersections, their sizes are equal to the sizes of $(L^{*} \cap L^{**}) \cap (L^{*} \cap R) = (L^{*} \cap L^{**}) \cap R$, and therefore the last but one equality holds.

So, in both cases the original operation tree is replaced by two operation trees of which the intersection is computed at the result site, without increasing the response transmission time. And these new operation trees do not contain more than n A-lists, so the induction hypothesis can be applied to them and the lemma follows.

□

From the set of expressions derived in Lemma 3.6 and 3.7 for a term a subset is chosen such that all A- and B-lists in the term are used in one or more expressions and such that the maximum response transmission time is minimized. The results of the intersections are intersected at the result site, giving the result of the term. Before proving Theorem 3.5 we will illustrate this by an example.

Example 3.4

The term $A \cap B \cap E'$ of example 3.4 is broken down into the six expressions:

A, B,
$A \cap B$,
$A \cap E'$, $B \cap E'$,
$A \cap B \cap E'$.

Which of these expressions will be used in the schedule of the term depends on their response transmission times. In subsections 3.3.5 and 3.3.6 two models of transmission cost are discussed.

□

Proof of theorem 3.5 Assume we have an operation tree for processing

$$A_1 \cap A_2 \cap \cdots \cap A_n \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m$$

with minimum response transmission time. By applying lemma 3.6 we can replace this operation tree by a set of trees

$$\{A \cap B'_1, A \cap B'_2, \ldots, A \cap B'_m\},$$

where $A = A_1 \cap A_2 \cap \cdots \cap A_n$. The intersection of this set of operation trees is computed at the result site and its response transmission time is no larger than the original operation tree. Because some of the intersections $A \cap B'_i$ are already computed at the result site we can replace such a tree by $\{A, B_i\}$. Lemma 3.7 equally applies to $A \cap B_i$ as to A, therefore, each element of the above set can in its turn be replaced by the expressions mentioned in the theorem, again without increasing the response transmission time.

So, the original operation tree has been replaced by a set of operation trees without sacrificing its optimality.

□

3.3.5. Equal Transmission Cost Model

In theorem 3.5 only the operation trees have been discussed. Nothing has been said about the sites where the operations are computed, except for the result site, which will compute all union operations. Therefore, all possible schedules for an operation tree have to be investigated. The way this is done depends on the transmission times between sites. If the queueing times and the transmission cost functions are the same between every pair of sites, then the search for the minimum response transmission time schedule can be restricted. For intersections only the sites where one of the operands resides have to be considered. Pieces of terms obtained in the previous subsection containing three A-lists do not have to be considered.

Proposition Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Equal Transmission Cost Model an intersection in the minimum response transmission time schedule is computed either at the site where one of the operands is located or at the result site.

Lemma 3.8 Under the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and in the Equal Transmission Cost Model expressions $A_i \cap A_j \cap A_k$ and $A_i \cap A_j \cap A_k \cap B'_p$, where none of the intersections is computed at the result site, do not have to be considered.

Proof Assume that A_i, A_j and A_k are located at S_i, S_j and S_k, and the result should be sent to S_r. Two schedules have to be considered. In the first one both A_i and A_j are transmitted in parallel to S_k where the intersections are computed. The result is sent to the result site. Such a schedule can be replaced by one that consists of a direct transmission of A_i to the result site and the transmission of A_j to S_k and the

result of the intersection $A_j \cap A_k$ to the result site. This latter schedule will not have a larger RTT than the original one.

The second schedule consists of the transmission of A_i to S_j, the result of the intersection $A_i \cap A_j$ to S_k, and finally $(A_i \cap A_j) \cap A_k$ to the result site. Transmitting A_i to both S_j and S_k in parallel and the results of the intersections $A_i \cap A_j$ and $A_i \cap A_k$ to the result site will not have a larger response transmission time:

$$\begin{aligned} RTT_r(\{A_i \cap A_j, A_i \cap A_k\}) \\ &= \max(TT_{ij}(A_i \cap A_j) + TT_{jr}(A_i \cap A_j), \\ &\qquad TT_{ik}(A_i) + TT_{kr}(A_i \cap A_k)) \\ &\leqslant \max(TT_{ij}(A_i) + TT_{jk}(A_i \cap A_j) + TT_{kr}(A_i \cap A_j \cap A_k), \\ &\qquad TT_{ik}(A_i) + TT_{kr}(A_i \cap A_k)) \\ &= TT_{ij}(A_i) + TT_{jk}(A_i \cap A_j) + TT_{kr}(A_i \cap A_j \cap A_k). \end{aligned}$$

The proof for $A_i \cap A_j \cap A_k \cap B_p'$ goes exactly the same.

□

Before showing how the algorithm that computes minimum response transmission time schedules for a term can be implemented, we introduce some notions.

Let the **minimum transmission time A-list to site** S_x be the A-list A_f such that

$$RTT_x(A_f) = \min_j TT_x(A_j).$$

Let the **second minimum transmission time A-list to site** S_x be the A-list A_s such that

$$RTT_x(A_s) = \min_{j \neq f} RTT_x(A_j),$$

where f is the index of the minimum transmission time A-list to site S_x.

Let the **minimum response transmission time intersection to site** S_x be the intersection of two A-lists, $A_i \cap A_j$, such that

$$RTT_x(A_i \cap A_j) = \min_{k,l} RTT_x(A_k \cap A_l).$$

Let the **minimum transmission time A-list to site** S_r **through** S_x be the minimum transmission time A-list to site S_r, with the restriction that it is first transmitted to S_x and then from S_x to S_r.

What alternatives are there to transmit the relevant parts of the A- and B-lists to the result site, S_r? There are two choices for transmitting A_i.

A1) A_i is directly transmitted from S_x to S_r.
$RTT_r(A_i) = TT_{xr}(A_i)$.

A2) A_i is intersected with another A_j at S_x before transmitting the result, $A_i \cap A_j$, to S_r.
For A_j not all A-list have to be considered. The best choice for A_j is the

minimum transmission time A-list to S_x. There is, however, a chance that that list is A_i itself. In that case we have to consider our second best choice, the second minimum transmission time A-list to S_x.

$RTT_r(A_i) = RTT_x(A_j) + TT_{xr}(A_i \cap A_j)$.

B_p can be transmitted in three ways:

B1) B_p is directly transmitted from S_x to S_r.

$RTT_r(B_p) = TT_{xr}(B_p)$.

B2) B_p is intersected with an A_i at S_x, and the result, $A_i \cap B'_p$, is transmitted to S_r.

The best choice for A_i is the minimum response transmission time A-list to S_r through S_x.

$RTT_r(B'_p) = TT_{yx}(A_i) + TT_{yr}(A_i \cap B'_p)$.

B3) B_p is intersected with $A_i \cap A_j$ at S_x and the result, $(A_i \cap A_j) \cap B'_p$, is transmitted to S_r.

This alternative has to be considered because an intersection of an A-list with a complemented list will have hardly reduced the size of the A-list. The best choice for $A_i \cap A_j$ is the minimum transmission time intersection to S_r through S_x. Because the cost of transmitting $(A_i \cap A_j) \cap B'_p$ from S_x to S_r is the same for all i and j we can suffice with taking the $A_i \cap A_j$ that is the minimum response transmission time intersection to S_x.

$RTT_r(B_p) = RTT_x(A_i \cap A_j)) + TT_{xr}((A_i \cap A_j) \cap B'_p)$.

Note: in A2 (B2 and B3) we do not have to consider transmitting A_i (B_p), because the minimum response time schedule for A_i (B_p) would be at least as large as the one of alternative A1 (B1).

To complete the algorithm we show how the minimum and second minimum transmission time A-lists and the minimum response transmission time intersections are computed. By computing the transmission times of all A-lists we can determine the minimum and second minimum transmission time A-lists to S_x and also to S_r through S_x. Let A_i and A_j be the two A-lists with the smallest transmission time to get to S_x. Then the minimum response transmission time intersection to S_x can be obtained by either sending A_i and A_j in parallel to S_x or to send the smallest of the two, say A_i, first to A_j and transmit $A_i \cap A_j$ to S_r. In fig. 3.14 we show the algorithm.

Example 3.5

The network in our example is fully connected, this means that every pair of sites is directly connected. The transmission time $TT(X) = 50 + |X|$. The construction of the schedules for the first term, $A \cap B \cap C \cap D'$, will be discussed in detail. The algorithm of fig. 3.14 has three for-loops. We will go through each of them. The results after the first for-loop are listed in table 3.15. Each entry consists of an expression and its RTT to get to site S_x. Let us look at the entries for site S_3. Because C is the smallest A-list and no A-list is located at site S_3 its minimum transmission time A-list is C. B is the second smallest and therefore the second

```
proc MRTT term=(term Q_t,site S_r)schedule:
        {Q_t = A_1 ∩ A_2 ∩ ··· ∩ A_n ∩ B'_1 ∩ B'_2 ∩ ··· ∩ B'_m}
begin
   schedule sch := empty schedule;
   foreach S_x
   do
      determine:
      the minimum transmission time A-list to S_x;
      the second minimum transmission time A-list to S_x;
      the minimum response transmission time intersection to S_x;
      the minimum transmission time A-list to S_r through S_x
   od;
   foreach A_i
   do
      consider alternatives A1 and A2;
      take the one with the smallest response transmission time, say its
      schedule is sch_i;
      sch := integrate(sch,sch_i)
   od;
   foreach B_p
   do
      consider alternatives B1, B2 and B3;
      take the one with smallest response transmission time, say its
      schedule is sch_l;
      sch := integrate(sch,sch_l)
   od;
   sch
end
```

Figure 3.14. Algorithm *MRTT term.*

minimum transmission time A-list to site S_3. The minimum transmission time intersection to S_3 is obtained by transmitting C to S_4 and send the result of the intersection, $B \cap C$, to S_3; response transmission time 250 + 50 = 300. C is also the minimum transmission time A-list to the result site, S_1, through S_3.

	S_2		S_3		S_4		S_5	
min. A-list	A	0	C	250	B	0	C	0
sec. min. A-list	C	250	B	450	C	250	B	450
min. int.	$A \cap C$	250	$B \cap C$	300	$B \cap C$	250	$B \cap C$	300
min. A-list S_1	C	250	C	250	B	0	C	0

Table 3.15. Results after first for-loop.

In the second for-loop for each of the A-lists the alternatives A1 and A2 are considered.

list	alt.	response transmission time
A	A1:	$TT_{21}(A) = 1050$
	A2:	$TT_{52}(C) + TT_{21}(A \cap C) = 250 + 50 = 300$
B	A1:	$TT_{41}(B) = 450$
	A2:	$TT_{54}(C) + TT_{41}(B \cap C) = 250 + 50 = 300$
C	A1:	$TT_{51}(C) = 250$
	A2:	$TT_{45}(B) + TT_{51}(B \cap C) = 450 + 50 = 500$

In the third loop a schedule is constructed for the complemented lists by considering the alternatives B1, B2 and B3.

list	alt.	response transmission time
D	B1:	$TT_{31}(D) = 2050$
	B2:	$TT_{53}(C) + TT_{31}(C \cap D') = 250 + 250 = 500$
	B3:	$TT_{54}(C) + TT_{43}(B \cap C) + TT_{31}((B \cap C) \cap D') = 350$

For each of the lists the schedule with the minimum response transmission time is chosen. So, for list A alternative A2 is taken, for B alternative A2, for C alternative A1 and for D alternative B3. Because B and C participate in other schedules we can drop their own schedules. To obtain the result of the term $A \cap B \cap C \cap D'$ the results of $A \cap C$ and $(B \cap C) \cap D'$ are intersected at the result site.

For the other term, $A \cap B \cap E'$, we can do exactly the same. Again, because both A and B participate in the schedule of E we drop their own. The minimum response transmission time schedule of E is obtained by alternative B3.

list	alt.	response transmission time
E	B3:	$TT_{42}(B) + TT_{21}((A \cap B) \cap E') = 450 + 50 = 500.$

□

The algorithm *MRTT query* of fig. 3.11 calling the algorithm *MRTT term* of fig. 3.14 is the algorithm that computes the minimum response transmission time schedule for a query in the Equal Transmission Cost Model.

Lemma 3.9 The worst case complexity of the algorithm *MRTT term* of fig. 3.14 is $O(nN + m)$, where n and m are the number of A-lists and B-lists, respectively and N is the number of sites participating in the query.

Proof The algorithm consists of three for-statements. We will investigate their complexity in turn. For each S_x and each A_i we have to compute either the RTT of A_i in S_x or the RTT of A_i in S_r if its transmission goes through S_x. This will cost $O(nN)$. The second loop is executed n times and its cost is constant. The same is

true for the last loop which is executed m times. So, the whole algorithm costs $O(nN + m)$.

□

Theorem 3.10 The worst case complexity of the algorithm *MRTT query* of fig. 3.11 in the Equal Transmission Cost Model, is $O(MN)$, where M is the number of literals in the disjunctive normal form of a query and N the number of sites participating in the query.

Proof To rewrite a query in its disjunctive normal form will take no longer than $O(M)$. The worst case for the algorithm that computes the minimum response transmission time schedule for a term is obtained if the whole query is just one term. Furthermore, the maximum of $nN + m$, where $n + m = M$ is reached if $n = M$. So, the worst case complexity of the algorithm is $O(MN)$.

□

3.3.6. Arbitrary Transmission Cost Model

For a network with an arbitrary topology or if the queueing delays for the different communication channels are not the same, the Equal Transmission Cost Model can no longer be used. In this subsection no particular assumptions are made about the cost to transmit data from one site to another (Arbitrary Transmission Cost Model).

What makes the general case - arbitrary transmission cost between sites - more difficult? The answer is that lemma 3.8 is no longer true. This means that all operation trees of theorem 3.5 have to be considered, and also that sites other than those where the lists in the considered operation tree reside, have to be included in the processing schedule. Again we will discuss the alternatives to transmit the relevant parts of the A- and B-lists.

What are our alternatives for transmitting A_i, located at site S_x, to the result site, S_r?

A1) Transmit A_i directly from S_x to S_r.

$RTT_r(A_i) = TT_{xr}(A_i)$.

A2) Transmit A_i to site S_y (y not necessarily different from x). Here again there are two alternatives:

a) Let S_y intersect A_i with an A-list, say A_j ($j \neq i$); for A_j we only need to consider the minimum or second minimum transmission time A-list to site S_y.

$RTT_r(A_i) = \max(TT_{xy}(A_i), RTT_y(A_j)) + TT_{yr}(A_i \cap A_j)$.

b) Let S_y intersect A_i with the result of another intersection, say $A_j \cap A_k$; for this intersection we only need to consider the minimum response transmission time intersection to site S_y.

$RTT_r(A_i) = \max(TT_{xy}(A_i), RTT_y(A_j \cap A_k)) + TT_{yr}(A_i \cap (A_j \cap A_k))$.

Among all these alternatives choose the one with the smallest response transmission time for A_i to get to S_r. The other alternative $(A_i \cap A_j) \cap A_k$ need not be

considered because the result of $A_i \cap A_j$ is already small and there is no way to improve the response transmission time of A_i to get to S_r by intersecting it with A_k.

Another notion is added to the ones introduced in the subsection on the Equal Transmission Cost Model.

Let the **minimum response transmission time intersection to site S_r through S_x** be the intersection $A_i \cap A_j$ such that

$$RTT_r(A_i \cap A_j) = \min_{p,q} (A_p \cap A_q),$$

or the intersection $A_i \cap (A_j \cap A_k)$ such that

$$RTT_r(A_i \cap (A_j \cap A_k)) = \min_{p,q,s} RTT_r(A_p \cap (A_q \cap A_s),$$

and both with the restriction that A_i its transmission goes through S_x.

This can be computed by considering for each A_p, S_x and S_y the transmission of A_i to S_y through S_x and intersect A_i with the minimum transmission time A-list or with the minimum response transmission time intersection to S_y. The result is transmitted to S_r.

For the transmission of B'_p, located at site S_x, to the result site there are five alternatives:

B1) Transmit B_p directly from S_x to S_r.
$RTT_r(B_p) = TT_{xr}(B_p)$.

B2) Transmit B_p to site S_y (y not necessarily different from x).

a) Consider intersecting B'_p with an A-list, say A_i. For such an A_i we only have to consider the minimum transmission time A-list to S_r through S_x.
$RTT_r(B_p) = \max(TT_{xy}(B_p), RTT_y(A_i)) + TT_{yr}(A_i \cap B'_p)$.

b) let S_y intersect B'_p with the result of the expression $A_i \cap A_j$. For this expression we only need to consider the minimum response transmission time intersection at site S_x.
$RTT_r(B_p) = \max(TT_{xy}(B_p), RTT_y(A_i \cap A_j)) + TT_{yr}((A_i \cap A_j) \cap B'_p)$.

B3) Again transmit B_p to site S_y and let S_y compute the intersection $A_i \cap B'_p$, transmit the result to S_z and intersect it there with either A_j or $A_j \cap A_k$. For A_i, A_j, A_k and S_z we only need to consider the minimum response transmission time intersection to S_r that goes through S_z. We will give the response transmission time if $A_i \cap B'_p$ is intersected with A_j.

$$\begin{aligned} RTT_r(B_p) = \max(&\max(TT_{xy}(B_p), RTT_y(A_i)) \\ &+ TT_{yz}(A_i \cap B'_p), \\ &RTT_z(A_j)) \\ &+ TT_{zr}((A_i \cap B'_p) \cap A_j) \end{aligned}$$

The heart of the algorithm that determines the minimum response transmission

time for a term consists of going through all the alternatives and taking for each A_i and B'_p the schedule with the smallest response transmission time. See fig. 3.16. The schedule for the term Q_t is the integration of the schedules of all its literals (A-lists and B-lists). Evidently, it will contain a lot of superfluous transmissions. They do not affect the response transmission time but it will be better to remove them to make the overall network load less heavy.

```
proc MRTT term=(term Qt,site Sr)schedule:
        {Qt = A1 ∩ A2 ∩ ··· ∩ An ∩ B'1 ∩ B'2 ∩ ··· ∩ B'm}
begin
   schedule sch := empty schedule;
   foreach Sx
   do
      determine:
      the minimum transmission time A-list to Sx;
      the second minimum transmission time A-list to Sx;
      the minimum response transmission time intersection to Sx;
      the minimum transmission time A-list to Sr through Sx;
      the minimum response transmission time intersection to Sr through
      Sx
   od;
   foreach Ai
   do
      consider alternatives A1, A2a and A2b;
      take the one with the smallest response transmission time, say its
      schedule is schi;
      sch := integrate(sch,schi)
   od;
   foreach Bp
   do
      consider alternatives B1, B1a, B2a and B3;
      take the one with the smallest response transmission time, say its
      schedule is schl;
      sch := integrate(sch,schl)
   od;
   sch
end
```

Figure 3.16. Algorithm *MRTT term.*

Lemma 3.11 The worst case complexity of the algorithm *MRTT term* of fig. 3.16 is $O(nN^2 + mN)$, where n and m are the number of A-lists and B-lists, respectively, and N is the number of sites in the computer network.

Proof In the first for-statement the computation of the minimum response transmission time intersection to S_r through S_x is most costly. For each A_i, S_x and S_z we

have to compute the transmission time to S_r. So, this will cost $O(nN^2)$. The second loop only takes $O(nN)$ and the last one $O(mN)$. Hence, the whole algorithm takes $O(nN^2 + mN)$.

□

Theorem 3.12 The worst case complexity of the algorithm *MRTT query* of fig. 3.11 in the Arbitrary Transmission Cost Model is $O(MN^2)$, where M is the number of literals in the disjunctive normal form of the query and N is the number of sites in the computer network.

Proof Goes exactly the same as the one of theorem 3.10.

□

Looking at the alternatives to be considered for the Arbitrary Transmission Cost Model compared to those for the Equal Transmission Cost Model, we can say that the problem of determining schedules to minimize the response transmission time of a query has become a lot more difficult. There are two factors responsible for this. First, the need to intersect the lists to get a small intermediate result and secondly, the choice of the site where this intersection should take place, taking into account the arrival of both operands and the final transmission of the result. To get a better understanding of distributed query processing, it is better to assume equal transmission cost, otherwise the network topology might cause unexpected schedules. However, one should realize that the results obtained in the Equal Transmission Cost Model are not always directly applicable to the general model.

Nowhere in the theorems nor in the resulting algorithms is any assumption made about the materialization presented to the optimizer. Therefore, the produced schedules are still optimal even if a redundant allocation is used. The optimizer will decide which copy to use; it may even use several copies of the same list in one schedule. Whether this is feasible depends on the mutual consistency of the copies. If copies are identical all the time, the optimizer is free to use as many copies as are required. If, on the other hand, copies are notified of changes independent of each other, they may be different and in that case the optimizer should be given a non-redundant materialization.

3.3.7. Response Transmission Times Before and After Serialization

We have implemented the minimum response transmission time algorithm for the Equal Transmission Cost Model. A comparison has been made between the response transmission times before and after serialization of parallel schedules that use the same communication channels. Furthermore, the serializations obtained by the heuristic approach discussed in subsection 3.1.6 and the optimal ones are compared. No further changes were made in the schedule, except transmissions of identical data over the same channel were avoided. As one would expect, the response transmission time after serialization is larger. However, one should keep in mind that an increase of the response transmission time after serialization does not necessarily mean that the obtained processing schedule is not optimal. For example, if two inverted lists that reside at the same site have to be united, the produced schedule will consist of the transmissions of both lists to the result site. Before serialization the response transmission time will be equal to the transmission time of the largest list. After serialization it is equal to the transmission time of a list whose size equals the sum of the sizes of the two lists. An alternative would be to unite the two lists before transmission, and, under the assumption that the size of a union is equal to the sum

of the sizes of the operands, the response transmission time of this schedule is not better than the serialized one.

We assume that the lists are distributed over a small network consisting of these sites; this figure is chosen rather small to compute the optimal serialization of the schedules. The disjunctive normal form of the queries are randomly generated by drawing the following parameters from a uniform distribution with integer values:

size of lists 1, 2, . . . , 10000
number of terms (*terms*) 1, 2, 3
number of lists per term (*list/term*) 1, 2, 3, 4
percentage of A-lists per term (*A-lists/term*) 20, 40, 60, 80, 100
size intersection 100
delay 1000

We will compare the response transmission times before and after serialization if we change the number of terms, the number of lists per term and the percentage of A-lists per term. Also the ratio between the two figures is given. The comparison is made by singling out a particular parameter and running a hundred queries for each discrete value of that parameter and letting the other ones be drawn randomly from their respective domains.

On the average the schedules produced by the heuristic serialization algorithm were 2% worse than the optimal serializations (only the optimal serializations that could be computed in a reasonable amount of time were compared with the heuristic ones). One should, however, keep in mind that the parameters were kept small to be able to compute the optimal serialization of the schedules.

From table 3.17 we may conclude that the response transmission time after serialization is much larger than under the Parallelism Assumption if one, or both, of the following conditions holds:

- the schedule consists of many parallel schedules for which the response transmission time is computed independent of each other, and that share few resources (e.g., many terms),
- communication channels are occupied for a long period of time by large transmissions (e.g., few lists per term or few A-lists per term).

We expect that the above conclusion also holds if response time, instead of only response transmission time, is minimized. A comparison between the optimal processing of a query and the schedules produced by first computing them under the Parallelism Assumption and then serializing them, is far more time consuming than the comparison discussed above, because of the many ways the lists can be united and intersected with each other.

3.3.8. Minimizing Response Time

The response time of a query is determined by the response times of the basic operations, which for an inverted file system are the union, the intersection and the transmission. We expect that finding the minimum response time is not feasible, because of the enormous number of solutions. One reason that causes this has already been discussed, namely the serialization of a schedule. Therefore, we proposed a separate optimization of the response transmission time and the response

terms	1	2	3
before	4229	5979	5989
after	4229	6469	7405
ratio	1.0	1.08	1.23

lists/term = 3, *A-lists/term* = 80%

lists/term	1.0	2.0	3.0	4.0
before	4488	6064	5751	3086
after	4562	6133	6234	3363
ratio	1.02	1.01	1.08	1.09

terms = 2, *A-lists/term* = 80%

A-lists/term	20%	40%	60%	80%	100%
before	5903	5772	6397	4652	5393
after	6354	5979	6594	4859	5639
ratio	1.08	1.04	1.03	1.04	1.05

terms = 2, *lists/term* = 3

Table 3.17. Varying different parameters.

processing time. Under the Transmission Assumption the optimizer first minimizes the response transmission time, which was done in the previous subsection. The **macro-schedule** obtained is then serialized as far as the transmissions are concerned, such that again the response transmission time is minimized. If local processing can be neglected compared to transmissions, the optimizer is finished. This is also true if minimizing the response processing time of a site has no influence on the macro-schedule, and so it can be done by the site itself. The approach of minimizing response processing time at both optimization time and execution time will be discussed in more detail now.

Giving every site the responsibility to minimize its contribution to the response time of the query has the advantage that the system can react more actively on the differences from the expected response transmission times of the lists. For example, a site has to compute the order in which a couple of lists is intersected or united. This order can be important, however, in general some freedom is left to choose an order. The response transmission time of the lists given by the schedule is merely an estimate

of their real arrival times. The flexibility can be obtained by starting with an initial execution-order based on the expected response transmission times and gradually adjusting it when some of the lists arrive.

The disadvantage of postponing the computation of the processing schedule for each site is that a complete view of the serialization of the schedule is lost. For example, a site has to execute operations in different parallel schedules. The effect of a certain serialization of the operations can not be estimated, because other sites may take conflicting decisions. To overcome this problem, the optimizer should include local processing in the schedule produced, when minimizing response transmission time. This is done as follows. First, a macro-schedule is computed which determines the duties that have to be done by the sites involved. To know the response transmission times of the intermediate results at a particular site, the macro-schedule is serialized. From this and the duties the **micro-schedules** can be determined, again under the Parallelism Assumption. Then the micro-schedules are **integrated** in the macro-schedule, which is serialized again. This is the final schedule which is transmitted to all sites involved.

Fig. 3.18 shows that two intersections of two parallel schedules have to be computed at site S before serialization. The rectangles stand for the computation of the result with which they are labeled. Their lengths stand for the duration of the computation. They are drawn next to the time scale to emphasize that they have not been serialized yet.

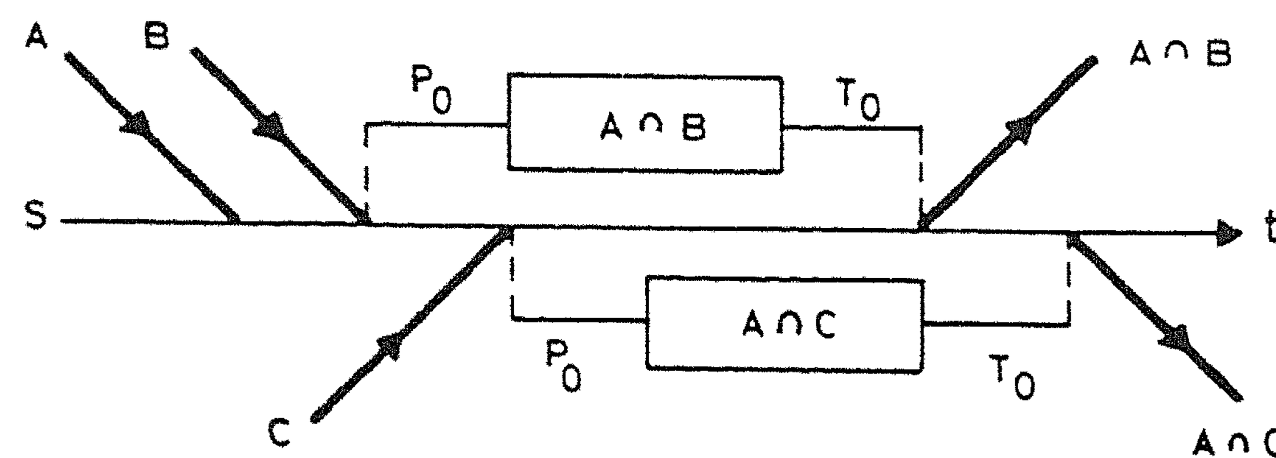


Figure 3.18. Two parallel micro-schedules.

At this point we assume that the macro-schedule has been computed, implying that the duties of the sites involved are known. How the micro-schedules can be computed is discussed now. Note, determining a micro-schedule is not the same as determining a schedule for a centralized database because not all data on which the operations are performed need be available from the start. The micro-schedules are merely computed to show this difference and the way they are integrated in the macro-schedule. Therefore, no attempt is made to incorporate knowledge about the way the data are stored on disk, such as clustering. We assume that the cost to retrieve a list from disk is proportional to its size (Local Processing Assumption).

Let us first take a look at a **non-result site**. Its major duty is to compute the intersection of two lists, which may themselves be the results of previously computed intersections. If so, the site merely has to wait until both operands of the intersection have arrived. The response processing time of such an intersection, say $A \cap B$, computed by S_x is

$$RPT_x(A \cap B) = \max(RTT_x(A), RTT_x(B)) + PT_x(A \cap B).$$

A more complex expression such as $A_i \cap A_j \cap A_k$, where both intersections have to be computed by the same site, will not occur, because the algorithm that minimizes the response transmission time considers the two intersections $A_i \cap A_j$ and $A_i \cap A_k$ first and it can easily be seen that their RTT_r's are no larger than the $RTT_r(A_i \cap A_j \cap A_k)$.

The only interesting subexpression tree that remains, is $A_i \cap A_j \cap B_l'$, where both intersections are computed by S_x. Three cases can be distinguished: (1) both A_i and A_j arrive before B_l, (2) either A_i or A_j arrives later than B_l, and (3) both A_i and A_j arrive later than B_l. In cases (1) and (3), first the intersection of the A-lists is computed and the result intersected with the complemented list B_l' (RPT_x'). In case (2) there are two alternatives: either the site S_x waits until both A-lists arrive and then does exactly the same as in case one and three, or the first arriving A-list is intersected with B_l' and the result is intersected with the other A-list (RPT_x''). The corresponding response processing times are:

$$RPT_x'((A_i \cap A_j) \cap B_l') = \max(RTT_x(A_i), RTT_x(A_j)) + PT_x(A_i \cap A_j) + PT_x((A_i \cap A_j) \cap B_l')$$

$$RPT_x''((A_i \cap B_l') \cap A_j) = \max(\max(RTT_x(A_i), RTT_x(B_l)) + PT_x(A_i \cap B_l'), RTT_x(A_j)) + PT_x((A_i \cap B_l') \cap A_j).$$

If we write $PT_x(X \cap Y) = P_0 + P_1(|X| + |Y|)$ (Local Processing Assumption), then we can easily derive the condition that the first alternative is better. Assume

$$RTT_x(A_i) \leqslant RTT_x(B_l) \leqslant RTT_x(A_j).$$

Then,

$$RPT_x' = RTT_x(A_j) + 2P_0 + P_1(|A_i| + |A_j| + |B_l|), \text{ and}$$

$$RPT_x'' = \max(RTT_x(B_l) + P_0 + P_1(|A_i| + |B_l|), RTT_x(A_j)) + P_0 + P_1(|A_i| + |A_j|).$$

If $RTT_x(A_j)$ is larger than the response processing time of the intersection between A_i and B_l' then the first alternative can not be better. So, $RPT_x' \leqslant RPT_x''$ if

$$RTT_x(A_j) \leqslant RTT_x(B_l) + P_1|A_i|.$$

Now consider the **result site**. Its major task is to compute all the union operations of the expression tree in the disjunctive normal form. Occasionally, it may also have to compute an intersection. First we consider an expression tree consisting of only union operations. In [Liu1976] an algorithm is given to compute the union of lists such that the total processing time is minimized. The processing schedule obtained has, however, also minimum response processing time, if all lists are already available at the result site. The reason is that only one CPU is used and the number of operations is the same. Liu's algorithm forms the basis of the algorithm that we propose, therefore we will discuss it first.

Consider a query that consists of only unions, and that all the inverted lists are available at the result site. The minimum total (or response) processing time is obtained by processing the query according to a tree with minimum weighted path length. The weight of the leaves, the lists, is their size and all the other nodes, the operations, have weight zero. Such a tree, called a Huffman tree, can be obtained by applying Huffman's algorithm [Huffman1952]. Huffman's algorithm manipulates a set of subtrees. Initially, this set contains all the lists of the query. During every iteration it takes two subtrees with smallest weight together in a new subtree, consisting of a union operation with the two selected subtrees as its operands. Finally, only one (sub) tree remains.

In general, not all operations are unions and not all the inverted lists will have the same response transmission time to the result site. A simple approach, which is similar to the algorithm of Liu, is to execute the operation that will increase the response processing time the least.

3.3.9. Minimizing Total Time

Minimizing the response time of a query is desirable for the user. However, the schedules produced by the query processing algorithm will contain many transmissions and operations that are superfluous. The latter ones are detected during serialization and can be removed. If the communication channels or the sites are already heavily loaded with transmissions or operations of other queries, minimizing the response time of a query will do no good for either the system and the user. Therefore, minimizing the total transmission time is also an important issue and will be considered here. In Theorem 3.13 and Corollary 3.14 it is shown that a query can be rewritten into its disjunctive normal form, without increasing the total transmission time. When minimizing the response transmission time the terms could be treated independently. Because of the forking points in a schedule, this can not be done when minimizing total transmission time. Hence, the schedules produced are not necessarily optimal.

Theorem 3.13 Under the Transmission Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model applying the distributive law to a query will not increase its total transmission time.

Proof Assume we are given a schedule for a query and part of it computes

$$(A \cup B) \cap C,$$

where A, B and C may be lists or intermediate results. Also, C may stand for an intermediate result or its complement. Somewhere in this schedule $A \cup B$ is computed, say at site S_1, and the result is sent to another site, say S_2, where it is

intersected with C. Simply postponing the union of A and B until they have been intersected with C is not possible, because between the union of A and B and the intersection with C there might be several forking points in the schedule, in the case that the result of $A \cup B$ is not only used at S_2. In the Equal Transmission Cost Model there is at most one such forking point. The way the schedule is changed will be discussed only for this case. The more general case follows right away.

The schedule is changed as follows. Instead of computing the union of A and B, they are concatenated, denoted by $Conc(A,B)$, and this concatenation is sent to the sites that previously received the union. Computing this concatenation is merely a trick to make sure that the queueing delay before transmission is counted only once in the total transmission time. It is not a new operation on lists and, therefore, it is not one of the basic operations in the processing schedules. If the total transmission cost was minimized this trick would not be needed. All the sites that previously received the union, except S_2, will split the concatenation and compute the union. Site S_2 also splits the concatenation but then both A and B are intersected with C. After that the two results are united.

As far as transmissions are concerned the only change in the schedule is that instead of $A \cup B$, $Conc(A,B)$ is transmitted. The transmission times of the union and of the concatenation are the same under the Intersection Assumption. More precisely,

$$|A \cup B| = |A| + |B| - |A||B| / |X|,$$

which equals $|A| + |B|$ if $|A||B| / |X|$ is neglectably small. So, $|A \cup B| = |Conc(A,B)|$. The cost of the former is

$$TT_{ij}(A \cup B) = T_0 + TC_{ij}(A) + TC_{ij}(B),$$

which is the same as $TT_{ij}(Conc(A,B))$. Hence, the total transmission cost has not changed.

□

Corollary 3.14 Under the Transmission Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model a schedule of a query with minimum total transmission time can be replaced by a schedule for the disjunctive normal form of the query without sacrificing its optimality.

Theorem 3.15 Under the Transmission Assumption and the Intersection Assumption and in the Arbitrary Transmission Cost Model the union operations of a query in disjunctive normal form can be executed at the result site without increasing its total transmission time.

Proof Instead of transmitting the result of a union, the concatenation is transmitted. If at the receiving site the union is required the concatenation is split and the union is computed.

□

The results obtained are exactly the same as for minimizing the response transmission time. However, here we have to be careful not to change the schedule too much. For example, the transmission of $Conc(A,B)$ has to go along exactly the same channels as $A \cup B$ does in the original schedule. The reason for this is that

$A \cup B$ might be needed in a different part of the schedule as well. Also, the transmission of C to the site that computes the intersection has to stay the same. In minimizing the response transmission time the computation of $A \cap C$ and $B \cap C$ could be treated independently because of the Parallelism Assumption. Here, this is out of the question, because it might mean that C is transmitted to several sites or that other lists are transmitted to C, thus increasing the total transmission time. Because the forking points have not been removed from the schedules we may not simply rewrite the query to its disjunctive normal form and determine the minimum total transmission cost for each of the terms independently.

Whether minimizing the total transmission time of a query can be solved in polynomial time is an open problem, as far as the author knows. Therefore, we will take a heuristic approach. The minimum total transmission time schedules for each of the terms will be determined independently, and after that, redundant transmissions are removed.

The algorithm to compute an efficient total transmission time schedule for processing a query is given in fig. 3.19. The procedure *TTT term* will be supplied later.

```
proc TTT query=(query Q)schedule:
begin
    schedule sch;
    put Q in disjunctive normal form;
                {say, Q1 ∪ Q2 ∪ ··· ∪ Qd}
    for t to d
    do
        sch := sch ∪ TTT term(Qt)
    od;
    sch
end
```

Figure 3.19. Algorithm *TTT query*.

In the following, we assume that a term looks like

$$A_1 \cap A_2 \cap \cdots \cap A_n \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m,$$

and that all lists reside at different sites. If this is not true the intersection(s) should be computed locally. Also, we assume that none of the B-lists resides at the result site. Again, if so, they can be deleted from the term without increasing the total transmission time of the optimal schedule.

Only the Equal Transmission Cost Model is discussed.

Theorem 3.16 Under the Transmission Assumption and the Intersection Assumption and in the Equal Transmission Cost Model the minimum total transmission time for processing

$$A \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m,$$

where $|B_i| \leqslant |B_{i+1}|$ for $i = 1, 2, \cdots, m - 1$, is obtained by a schedule consisting either of the transmissions of A and all B_i's to the result site, or of the transmissions of $B_1, B_2, \ldots, B_k$, where $|B_k| \leqslant |A|$ and $|A| < |B_{k+1}|$ to the result site and the following transmissions

$$A \rightarrow B_{k+1} \rightarrow B_{k+2} \rightarrow \cdots \rightarrow B_m \rightarrow \text{result site.}$$

Proof Transmitting B_i to B_j and the result $B_i \cup B_j$ (Note, that $(B_i \cup B_j)' = B_i' \cap B_j'$) to the result site will cost no less than transmitting both B_i and B_j to the result site. Therefore, for B_i to be part of the processing it is either transmitted to the result site or it is part of

$$A \rightarrow B_{k+1} \rightarrow B_{k+2} \rightarrow \cdots \rightarrow B_m \rightarrow \text{result site.}$$

In the first alternative the cost for B_i to be part of the schedule is $TC(B_i)$. For the second alternative we charge the cost of a transmission leaving the site where a list L resides to L itself. Then the cost for B_i is $T(A)$. Thus, if $|B_i| \geqslant |A|$, it is better to transmit B_i directly to the result site.

The remaining cost of the second alternative is

$$(m - k + 1)TC(A).$$

The cost of any other schedule for $B_{k+1}, \ldots, B_m$ containing at least $m - k + 1$ transmissions can be no less because A is the smallest remaining list and there is no way to reduce the sizes of the lists. The total transmission cost of the second alternative is

$$\sum_{i=1}^{k} TC(B_i) + (m - k + 1)TC(A).$$

If A resides at the result site we also have to consider transmitting all the B_i's to the result site, because it means one transmission less. Its cost is

$$\sum_{i=1}^{m} TC(B_i).$$

Again, among all schedules containing m transmissions and in which all B_i's are included, it is the cheapest one, because transmitting one B-list to another does not decrease the total transmission time.

□

Theorem 3.17 Under the Transmission Assumption and the Intersection Assumption and in the Equal Transmission Cost Model the minimum total transmission time for processing

$$A_1 \cap A_2 \cap \cdots \cap A_n \cap B' \quad (n \geqslant 3),$$

where A_1 and A_2 are the smallest and second smallest A-list, respectively, is either obtained by the schedule

$$A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow B \rightarrow \text{result}$$

site,

or by

$$A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_{i-1} \rightarrow A_{i+1} \cdots \rightarrow A_n \rightarrow B \rightarrow \text{result}$$

site,

if A_i resides at the result site.

Proof The total transmission time of the first alternative is

$$TC(A_1) + nT_0 = (n + 1)T_0 + T_1|A_1|.$$

Every other schedule that contains $n + 1$ or more transmissions will be more costly because A_1 is the smallest A-list.

The total transmission time of the second alternative is either

$$TC(A_1) + (n - 1)T_0 = nT_0 + T_1|A_1|,$$

or

$$nT_0 + T_1|A_2|.$$

depending on whether i equals 1 or not. Such a schedule containing only n transmissions can not contain the transmission of A_i, where A_i resides at the result site, otherwise the schedule would not visit all other lists in the term. Hence, every other schedule that has n transmissions must at least be as expensive as the second alternative.

Which of these two alternatives is better depends on the sizes of A_1 and A_2, and where A_1 resides.

Evidently a schedule with less than n transmissions is unfeasible, because all lists reside at different sites.

□

Corollary 3.18 Under the Transmission Assumption and the Intersection Assumption and in the Equal Transmission Cost Model the minimum total transmission time for processing

$$A_1 \cap A_2 \cap \cdots \cap A_n \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m \qquad (n \geq 3)$$

is obtained by the schedule

$$S \rightarrow B_2 \rightarrow \cdots \rightarrow B_m \rightarrow \text{result site,}$$

where S is the minimum total transmission time schedule for processing

$$A_1 \cap A_2 \cap \cdots \cap A_n \cap B'_1,$$

with the deletion of the last transmission to the result site.

Theorem 3.19 The minimum total transmission time for processing

$$A_1 \cap A_2 \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m$$

is obtained by one of the four following schedules

$$A_1 \rightarrow A_2 \rightarrow B_1 \rightarrow B_2 \rightarrow \cdots \rightarrow B_m \rightarrow \text{result site},$$

or

$$A_2 \rightarrow A_1 \rightarrow B_1 \rightarrow B_2 \rightarrow \cdots \rightarrow B_m \rightarrow \text{result site},$$

or by the optimal schedule for processing

$$A_1 \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m,$$

or

$$A_2 \cap B'_1 \cap B'_2 \cap \cdots \cap B'_m.$$

Proof Goes along the same lines as the one of theorem 3.16.

□

The algorithm *TTT term* to compute an efficient total transmission time schedule for a term is given in fig. 3.20.

```
proc TTT term(term Qt)schedule:
        {Qt = A1 ∩ A2 ∩ ··· ∩ An ∩ B'1 ∩ B'2 ∩ ··· ∩ B'm}
begin
   schedule sch;
   if n = 1
      then
         determine schedule according to Theorem 3.16(sch)
   elif n = 2
      then
         determine schedule according to Theorem 3.19(sch)
      else
         determine schedule according to Corollary 3.18(sch)
   fi;
   sch
end
```

Figure 3.20. Algorithm *TTT term.*

Example 3.6

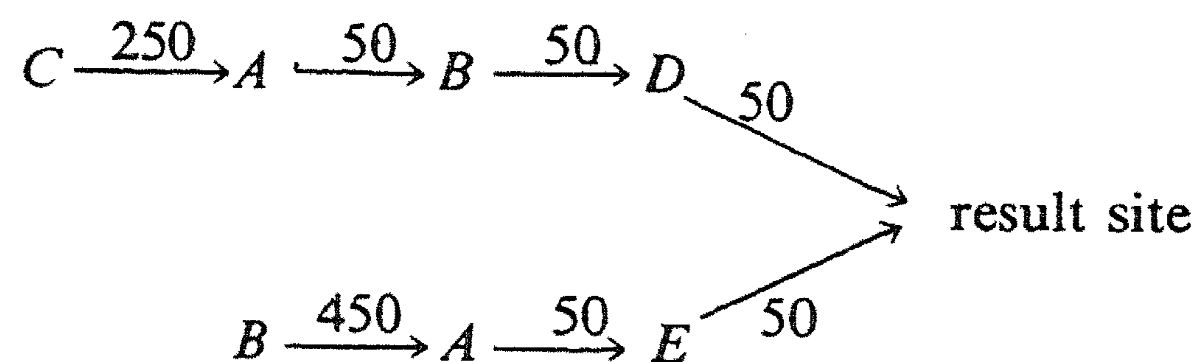
In this example we will use the same query and the same data as in example 3.3 and 3.5 of subsections 3.3.3 and 3.3.5, respectively.

The total transmission time schedule for processing $A \cap B \cap C \cap D'$ is obtained by applying corollary 3.18. The list C is transmitted to A, $C \cap A$ is sent to B, then $(C \cap A) \cap B$ to D, and the final result to the result site. The total transmission time is

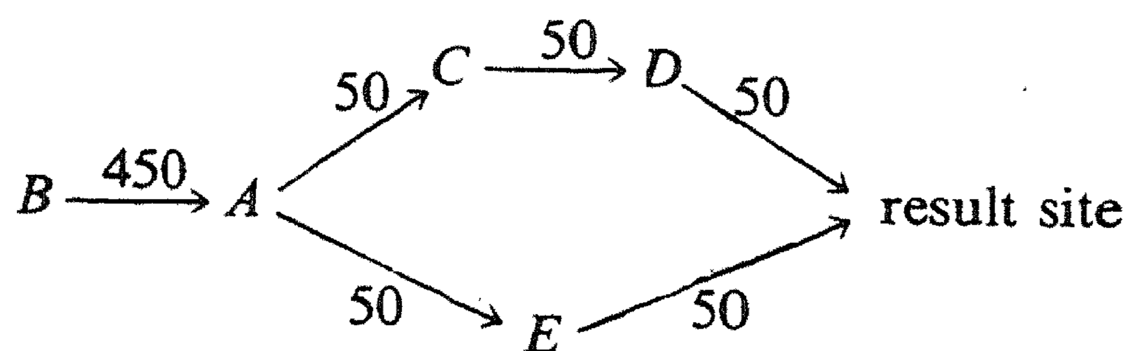
$$\begin{aligned} TTT &= TC(C) + TC(C \cap A) + TC((C \cap A) \cap B) \\ &\quad + TC(((C \cap A) \cap B) \cap D') \\ &= 250 + 50 + 50 + 50 = 400. \end{aligned}$$

In a similar way the total transmission time of the schedule for $A \cap B \cap E'$ can be computed; $TTT = 550$. So, the total transmission time of the whole query is 950. Fig. 3.21(a) shows the schedule.

This example nicely shows that considering the two terms independently does not necessarily lead to an optimal solution. Because both A and B occur in both terms it would have been better to start off with transmitting B to A. Then the schedules for both terms split. On the one hand the result of $A \cap B$ is sent to C and from there $(A \cap B) \cap C$ is sent to D and the final result to the result site; on the other hand $A \cap B$ is also sent to E and then $(A \cap B) \cap E'$ is sent to the result site. The TTT of this schedule is 700. In fig. 3.21(b) this schedule is shown.



(a)



(b)

Figure 3.21. Two schedules for $A \cap B' \cap (C \cup D)$.

□

Minimizing the total processing time for the non-result sites can be done using Liu's algorithm. If at the result site only union operations are computed the same algorithm can be used. However, sometimes at the result site also intersections are computed. In that case it might happen that the distributive law can be applied such

that the intersection is moved upward in the expression tree, thus eliminating a variable. The schedule obtained is not necessarily optimal because Liu's algorithm assumes that all variables in the query are distinct, which is not necessarily true.

3.3.10. Summary

The idea of translating a user query directly to the basic operations without first decomposing it at the logical level has been investigated for the inverted file organization. The basic operations were the transmission of data and the set operations. Both minimizing response time and total time were considered.

Minimizing response time is done in two phases. First, a macro-schedules is constructed to minimize response transmission time. This macro-schedule determines the duties of the sites involved. Secondly, each of these sites determines a micro-schedule which is integrated into the macro-schedule. Under the Transmission Assumption, Parallelism Assumption and the Intersection Assumption optimal response transmission time schedules can be produced. This is done by first rewriting the query into its disjunctive normal form and then breaking each term into small expressions, consisting of at most three A-lists and one B-list. All the union operations are executed at the result site. The macro-schedule obtained is serialized to order transmissions that share the same communication channels. A micro-schedule is computed based on the expected arrival times of intermediate results determined by the serialized macro-schedule. The advantage of letting the sites compute their own micro-schedule is that only locally available information about data storage can be taken into account. After the integration of the micro-schedules into the macro-schedule, it is serialized again to order local executions.

The same approach for minimizing total time does not necessarily lead to optimal schedules because of the forking points. Therefore, a heuristic approach is taken. Again, the query is rewritten into its disjunctive normal form and the union operations are executed at the result site. Only now schedules for complete terms are determined. In general, such a schedule consists of the transmission of the smallest A-list in the term along the other A- and B-lists in the term, and finally to the result site.

The effect of allowing for an arbitrary computer network topology on the optimality of the schedules and the complexity of the algorithms was investigated for minimizing response time. Because also sites that do not contain lists referenced in the query had to be considered for executing an intersection, the execution time of the query processing algorithms will depend on the size of the computer network. The schedules obtained remain optimal.

3.4. Distributed Query Processing Using Semi-Join

The basic operations in this section are the relational operations, restriction, projection and join [Codd1970], extended with the semi-join. This semi-join is not necessary, but it is a handy tool for reducing the amount of data to be transmitted before the computation of the join. Although not recognized at the time, Wong was most probably the first to use a sort of semi-join in distributed query processing [Wong1977]. Later this operation was more formalized [Bernstein1981a, Hevner1979a] and is now considered as a useful operation.

Our own work in this area has been done partly in parallel with A.R Hevner

and S.B. Yao [Hevner1979a, Hevner1979b, Apers1979a, Apers1979b], and partly in cooperation with them [Apers1980b]. To make this monograph self-contained some of their results will be reviewed too.

The basic idea behind the proposed algorithms is to reduce the sizes of the relations referenced in the query as much as possible and to transmit them to the result site where the joins are computed. The reduction is achieved by semi-joins. The query is considered as a whole and is not decomposed into subqueries. Integrating the proposed algorithms with a decomposition process is discussed in section 3.5.

Although the relations may be stored redundantly the optimizer expects a non-redundant materialization. Furthermore, we assume that local processing cost can be neglected completely compared to transmission cost. This implies that the results are only applicable for computer networks that have this property. If local processing cost can not be neglected we have to keep in mind that applying semi-joins may require re-scanning some of the relations.

3.4.1. Estimating Technique and Schedules

The expected cost of schedules can only be compared if the sizes of intermediate results can be estimated. To estimate the number of tuples that remain in a relation after a semi-join is applied, we need to know something about the attribute values in the relations. Let the domain of attribute A be denoted by D. The attribute values of tuples of a given relation form a subset of D. The cardinality of the subset divided by the cardinality of D will be called the **selectivity** of that attribute of the relation. The selectivity of the jth attribute of R_i is denoted by p_{ij} $(0 \leq p_{ij} \leq 1)$. We assume that the subsets corresponding to attributes of different relations are independent of each other, so that we may assume that the selectivity of the intersection of two subsets is the product of the selectivities of the two subsets.

To show how these selectivities are used to estimate the resulting size of a relation after a semi-join has been applied we assume that we know the following about the relations.

For each relation R_i, $i = 1, 2, \ldots, M$:

n_i : number of tuples,
α_i : number of attributes,
s_i : size (in number of bytes).

For each attribute A_{ij}, $j = 1, 2, \ldots, \alpha_i$ of relation R_i:

D_{ij} : domain of A_{ij},
u_{ij} : subset of D_{ij}, containing all values occurring in A_{ij},
p_{ij} : selectivity $(p_{ij} = |u_{ij}| \,/\, |D_{ij}|)$,
b_{ij} : size (e.g., in number of bytes) of u_{ij} $(b_{ij} = |u_{ij}| \times$ number of bytes in $A_{ij})$.

Suppose, the join $R_i(A_{ij} = A_{kl})R_k$ has to be computed. To omit the tuples of R_i that are not part of the join, the unique values of attribute A_{kl}, u_{kl}, with selectivity p_{kl}

are transmitted to relation R_i. The parameters of the remaining fragment, after the semi-join with A_{kl}, will be:

$$s_i \leftarrow s_i \times p_{kl},$$
$$p_{ij} \leftarrow p_{ij} \times p_{kl},$$
$$b_{ij} \leftarrow b_{ij} \times p_{kl}.$$

We discuss these parameters as if they were the changed parameters of the original relation.

In the second line the independence between the subsets u_{ij} and u_{kl} is used. The third line is a direct consequence of this independence. To compute the resulting size of R_i (s_i), we assume that each value in u_{ij} has an equal probability of being used in a tuple of R_i. Although the assumptions to estimate the resulting parameter seem restrictive, the results may be applied to a much larger class of situations [Rosenthal1981].

The semi-join operation is used to reduce the size of a relation before transmitting it to the result site where the joins are computed. A join is computed by simply concatenating matching tuples on the joining attribute. All the data transmissions used for reducing a relation and the final transmission of the reduced relation to the result site form a **schedule for a relation.** Fig. 3.22 shows such a schedule. By considering attributes as projected relations, schedules for attributes are defined as well. A **schedule for a query** consists of the collection of the schedules for the relations that are used in the query and that do not reside at the result site.

Although the tuples of a relation are transmitted we will speak of the transmission of a relation, and the same for attributes.

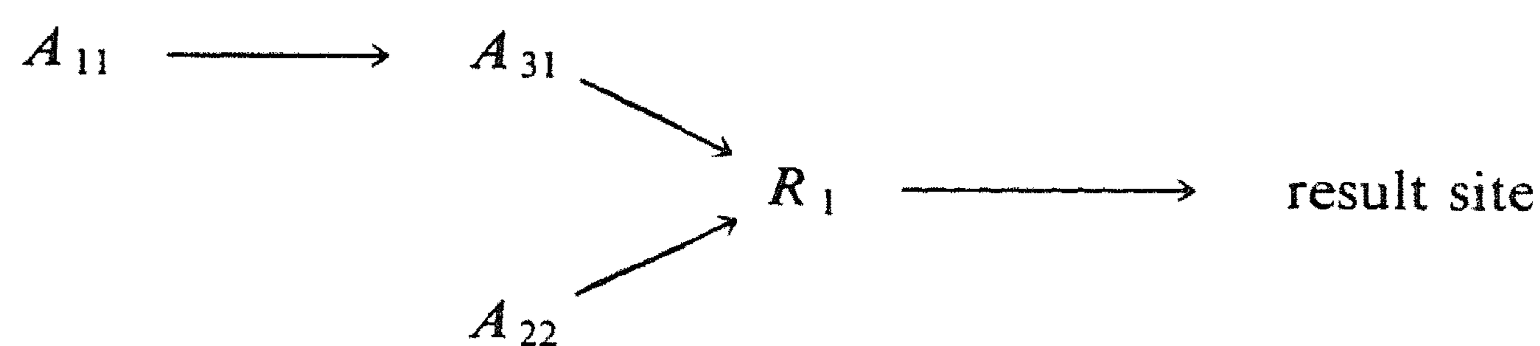


Figure 3.22. Schedule for relation R_1.

In general, more than one semi-join may be applied to a relation, or a semi-join is computed between a relation and an intermediate result that is already the result of another semi-join. Therefore, we define the **incoming selectivity** of a schedule for a relation as the product of the selectivities of all attributes in the schedule where the selection of each attribute counts only once and where the selection of the attributes of the relation itself do not count at all. For example, the incoming selectivity of relation R_1 in fig. 3.22 is $p_{31} \times p_{22}$. The size of the relation to be transmitted times the incoming selectivity of its schedule is the size of the reduced relations. Note, the selectivities caused by local predicates have already been applied.

We assume that the size of intermediate results are correctly estimated by using selectivities (**Selectivity Assumptions**).

3.4.2. Simple and General Queries

To start with local processing, is, in general, the best thing to do in a distributed environment. With this we mean the computation of restrictions and projections and, if more than one relation reside at the same site, semi-joins. Logically, the relations at one site are viewed as a single relation, although the Cartesian product between them is not computed. As a consequence all the algorithms assume that only one relation occurs per site. After initial local processing, the following parameters result:

m : number of relations in the remaining query,
α_i : number of attributes in relation R_i,
β_i : number of joining attributes in relation R_i.

In [Hevner1979a] the algorithms *PARALLEL* and *SERIAL* were introduced and investigated. These algorithms produce minimum response transmission time and minimum total transmission time schedules, respectively, for simple queries. **Simple queries** contain, after initial local processing, relations consisting of only one and the same attribute, the joining attribute. Thus $\alpha_i = \beta_i = 1$, for $i = 1, 2, \ldots, m$.

Because our results in the next subsection make use of these algorithms we will briefly elaborate on them. Algorithm *PARALLEL* (shown in fig. 3.23) creates for every relation, R_i, a schedule that forms the parallel integration of schedules for other relations. The destination site for the parallel schedules is the site where R_i resides. At this site the semi-joins with R_i are computed, and the result R_i is transmitted to the result site. Algorithm *SERIAL* (fig. 3.24) constructs a schedule that consists of the transmission of the smallest to the second smallest, etc., and finally to the result site. Because one of the relations may reside at the result site the algorithm considers a second alternative to obtain an optimal schedule. In both algorithms the struct **relation** contains all the parameters of subsection 3.4.1 needed to compute the cost of the schedules.

The complexity of algorithm *PARALLEL* is $O(m^2)$ and that of *SERIAL* $O(m \log m)$, where m is the number of relations involved in the query [Hevner1979b].

Example 3.7

As an example database throughout this section we will use the database of a real-estate agent. It consists of a relation *SELLER*, which contains information about sellers and the identification codes of the properties they want to sell. The relation *PROP* supplies information about the property that is for sale. The relation *SALE* is concerned with the contract between a buyer and a seller. The three relations and their attributes are listed below:

SELLER(*SNAME*,*ADDRESS*,*CITY*,*PROP#*),
PROP(*PROP#*,*TYPE*,*LOCATION*),
SALE(*BNAME*,*SNAME*,*PROP#*,*DATE*).

```
proc PARALLEL=([]relation R;site S_r)[]schedule:
begin
   int m = upb R;
   [1:m]schedule sch;
   order relations such that R[1].s ≤ R[2].s ≤ . . . ≤ R[m].s;
            {R[i].s is the size of relation R_i}
   for i to m
   do
      construct a schedule for R_i consisting of the transmission of R_i to
      the result site;
      for j to i − 1
      do
         construct integrated schedule for R[i] to S_r
         consisting of parallel schedules for R[1], . . . , R[j]
      od;
      sch[i] := schedule with minimum RTT among these i schedules
   od;
   sch
end
```

Figure 3.23. Algorithm *PARALLEL*.

```
proc SERIAL=([]relation R;site S_r)schedule:
begin
   int m = upb R;
   order relations such that R[1].s ≤ R[2].s ≤ · · · ≤ R[m].s;
   consider both schedules:
            R_1 → R_2 → · · · → R_m → S_r
   and, if R_r resides at the result site, S_r
            R_1 → R_2 → · · · → R_{r-1} → R_{r+1} → · · · → R_m → S_r;
   schedule with smallest TTT
end
```

Figure 3.24. Algorithm *SERIAL*.

The query stated at a site other than the ones where the three relations are located:

> *Give the identification numbers of properties, located in the Santa Cruz Mountains, that have been sold after 1970 by people living in San Jose.*

The parameters of the relations after local processing are shown in table 3.25.

relation	$A_{i1} = PROP\#$	
	b_{i1}	p_{i1}
$R_1 : SALE$	800	0.8
$R_2 : SELLER$	400	0.4
$R_3 : PROP$	300	0.3

Table 3.25. Parameters after local processing.

Algorithm *PARALLEL* constructs schedules for the relations in ascending order of their sizes. The schedule for *PROP* is:

$$A_{31} \xrightarrow{300} \text{result site}$$

$$RTT = 300$$

For *SELLER* the following two alternatives are considered:

$$A_{21} \xrightarrow{400} \text{result site}$$

$$RTT = 400$$

$$A_{31} \xrightarrow{300} A_{21} \xrightarrow{120} \text{result site}$$

$$RTT = 420$$

The first one is chosen because its *RTT* is smallest. For *SALE* three schedules are constructed:

$$A_{11} \xrightarrow{800} \text{result site}$$

$$RTT = 800$$

$$A_{31} \xrightarrow{300} A_{11} \xrightarrow{240} \text{result site}$$

$$RTT = 540$$

$$\left.\begin{array}{l} A_{31} \xrightarrow{300} \\ A_{21} \xrightarrow{400} \end{array}\right\} A_{11} \xrightarrow{96} \text{result site}$$

$$RTT = 496$$

The last one shows the parallel integration of the schedules for *PROP* and *SELLER*. Because all relations are part of this schedule and its *RTT* is smallest among the ones for *SALE* it is the schedule for the query.

Algorithm *SERIAL* only considers the first alternative because none of the relations is located at the result site. The computed schedule is

$$A_{31} \xrightarrow{300} A_{21} \xrightarrow{120} A_{11} \xrightarrow{96} \text{result site}$$

$$TTT = 416$$

□

If after the initial local processing not all α_i's are 1 the query contains either target attributes that are no joining attributes, or it contains several joining attributes. The resulting query is then called a **general query** and is characterized by $\alpha_i \geq \beta_i \geq 1$, for $i = 1, 2, \ldots, m$. In the next two subsections schedules for minimizing response transmission time and total transmission time are constructed.

3.4.3. Minimizing Response Transmission Time

The response transmission time of a query will be obtained by minimizing the response transmission times for each of the relations. So, again the Parallelism Assumption (see subsection 3.1.5) is used, which states that parallel schedules do not influence each other's response transmission time. This means that the relations can be treated separately.

In general, a relation R_i will, after initial local processing, consist of target and joining attributes. The attributes are renumbered such that all joining attributes come first and then the target attributes that are no joining attributes. This numbering is done such that the jth joining attribute of R_i and R_k are the same. Before transmitting relation R_i to the result site it is as much as possible reduced in size such that the response transmission time of its schedule is minimized. This is done by computing semi-joins with the joining attributes of R_i. Let one of them be A_{ij}, then the unique attribute values u_{kj} of other relations R_k are sent to R_i. Because we want to compute the minimum response transmission time of R_i we are interested also in the minimum response transmission time schedules for A_{kj}. Therefore, we define for each attribute A_{pq} a simple query. The relations involved in such a query are the relations in the original query projected on the attribute A_{pq}. The algorithm *PARALLEL* then computes minimum response transmission time schedules for transmitting relations consisting of only one joining attribute to the result site (in this case the site of R_i).

Algorithm *RESPONSE GENERAL* (fig. 3.26) orders the schedules of all A_{kj} with respect to their minimum response transmission time. Then it constructs a schedule for R_i for each A_{kj}. This schedule consists of the parallel integration of the schedules of all A_{pq} with a minimum response transmission time less than or equal to the one of A_{kj}, and the transmission of the reduced R_i to the result site. Among all schedules for R_i the one with the smallest response transmission time is chosen.

Example 3.8

We will use the same database as in example 3.7. A minimum response transmission time schedule is constructed for the query shown in fig. 3.27:

> *Give the names and addresses of sellers that live in Los Angeles, and the type of property that were sold after 1975.*

We assume that the relations are located at different sites other than the result site. The first part of the processing schedule will consist of the restrictions

```
proc RESPONSE GENERAL=([]relation R)[]schedule:
begin
   int m = upb R;
   [flex] schedule candsch, intsch, sch;
   define for each joining attribute A_ij a simple query with arbitrary
   result site;
   apply algorithm PARALLEL to each simple query(candsch);
        {candsch contains the schedules for all joining attributes}
   for i to m
   do
      put the schedules that have the same joining attributes as R[i] at
      the beginning of candsch in ascending order on their RTT's, say
      there are l of them;
      for j from 0 to l
      do
         intsch[j] := integrated schedule for R[i] consisting of the
         parallel schedules candsch[1], . . . , candsch[j], and compute
         transmission cost of R[i] by means of the incoming selectivity
      od;
      sch[i] := schedule in intsch with smallest RTT
   od;
   sch
end
```

Figure 3.26. Algorithm *RESPONSE GENERAL*.

CITY = Los Angeles and *DATE* > 1975. After this local processing we assume that the data shown in table 3.28 results.

relation R_i	size	A_{i1} = *PROP#*		A_{i2} = *SNAME*	
		b_{i1}	p_{i1}	b_{i2}	p_{i2}
R_1 : *SALE*	10000	1400	0.7	1000	0.8
R_2 : *SELLER*	6000	400	0.2	900	0.6
R_3 : *PROP*	5000	800	0.4		

Table 3.28. Parameters after local processing.

For each joining attribute, *PROP#* and *SNAME*, a simple query is defined. Algorithm *PARALLEL* is applied to both of them resulting in the following schedules:

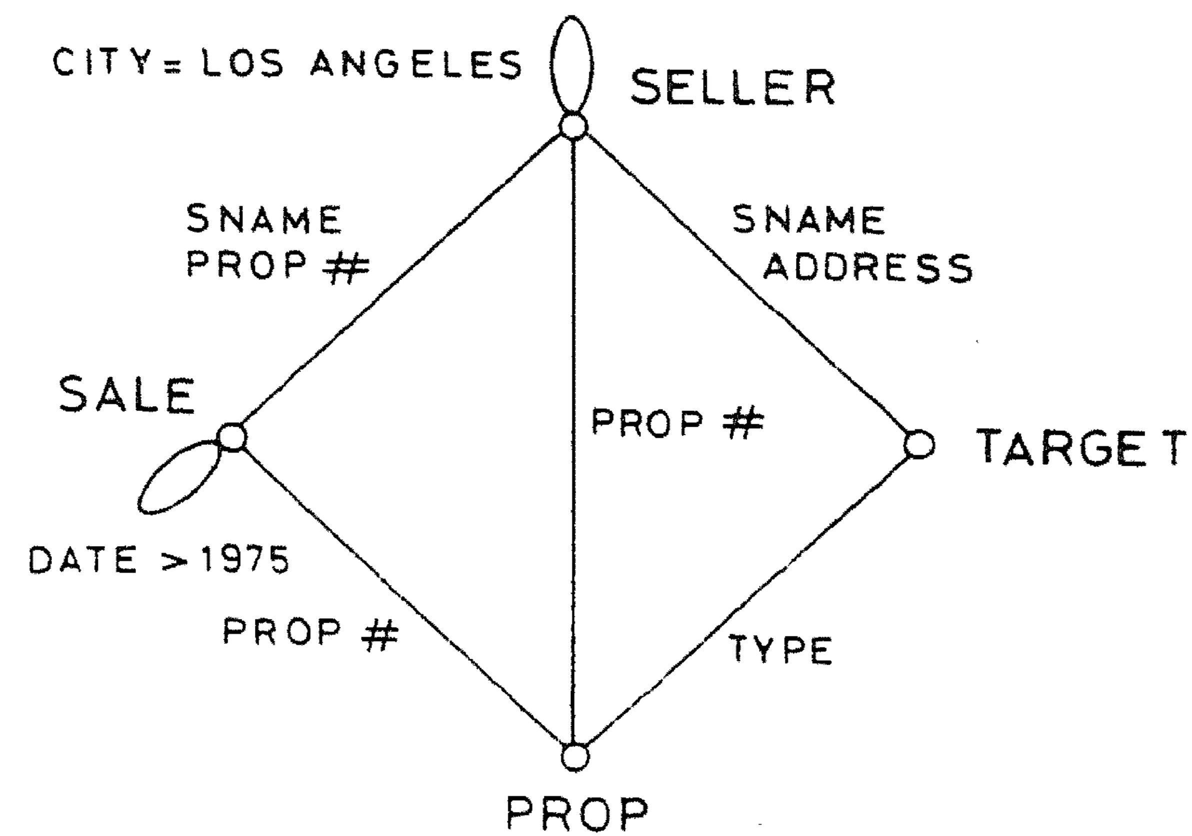


Figure 3.27. Graphical representation of example query.

PROP#

$A_{21} \xrightarrow{400}$ arbitrary site

$A_{31} \xrightarrow{400} A_{31} \xrightarrow{160}$ arbitrary site

$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112}$ arbitrary site

SNAME

$A_{22} \xrightarrow{900}$ arbitrary site

$A_{12} \xrightarrow{1000}$ arbitrary site

The construction of the schedule for relation *SELLER* will be discussed in detail. Schedules whose last transmission consist of attribute values of *SELLER* itself do not need to be considered. The remaining schedules are ordered on their *RTT*. Table 3.29 shows this.

A_{ij}	*RTT*
A_{31}	560
A_{11}	672
A_{12}	1000

Table 3.29. *RTT*s of the candidate schedules.

For each of the attributes A_{ij} in table 3.29 a schedule for *SELLER* is constructed consisting of the integration of the parallel schedules of the other attributes in the table whose *RTT* is less than or equal to the *RTT* of A_{ij}. Among all these schedules for *SELLER* the one with minimum response transmission time is chosen.

attribute A_{31}

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} SELLER \xrightarrow{2400} \text{result site}$$

$$\begin{aligned} RTT &= TT(400) + TT(0.2 \times 800) + TT(0.4 \times 6000) \\ &= 400 + 160 + 2400 \\ &= 2960 \end{aligned}$$

attribute A_{11}

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} SELLER$$
$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} SELLER$$
$$SELLER \xrightarrow{1680} \text{result site}$$

$$\begin{aligned} RTT &= \max(TT(400) + TT(0.2 \times 800), \\ &\qquad TT(400) + TT(0.2 \times 800) + TT(0.2 \times 0.4 \times 1400) \\ &\quad + TT(0.4 \times 0.7 \times 6000) \\ &= 400 + 160 + 112 + 1680 \\ &= 2352 \end{aligned}$$

attribute A_{12}

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} SELLER$$
$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} SELLER \xrightarrow{1344} \text{result site}$$
$$A_{12} \xrightarrow{1000} SELLER$$

$$\begin{aligned} RTT &= \max(TT(400) + TT(0.2 \times 800), \\ &\qquad TT(400) + TT(0.2 \times 800) + TT(0.2 \times 0.4 \times 1400), \\ &\qquad TT(1000)) \\ &\quad + TT(0.4 \times 0.7 \times 0.8 \times 6000) \\ &= 1000 + 1344 \\ &= 2344 \end{aligned}$$

The last schedule has the smallest response transmission time and is chosen as schedule for *SELLER*. Note, the parallel schedule $A_{21} \rightarrow A_{31} \rightarrow SELLER$ can be omitted because it does not reduce the size of *SELLER*.

The minimum response transmission time schedule for the other relations are:

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} SALE \xrightarrow{800} \text{result site}$$

$$RTT = 1360$$

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} PROP \xrightarrow{700} \text{result site}$$

$$RTT = 1372$$

The response transmission time of the query is determined by the largest RTT of the relations, which is 2344.

□

Theorem 3.21 The complexity of algorithm *RESPONSE GENERAL* is $O(\delta m^2 \log_2 \delta)$, where m is the number of relations and δ is the number of joining attributes in the query.

Proof The computation of the candidate schedules for the δ joining attributes takes $O(\delta m^2)$. There are at most m candidate schedules per joining attribute. To put them in ascending order on their RTT is the same as merging δ lists of length m, and this takes $O(\delta m \log_2 \delta)$. This has to be done for each relation, so the complexity is $\delta m^2 \log_2 \delta$.

□

Note, putting the schedules in array *candsch* in ascending order on their RTT can also be done once, namely before entering the for-loop of i. The resulting algorithm has a worst case complexity that is slightly better than the one of algorithm *RESPONSE GENERAL*, namely $O(\delta m^2)$. However, we expect that a relation, most of the time, will have fewer than δ joining attributes resulting in a better average case complexity of algorithm *RESPONSE GENERAL*.

The quality of the schedules produced by algorithm *RESPONSE GENERAL* depends on whether there is a correlation between two attributes of the same relation. If so, it might happen that a restriction on one attribute can decrease the size of the unique attribute value set of the other attribute, or that not every attribute value has an equal probability of being part of a tuple in the relation that results after the restriction. This is called **attribute dependence.** In [Pelagatti1979] an attempt has been made to investigate the cost of schedules without assuming attribute independence. It seems feasible to keep track of a small number of dependences. To consider all dependences might require a database as large as the original database.

Theorem 3.22 Under the Transmission Assumption, the Parallelism Assumption and the Selectivity Assumption and in the Equal Transmission Cost Model the algorithm *RESPONSE GENERAL* computes minimum response transmission time schedules for relation R_i, if there is attribute independence in relation R_i.

Proof We will show that to compute the minimum response transmission time schedule for R_i, it suffices to consider the type of integrated schedules computed by algorithm *RESPONSE GENERAL*.

Assume that we are given a minimum response transmission time schedule for R_i. Let S be this schedule without the final transmission of R_i. Then S is the

parallel integration of schedules for attributes. Say, the schedule for A_{kj} has the largest response transmission time to the site where R_i resides. The schedule for A_{kj} considered by *RESPONSE GENERAL* has a minimum response transmission time because it is computed by algorithm PARALLEL [Hevner1979b]. So, if the schedule for A_{kj} in the optimal schedule for R_i differs from the one produced by *PARALLEL* it can be replaced by the parallel integration of the schedules of all the attributes whose transmissions are part of the schedule for A_{kj}, without increasing the response transmission time of S. Hence, only the schedules for the attributes produced by algorithm *PARALLEL* have to be considered.

Algorithm *RESPONSE GENERAL* computes a schedule for R_i for every A_{pq} and when it constructs such a schedule it contains the schedules for all attributes whose response transmission time is less than or equal to the response transmission time of the schedule for A_{pq}. Hence, one of these constructed schedules contains the parallel schedules for at least as many attributes as S. Therefore, the incoming selectivity of the integrated schedule considered by the algorithm is at least as small. Thus, the transmission time of R_i to the result site is no larger than in the minimum response transmission time schedule.

So, the schedule considered by the algorithm has minimum response transmission time.

□

Corollary 3.23 Under the Transmission Assumption, the Parallelism Assumption and the Selectivity Assumption and in the Equal Transmission Cost Model the algorithm *RESPONSE GENERAL* computes a minimum response transmission time schedule for a query, if in all relations involved there exists attribute independence.

The algorithm presented in [Hevner1979a] does not compute optimal schedules and does not necessarily run in polynomial time, as was shown in [Apers1979a]. Also, the improved version does not always produce minimum response transmission time schedules, in spite of the claims [Hevner1979b].

To obtain the real response transmission time one should again serialize the parallel schedules. The effect of this on the response transmission time will depend on the structure of the query and the locations of the relations. For example, if two relations, located at different sites, are joined on two or more attributes it might happen that one relation will send all its attribute values to the other, which of course can not be done in parallel. Another example is, if two relations are located at the same site and both have to be sent to the result site.

3.4.4. Minimizing Total Transmission Time

When minimizing the total transmission time of a relational calculus query the same problems are encountered as discussed in the subsection on minimizing the total transmission time of a query on inverted lists. In [Hevner1979b] it was shown that this problem is NP-complete. Therefore, we will adopt a heuristic approach. In the first approach the schedules for the relations are determined separately, and the schedule for the query is obtained by **integrating** these schedules, such that **redundant transmissions** are removed. In the second approach the schedule for the query is obtained by considering the schedules for the relations all at once. The advantage of the latter is that more elaborate schedules can be considered which reduce the relations in size at low cost.

Let us first consider the transmission of one relation where semi-joins on only one joining attribute are allowed. The query may reference several relations but they will be treated independently. In algorithm *RESPONSE GENERAL* each attribute A_{ij} had its own schedule of which some were part of the integrated schedule for a relation. Here, a similar approach will be taken. Algorithm *SERIAL* produces a schedule, S, for a simple query. From this schedule we construct **prefix schedules.** A prefix schedule for attribute A_{kj} will consist of all transmissions of S which precede the transmission of A_{kj} and the transmission of A_{kj} itself. The transmission of A_{kj} has an undefined destination. When this prefix schedule is used in the schedule for relation R_i the site, where R_i resides, becomes the destination site. We have to realize that, although, schedule S is optimal for the corresponding simple query, the prefix schedule for A_{kj} is not necessarily an optimal schedule for the transmission of A_{kj}. For the transmission of R_i all these prefix schedules are considered in turn. The schedule for R_i consists of the prefix schedule of one A_{kj} with the final transmission of A_{kj} to R_i, followed by the transmission of R_i to the result site.

Some of these prefix schedules may contain the transmission of the values of the joining attribute of R_i, A_{ij}. The selectivity of this attribute does not count in the incoming selectivity of a schedule for R_i itself. Its only purpose is to reduce the transmission time of other attributes in the schedule. To ensure that the produced schedule for R_i has minimum total transmission time we also have to consider prefix schedules from which the transmission of A_{ij} has been deleted. Fig. 3.30 shows the described algorithm.

Now we will discuss the quality of the schedule for R_i produced by algorithm *simple TOTAL GENERAL* and the complexity of the algorithm.

Theorem 3.24 Under the Transmission Assumption and the Selectivity Assumption and in the Equal Transmission Cost Model the algorithm *simple TOTAL GENERAL* computes a schedule for R_i that has minimum total transmission time, if semi-joins on only one attribute are allowed.

Proof Let the joining attribute be the jth one for all relations. The optimal schedule for R_i will not consist of parallel schedules, for the same reason as the schedule produced by algorithm *SERIAL* does not contain parallel transmissions [Hevner1979b]. So, we may speak of the last attribute transmitted to R_i in the schedule. Say this attribute is A_{kj}. The order of the transmissions in the optimal schedule is determined by the sizes of the attributes. The smallest one is sent to the second smallest one, and so on. Again, this is based on the optimality of the schedules produced by *SERIAL.*

We will show now that all attributes whose sizes are less than the size of A_{kj} will be part of the optimal schedule except maybe for A_{ij}. Assume that attribute A_{pj} with a size smaller than that of A_{kj}, is not part of the optimal schedule. Let A_{qj} be the attribute in the optimal schedule which follows A_{pj} in size (i.e., $|A_{pj}| \leq |A_{qj}|$) and no other attribute in the schedule has this property. Then replacing A_{qj} by A_{pj} will not increase the total transmission time of the schedule because A_{pj} is not larger than A_{qj} and its selectivity is also not larger. This replacement can be repeated until A_{kj} should be replaced but this contradicts the choice of the optimal schedule.

The replacement argument does not apply to A_{ij}, an attribute of R_i, and attributes whose sizes and selectivities are the same as the ones of A_{kj}. Therefore, the

```
proc simple TOTAL GENERAL=([]relation R;int j)[]schedule:
begin
   int m = upb R;
   schedule S, S';
   [1:m]schedule sch;
   [1:m]relation sq;
              {the jth joining attribute defines a simple query}
   for i to m
   do put A_ij in sq od;
   S := SERIAL(sq);
   for i to m
   do
      S' := delete transmission of A_ij from S;
      for s in {S,S'}
      do
         construct for each prefix schedule from s a schedule for relation
         R[i]
      od;
      sch[i] := schedule for R[i] with the smallest TTT among the
      above constructed ones, or the schedule only consisting of the
      transmission of R[i] to the result site, whichever is smallest
   od;
   sch
end
```

Figure 3.30. Algorithm *simple TOTAL GENERAL*.

algorithm *simple TOTAL GENERAL* must consider all prefix schedules, with and without the transmission of A_{ij}. Hence, the minimum total transmission time schedule for R_i is among the ones considered by the algorithm.

□

Theorem 3.25 The complexity of the algorithm *simple TOTAL GENERAL* is $O(m \log_2 m)$.

Proof Based on the complexity of algorithm *SERIAL*, which is $O(m \log_2 m)$.

□

Example 3.9

For each joining attribute, *PROP#* and *SNAME*, a simple query is constructed and algorithm *SERIAL* is applied to both of them resulting in the following two schedules:

attribute *PROP#*

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} \text{arbitrary site}$$

attribute *SNAME*

$$A_{22} \xrightarrow{900} A_{12} \xrightarrow{600} \text{arbitrary site}$$

The construction of a schedule for relations *SALE* and *SELLER* based only on semi-joins with attribute *SNAME* is shown in detail. For relation *SALE* only the prefix schedule consisting of the transmission of A_{22} is of interest. The resulting schedule for *SALE* is

$$A_{22} \xrightarrow{900} SALE \xrightarrow{6000} \text{result site}$$

with total transmission time 6900.

For relation *SELLER* the complete schedule produced by *SERIAL* is considered and also this schedule with the transmission of A_{22}, which is part of *SELLER*, deleted. The two schedules and their total transmission times are:

$$A_{22} \xrightarrow{900} A_{12} \xrightarrow{600} SELLER \xrightarrow{4800} \text{result site}$$

$$TTT = 6300$$

$$A_{12} \xrightarrow{1000} SELLER \xrightarrow{4800} \text{result site}$$

$$TTT = 5800$$

The second alternative is the best one.

The schedules for the relations based only on semi-joins with attribute *PROP#* are:

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} SALE \xrightarrow{800} \text{result site}$$

$$TTT = 1360$$

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} SELLER \xrightarrow{1680} \text{result site}$$

$$TTT = 2352$$

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} PROP \xrightarrow{700} \text{result site}$$

$$TTT = 1372$$

□

In general, a query will contain joins on several attributes. If a relation contains several joining attributes it can be reduced in size by computing semi-joins on all of them. Under assumption of attribute independence the selectivity of one joining attribute has no effect on the other ones; the schedule for a relation may be the integration of the parallel schedules for different joining attributes. Note, that this is the first time that parallelism is allowed. The construction of an integrated schedule for a relation R_i, involves a choice among all prefix schedules of all the joining attributes. If δ is the number of joining attributes in the query, this can be done in $O(m^{\delta})$, which for small δ is feasible. Here, we will give a more efficient algorithm that only manipulates the schedules produced by algorithm *simple TOTAL GENERAL* for each joining attribute.

Algorithm *TOTAL GENERAL* considers for relation R_i the schedules produced by algorithm *simple TOTAL GENERAL* for each of its joining attributes in ascending order of their total transmission time. For each such schedule a parallel integration is constructed with other schedules whose total transmission times, including the final transmission of R_i, are smaller. Among these constructed schedules the one with the smallest total transmission time is chosen.

To compute the total transmission time of a query we may simply add the ones of the relations. However, the schedule for the query may contain many superfluous transmissions because the schedules for the relations were computed independently. Also, because of the same reason, a transmission in a prefix schedule for an attribute may go to a site containing a relation and the selectivity of the mentioned schedule is not applied to the relation. This kind of selectivity is called coincidentally available selectivity. Applying this selectivity to the relation will reduce it further in size and, therefore, decrease the total transmission time. All these matters are taken care of when computing the total transmission time of a query.

Note, that removing identical transmissions means moving forking points to the right of the time scale of the schedule.

Theorem 3.26 The complexity of the algorithm *TOTAL GENERAL* is $O(\delta m(\log_2 \delta + m \log_2 m))$, where m is the number of relations and δ is the number of joining attributes in the query.

```
proc TOTAL GENERAL=([]schedule R)schedule:
begin
   int m = upb R;
   [1:m]schedule sch;
   [1:m,1:δ]schedule candsch;
   for j to δ
   do candsch[,j] := simple TOTAL GENERAL(R,j) od;
   for i to m
   do
      put schedules in candsch[i,] in ascending order on their TTT;
      for j to α_i
      do
         construct integrated schedule for R[i] consisting of the parallel
         schedules candsch[i,1], . . . , candsch[i,j]
      od;
   od;
   remove superfluous transmissions(sch)
end
```

Figure 3.31. Algorithm *TOTAL GENERAL*.

Proof In the for-loop for j at most δ times the algorithm *simple TOTAL GENERAL* is applied, which takes $O(\delta m \log_2 m)$. Ordering the schedules takes $O(\delta \log_2 \delta)$. Hence, the theorem follows.

□

Example 3.10

The final schedule for a relation is obtained by integrating the parallel schedules for each joining attribute and selecting the one with minimum total transmission time. For the schedule for relation *SALE* two alternatives are considered:

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} SALE \xrightarrow{800} \text{result site}$$

$TTT = 1360$

$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160}$ *SALE* $\xrightarrow{480}$ result site

$A_{22} \xrightarrow{900}$ *SALE*

$TTT = 1940$

The first alternative is chosen as schedule for *SALE*. The schedules for the other relations are:

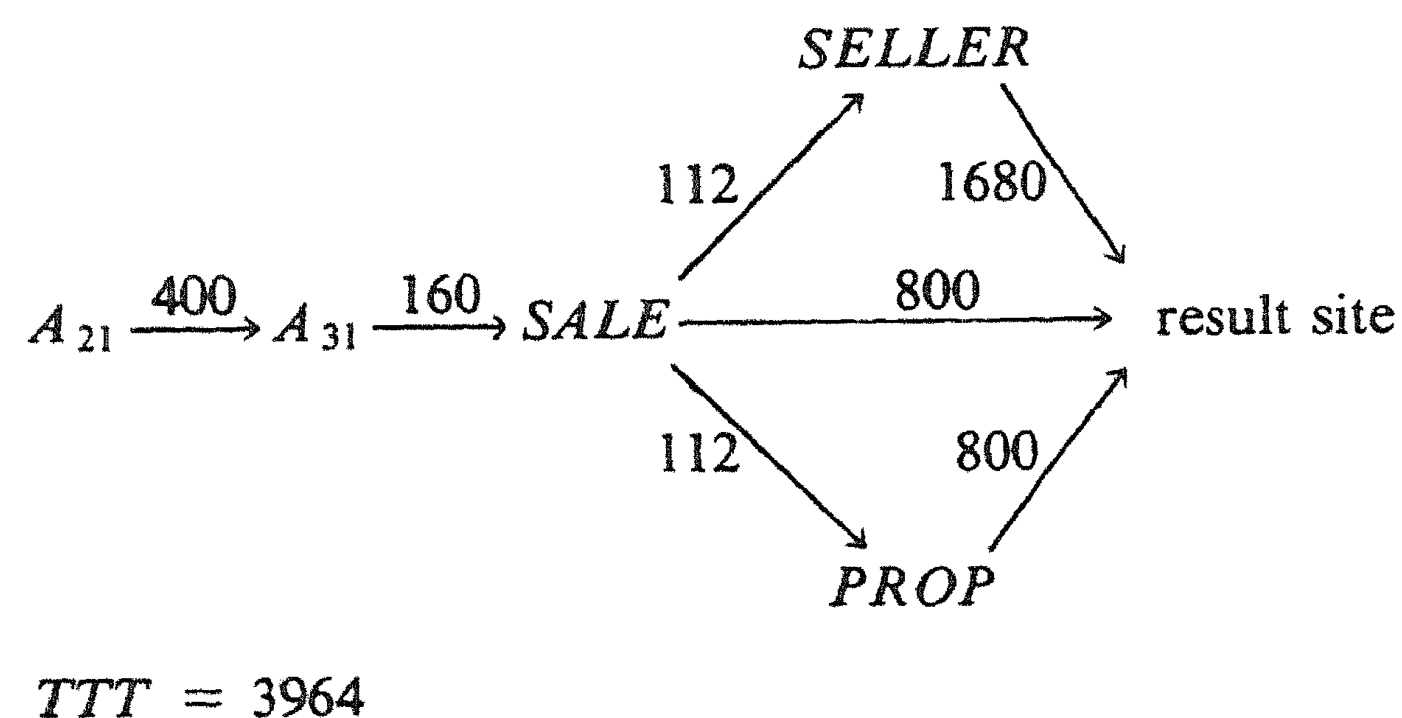
$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} SELLER \xrightarrow{1680} \text{result site}$$

$TTT = 2352$

$$A_{21} \xrightarrow{400} A_{31} \xrightarrow{160} A_{11} \xrightarrow{112} PROP \xrightarrow{700} \text{result site}$$

$TTT = 1372$

The schedule for the query is now obtained by removing all redundant transmissions:



$TTT = 3964$

Note: no transmissions of the joining attribute *SNAME* are part of the schedule because its selectivity is too high and it is too costly to transmit.

□

How many superfluous transmissions there are in the integrated schedule for the query depends on many factors. One of the negative influences is the deletion of the transmission of A_{ij} in the schedule for R_i, because it prevents the moving of forking points further to the right on the time scale. A way to overcome this kind of problem is to guarantee that the forking points are at the end of a schedule for a joining attribute, just before the last transmission, which is sent to all relations. Langefeld and Swart came up with a variation [Apers1981c], called *COLLECTIVE*, which nicely shows this. We will explain it for only one joining attribute. The joining attributes of all the relations together form a simple query. Applying algorithm *SERIAL* will produce one schedule and the last transmission will have an undefined destination. This schedule will be used for all relations, or, which is exactly the same, the destination is made equal to the set of sites where the relations reside. So, the whole schedule, except for the last transmission, is shared by all relations. This means that just before the last transmission, there is a forking point.

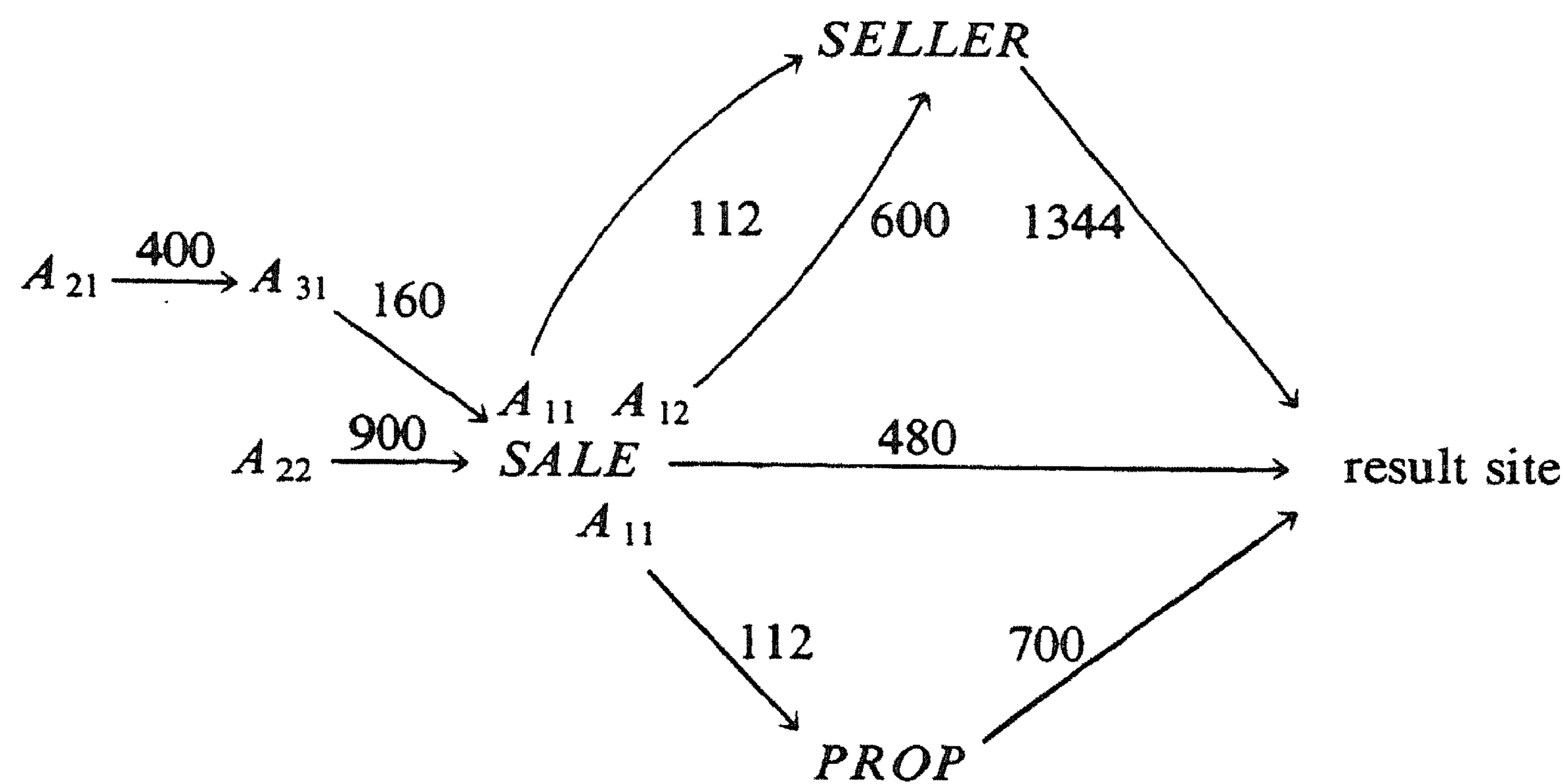
For small relations even this last transmission may be too expensive. Therefore, the schedule of the query may be modified slightly by dropping this last transmission to such a relation, if the total transmission time decreases. This is checked for all relations. If it happens that all relations prefer to drop the last transmission, the whole schedule is not used by any of the relations and can be

deleted completely. If only one joining attribute is considered this is not very likely. Shortening the schedule was not considered to keep the algorithm simple.

If the query contains several joining attributes the algorithm *SERIAL* is applied to each of them. Each of the schedules produced will direct its last transmission to all of the relations. Again, it is checked whether the last transmission is more costly than it reduces the relation to which it is sent. Here, small relations are more common because a relation may have been reduced by other attributes and, therefore, deletion of transmissions or even whole schedules will be more common.

A comparison between the algorithms is given in section 3.5

Example 3.11 The schedules produced by algorithm *SERIAL* for the two simple queries of *SNAME* and *PROP#* are part of the schedules for all three relations. The initial schedule for the query is:



$TTT = 4976$

Algorithm *COLLECTIVE* tries to remove the transmission of A_{12} to *SELLER*. This decreases the *TTT* with 600 but it also increases the *TTT* with 336 because *SELLER* will be less reduced. The net result is negative so this transmission is removed. The removal of the transmission of A_{11} to *SELLER* increases the net result of *TTT* and is therefore not carried out. Again, the removal of the transmission of A_{22} to *SALE* decreases the *TTT* with 580. Other removals increase the *TTT*. The resulting schedule is the same as in example 3.10.

□

3.4.5. Summary

The basic ideas obtained for the inverted file organization were applied to queries in the relational data model. The join operation, which may produce a large result, is computed at the result site and the semi-join operation is used to reduce relations in size before transmitting them to the result site. Both minimizing response transmission time and total transmission cost were investigated. The algorithms proposed make use of the algorithms introduced by A.R. Hevner for simple queries.

For minimizing response transmission time optimal schedules can be obtained under the Transmission Assumption, the Parallelism Assumption and the Selectivity Assumption. The unique values of joining attributes of certain relations are

transmitted to other relations to compute semi-joins. After a relation has been reduced in size it is sent to the result site. To be able to compute the joins the joining attributes of a relation have to be sent as well, although they are not necessarily target attributes.

For minimizing total transmission cost two approaches are investigated. One approach computes a schedule for each relation, independent of the others, and then integrates these schedules into a schedule for the query. During integration redundant transmissions are removed. The other approach tries to construct schedules for each joining attribute, which are used in the schedule for all relations. By local optimization some of the transmissions or even complete schedules are dropped if they cost too much compared to the reduced cost to transmit the relations. In the next section the algorithms for minimizing total transmission cost are compared with algorithms that decompose the query at the logical level.

3.5. Comparison Between Distributed Query Processing Algorithms

The purpose of the comparison between the algorithms is not so much to show that one is definitely better than the other, but to get a better understanding of distributed query processing. Therefore, a common processing model is required that may differ in certain aspects from the original one used by each method.

The query processing model is based on the **selectivity theory** used by [Hevner1979b] (see also subsection 3.4.1). This means that for every attribute of every relation the selectivity and the size are known. This not only makes it possible to compute the size of a result after a semi-join has been applied, but also the size of a join. For example, $R_1(A = A)R_2$ is a join between R_1 and R_2 on attribute A, with domain D. Under the assumption of subsection 3.4.1 the expected number of tuples of R_1 with a particular attribute value for A is $|R_1| \,/\, |D|$. This is also true for R_2. Therefore, for each value in D there are $|R_1| \times |R_2| \,/\, |D|^2$ tuples in the join. Hence, the expected size of the join is

$$\frac{|R_1| \times |R_2|}{|D|}.$$

In [Rosenthal1981] it was shown that even under some less restrictive assumptions this expectation is reasonable. The selectivity of attribute A after the join is the product of the selectivities of attribute A in R_1 and in R_2. To estimate the result after a projection we assume that one of the remaining attributes is a key-attribute or nearly a key-attribute. So, the number of tuples remains the same, only the size becomes smaller. See also [Epstein1979] for a discussion about perfect information to determine the sizes of intermediate results.

The unit of allocation is assumed to be a complete relation. Before optimization a non-redundant **materialization** is determined; we assume that each relation is stored at a different site. As **cost function** the total transmission cost is used.

The algorithms that are compared belong to two families, the **semi-join family** and the **decomposition family**. Our goal is to show the usefulness of the semi-join operation and the integration of this operation in a decomposition process.

One of the members of the semi-join family is algorithm *TOTAL* (*GENERAL*). In subsection 3.4.4 it was shown that, if the relations were considered separately, the deletion of certain transmissions would decrease the total transmission time. However, one might expect that this deletion means that fewer redundant transmissions

can be removed when considering the transmission of all relations. *TOTAL** will be the version of *TOTAL* where the deletion discussed in subsection 3.4.4 is omitted. Also, the algorithm *COLLECTIVE* is considered.

For the decomposition family the algorithm of Epstein is used, and will be called *EXHDECOM* (exhaustive decomposition). This algorithm does not produce schedules that deliver the result to a specified result site. Therefore, in general, the cost of transmitting the result to the result site have to be added to the cost of the processing schedule obtained. This will increase the total cost enormously if the result is large. A comparison which is more fair with respect to the other algorithms is obtained by taking this final transmission into account as well. This version is denoted by *EXHDECOM**.

EXHDECOM is ideal to investigate the integration of a decomposition process and the use of the semi-join operation as far as the produced schedules are concerned, simply because it investigates all possible decompositions. The algorithm selects a subset of the relations in the query and computes the cost of processing the corresponding subquery. This processing amounts to sending all the relations in the subset chosen to the largest one where the joins are computed. Instead of simply transmitting the relations we could apply algorithm *TOTAL* or any other member of the semi-join family with as result site the site where the largest relation resides. This is not feasible because of the time consuming character of *EXHDECOM*. Therefore, to show the usefulness of the semi-join operation we consider the following. Compare the cost of directly sending a relation R_i to the largest relation R_l of the chosen subset, with the alternative where first a semi-join on R_i is applied and then the reduced R_i is sent to the site of R_l. The semi-join is computed by sending the unique values of the joining attribute of R_l to the site of R_i. If there is no joining clause between R_i and R_l then this alternative is not applicable. This version will be called *SJ EXHDECOM* (semi-join *EXHDECOM*). The cost of the schedules of this version are of course not as good as if algorithm *TOTAL* were applied to the subqueries. *EXHDECOM* together with *TOTAL* produces schedules that can be no worse than merely applying *TOTAL*.

The schedules produced by the algorithms are compared on their total transmission cost. The algorithms are applied to randomly generated queries based on the following parameters:

- number of relations per query (m)
  This parameter ranges from 2 to 5. For each of these values exactly 20 queries were generated, resulting in 80 queries for each test run.
- number of joining attributes per query (*join attr*)
  The number of joining attributes is a random number between 1 and 4, but never larger than the number of joining clauses (see below).
- number of target attributes per query (*targ attr*)
  The number of target attributes is a random number between 1 and 40; some of these may be joining attributes.
- number of joining clauses per query
  To make the query connected, this parameter should at least be equal to the number of relations minus one. For each joining clause a joining attribute is chosen and two relations are selected. The choice of relations is such that the query is connected. The maximum value of this parameter is kept small (the

number of relations plus two), because applying transitivity will increase the number of clauses.

- ☐ selectivity of an attribute (p)
  For each joining attribute of each relation a random number between between 0 and 1 is chosen, which denotes its selectivity.
- ☐ domain size of an attribute ($|D|$)
  The size is drawn from a negative exponential distribution with as an average 25. Queries in which for a particular relation the product of the selectivity and the domain size of an attribute is less than 1, is rejected.
- ☐ attribute size of an attribute
  The number of bytes in an attribute is a random number between 2 and 100.
- ☐ number of tuples per relation
  To guarantee the selectivity for each joining attribute, $|R_i| \geqslant |D_{ij}| p_{ij}$. On the other hand, R_i may not contain more tuples than $\prod_j |D_{ij}| p_{ij}$, where j ranges over all attributes in R_i, including the target ones. The number of tuples of R_i is chosen at random within the denoted limits.
- ☐ location of relations
  Each relation is assumed to reside at a different site other than the result site.

Five parameters are chosen to investigate the effect of their value on the produced schedules by the algorithms, namely the number of relations, the number of joining attributes, the number of target attributes, the cardinality of the domains, and the selectivity.

The average cost of the schedules produced by each of the algorithms are shown in table 3.32. For particular values of the five parameters 80 queries were generated, resulting in 1520 queries.

family	algorithm	average cost in bytes	average CPU-sec. required.
semi-join	*TOTAL*	12081	0.03
	*TOTAL**	11960	0.03
	COLLECTIVE	11700	0.02
decomposition	*EXHDECOM*	10353 + $\|R_r\|$	0.58
	SJ EXHDECOM	6818 + $\|R_r\|$	0.82
	*EXHDECOM**	11574	9.50

Table 3.32. Average cost of a schedule in bytes and the amount of CPU-seconds required to compute it; $|R_r|$ is the size of the result relation.

The results of the algorithms in the semi-join family are close to each other. We may conclude, however, that *COLLECTIVE* performs better than the other two

at less cost in terms of CPU-seconds. Because a detailed description of *TOTAL* can be found in this thesis we use it to represent the semi-join family.

Within the decomposition family the results are difficult to compare. The reason for this is that both *EXHDECOM* and *SJ EXHDECOM* produce schedules that do not include the final transmission to the site where the result should be delivered. *EXHDECOM**, on the other hand, takes this transmission into account as well. The difference between *EXHDECOM* and *SJ EXHDECOM* is caused by the usage of the semi-join operation. Although only a straightforward application of the semi-join operation was implemented, already a 34% improvement was obtained. This could be further improved if a member of the semi-join family was used. Ideally, this operation should also be used in *EXHDECOM**, however, this would make the algorithm extremely costly. We merely want to point out that the usage of a semi-join operation in a decomposition process has a positive effect (a similar conclusion is reached in [Bernstein1981c]). The algorithm *EXHDECOM** is chosen to represent the decomposition family.

However, one should keep in mind that the results produced by this algorithm are obtained at a very high cost compared to the ones of the algorithms in the semi-join family, and its application may, therefore, not be suitable for ad hoc queries. Also, any decomposition algorithm that runs in a time comparable to the one of a semi-join algorithm, will most likely produce worse results than *EXHDECOM** [Epstein1979, Epstein1980b].

To understand the behaviors of the algorithms in the semi-join and decomposition families, the two representatives are compared in more detail.

The number of relations (m) in the query is varied from 2 to 5. The cost of the *TOTAL*-schedules decreases at first because on the one hand there are more relations but on the other hand there are more (redundant) transmissions causing a higher chance of coincidentally available selectivity. The former effect wins if m increases. The more relations the more irreducible components there will be, giving *EXHDECOM** a better chance to decompose the queries, which decreases the cost of the schedules.

m	2	3	4	5
TOTAL	14077	8976	10779	10946
EXHDECOM*	15508	10092	10346	10540

Table 3.33. Varying the number of relations.

Increasing the number of joining attributes (*join attr*) decreases the chance that several clauses in the query involve the same joining attribute because the number of joining clauses ranges between $m - 1$ and $m + 2$. This implies that the effectiveness of applying semi-joins decreases, showing an increase of the cost of the schedules. *EXHDECOM**, on the other hand, starts to take advantage of the small intermediate results.

A larger number of target attributes (*targ attr*) means larger results after a join. For a small number of target attributes *TOTAL** clearly has a disadvantage, because

join attr	1	2	3	4
TOTAL	7280	13009	11049	13043
*EXHDECOM**	9900	13343	10465	10807

Table 3.34. Varying number of joining attributes.

all relations have to transmit their joining attributes to the result site although most of these attributes are not part of the result. For increasing number of target attributes this effect disappears.

targ attr	1	10	20	40
TOTAL	4782	8000	9364	14529
*EXHDECOM**	2903	10369	10546	11744

Table 3.35. Varying the number of target attributes.

The size of D, a domain, has a direct influence on the sizes of the relations. Small $|D|$ means small relations and a large $|D|$ means large relations. This explains why the cost becomes larger if $|D|$ increases. The estimated size of a join gets smaller if $|D|$ grows. This effect can be noted by the lower increase of the cost of the schedules produced by $EXHDECOM^*$.

$\|D\|$	10	25	50	75
TOTAL	7596	10555	15661	23989
*EXHDECOM**	11441	10285	14871	21823

Table 3.36. Varying the domain size.

The application of semi-joins is too expensive if the selectivities (p) are high (close to 1), as we can see from the results of $TOTAL$.

Overall we may conclude that if intermediate results are small $EXHDECOM^*$ outperforms $TOTAL$. Algorithm $TOTAL$ performs well if relations can be reduced

p	0.0-0.33	0.33-0.66	0.66-1.0
TOTAL	1765	9375	32465
EXHDECOM*	4619	10568	19737

Table 3.37. Varying the selectivities.

in size a great deal before transmitting them to the result site. This situation occurs if the domain sizes are small, the selectivities are small or the same attribute is used in several clauses. Furthermore, the cost to obtain a schedule by an algorithm of the decomposition family is far more expensive than by one of the semi-join family.

4. DATA AND OPERATION ALLOCATION IN A DISTRIBUTED DATABASE

The data of an operational distributed database is assigned to various sites in the computer network. Various aspects influence the reasons why the data are allocated in a particular way; some are quantifiable others are not [Charrel1981]. One non-quantifiable aspect is site autonomy. Site autonomy gives the owner of the data a more secure feeling, because his data are spinning around on its own disk at a site under his control and no operator at another site can tamper them. This feeling of security is, of course, relative. Some of the quantifiable aspects are: availability [Martella1981], response time and utilization of resources. In this monograph we will confine ourselves to the quantifiable aspects.

The organization of this chapter is as follows. In section 4.1 we define the data and operation allocation problem and compare it with the well-known file allocation problem. Literature on the latter is briefly reviewed in section 4.2. In section 4.3 the notion of a processing-schedules graph is introduced. Two different approaches to data allocation and the way the objects to be allocated are determined, are discussed in sections 4.4 and 4.5. A comparison is given in section 4.6. Finally, in sections 4.7 and 4.8 algorithms are discussed to determine allocations that minimize either total transmission cost or average response time.

4.1. Data and Operation Allocation and File Allocation Problem

Before distributed database management systems were investigated, networks existed already for many years. On top of these networks a distributed file system can be constructed. The problem where to allocate a file and its copies, given a known set of retrievals and updates, such that a cost function is minimized, is known as the **file allocation problem.** In most research the cost function used is the total transmission cost. This problem is known to be NP-complete [Eswaran1974]. Many variants of this problem exist. It can be made more realistic by adding bandwidth constraints on the communication channels, by adding response time constraints on the retrievals, etc.. Some of them will be discussed in section 4.2.

A distributed file system is very much unlike a distributed database, and 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 unit 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 chapter 3 we observed that to process a query, also data transmissions between sites where fragments are allocated are needed. In [Elam1978, Chu1980] attempts were made to characterize these. This means that the fragments can not 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 and the frequencies of their use, determine first the fragments to be allocated and secondly allocate these fragments and their copies and the operations on them to the sites of the computer

network such that a certain cost function is minimized. We will often use the term data allocation when we mean data and operation allocation.

The data allocation problem can be viewed as a sort of dual problem of the distributed query processing problem. There, the allocation is given and the schedules have to be determined. Here, the queries and updates are given and the allocation has to be fixed.

4.2. Overview of File 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 [Hevner1981].

Chu was probably the first to work on the file allocation problem. In [Chu1969, Chu1973] he presented a simple model that only allows for a non-redundant 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. Both a query and an update consist of a request to a site where the required file is located and the answer back to the requesting site. The latter transmission receives priority over the other transmissions. 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 [Casey1972] 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 only one. The model also allows for different cost rates for query and update transmissions. The reason for this is that, for example, updates can be done on a periodic basis, making them less costly. The optimization goal is to minimize the cost in dollars of the transmissions plus the storage cost of the files. In [Eswaran1974] it was shown that the file allocation problem modeled this way is NP-complete by reducing the Set Covering Problem to it [Garey1979].

In [Levin1974, Levin1975] a problem was emphasized which is deeply entangled with the file allocation problem, namely the allocation of the application programs that access the files. Data can relatively easily be stored at different sites or transmitted from one site to another. Programs, however, because of the programming language in which they are written, are not as portable as one might wish. Although, programs are placed between the files and the result site in the processing schedule the access of the files is still simple. For example, a user can use application programs only independently of each other, meaning that the result of an application program is sent to the result site and can not be used as input of another application program. A second important aspect discussed by Levin and Morgan is the change in the access pattern over time. For example, in a computer network that has sites in different time zones the access of a particular file may be time dependent and moving it to another time zone during the day may minimize transmission cost. Also, more general changes in the access pattern can be characterized but only over a limited period of time. A third aspect discussed is the available information about the access pattern. Quite often, the frequencies and the amounts of data transmitted are not known. To overcome this problem all quantities in the formula to be optimized are replaced by random variables with known probability distribution functions, rendering the formula into a random variable itself.

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 [Mahmoud1976] the capacity of the communication channels may be determined besides the allocation of the files. The resulting model is a non-linear integer programming problem for which a heuristic approach is used to solve it.

In [Ramamoorthy1979] 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. To enforce this simple way of query processing, on the one hand, and to reduce the sizes of the relations to be transmitted to the result site by semi-joins, on the other hand, information indicating whether tuples are part of a particular join is included in the relations themselves. The latter introduces a problem, namely for which joins should such information be added; however, this can be determined at database design time.

4.3. Data and Operation Allocations and Their Costs

Which objects are to be allocated in a distributed database depends on the cost function to be minimized, the kind of queries and updates that are used, etc.. In subsection 4.3.1 this problem will be discussed in detail. Here it suffices to know that the objects are fragments of relations.

To allocate these 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, which 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. To be able to discuss allocations and their costs, and to be able to manipulate allocations, we introduce some notions.

A **nucleus-site** is a 2-tuple, (F,O), where F is a set of fragments and O is a set of operations. An **operation** is a 3-tuple (i,λ,x), where x is the execution time of the operation and λ the frequency with which the ith transaction of which the operation is part, is executed. The actual operation itself is not represented since we are not interested in it. A nucleus-site may have assigned to it a set of other nucleus-sites; this set will be called the **assigned set.** There are two types of nucleus-sites, namely **physical sites** and **virtual sites.** A physical site is used to represent a site in the computer network, and a virtual site represents a fictitious site, of which the purpose will become clear in a moment. A virtual site can be **assigned** to at most one other nucleus-site, and the consequence is that it is then placed in the assigned set of that nucleus-site. A physical site can never be assigned to another nucleus-site.

The **union** of two nucleus-sites is a nucleus-site whose fragment-set is the union of the fragment-sets of the two nucleus-sites, whose operation-set is the union of the operation-sets, 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. Between two physical sites the union is not defined.

The relation **being assigned to** is transitive, i.e., if A is assigned to B, and B is assigned to C then A is assigned to C as well. However, A is assigned to C and B is assigned to C does not mean that the union of A and B is assigned to C. Also, A is assigned to B does not necessarily mean that A belongs to the assigned set of B.

Both the union and the assignment are used to construct certain allocations such that the cost can be computed; only the union is permanent, while an assignment can be undone.

Because most of the distributed query processing algorithms determine at which sites certain operations have to be executed, we assume that the operation allocation is fixed if the data allocation is given. An **initial allocation** is an allocation where a fragment-set of a virtual site contains at most one fragment and the fragment-sets of the physical sites are empty and where none of the virtual sites is assigned to a physical site. A **partially specified allocation** is an allocation where some of the virtual sites have been assigned to (or united with) nucleus-sites. A **completely specified allocation** is one in which all virtual sites have been united with physical sites.

The assumption that the operation allocation is fixed, given the data allocation, is acceptable if we are only interested in minimizing total transmission cost. On the other hand, when minimizing response time, the operation allocation is just as important as the data allocation. In that case for the initial allocation the operations are not included in the operation-sets of the virtual sites that contain fragments, but each of them is put in an operation-set of a newly created virtual site. We will come back to this point in section 4.8.

The computation of the cost of an allocation is done by means of a **processing-schedules graph.** Such a graph 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 3-tuple (i,λ,d), where d stands for the amount of data transmitted from one adjacent node to the other for processing the ith transaction and λ 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 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 to compute the cost of an allocation from the processing-schedules graph.

For every allocation a processing-schedules graph can be **constructed.** The schedule for a query or update, given an allocation, is computed as follows: imagine that all physical and virtual sites which contain the required fragments (whether one is assigned to the other or not) are different sites in a computer network, and that these sites have allocated to them the fragments in their corresponding fragment-sets; also, the physical site that corresponds to the result site is treated as a separate site, if this was not already done because it contained fragments. This hypothetical allocation together with the transaction is given to the query processing algorithm to compute a schedule. Such a schedule consists of data transmissions and operations. For each data transmission an edge is created between the corresponding sending and the corresponding receiving physical or virtual site. And, each operation is added to the operation-set of the appropriate physical or virtual site. Note that the schedules do not take into account the fact that certain virtual sites are assigned to nucleus-sites.

A nucleus-site in the processing-schedules graph together with its assigned set 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. Fig. 4.1 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. If VS_j is united with PhS_i the box of VS_j together with the edge between VS_j and PhS_i disappears. The remaining adjacent edge of VS_j is inherited by PhS_i. If VS_j and VS_k are united both their boxes disappear together with the edge between them, and a new box is created which inherits the adjacent edges of both VS_j and VS_k. The status of this new box depends on how the original virtual sites are related. For example, if VS_k was assigned to VS_j which in its turn was assigned to PhS_i and VS_j and VS_k are united, then the resulting virtual site of the union is assigned to PhS_i. The same result is obtained if VS_j and VS_k were, independent of each other, assigned to PhS_i. If VS_j and VS_k were assigned to different nucleus-sites then it is not defined to which assigned set their union belongs.

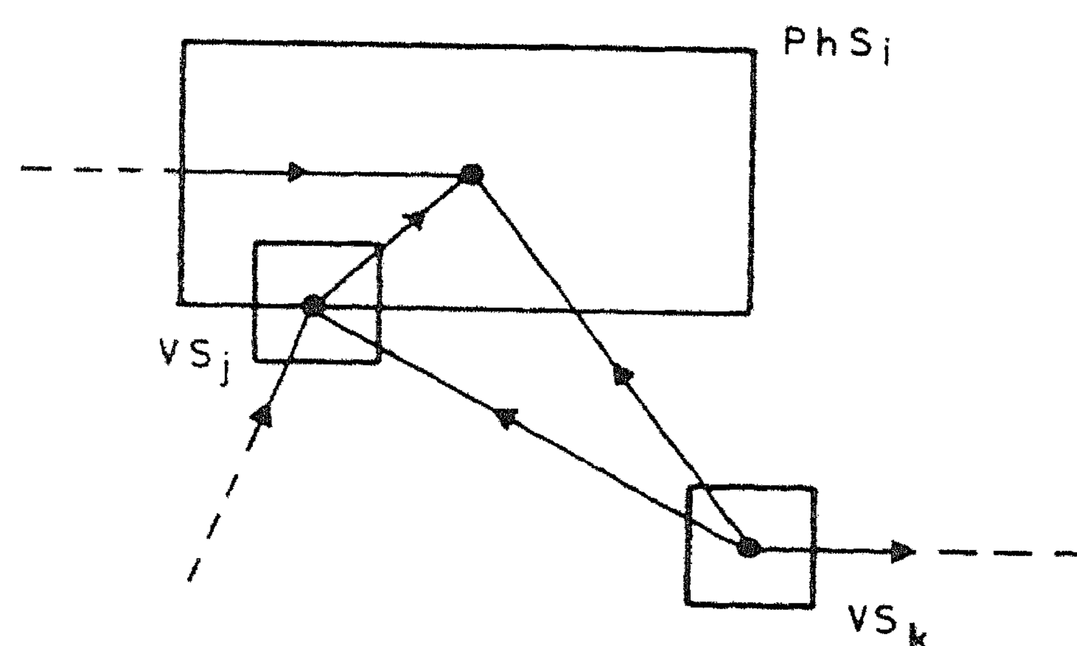


Figure 4.1. Graphical representation of a physical site with virtual sites.

Example 4.1

Fig. 4.1 shows a processing-schedules graph of a partially specified allocation for three transactions. It is constructed under the assumption that all physical and virtual sites are different sites in a computer network. There are two physical sites, PhS_1 and PhS_2, and three virtual sites, VS_1, VS_2 and VS_3, of which the latter is assigned to PhS_2. Transaction 1 is a query, which is executed 10 times per unit of time, and it computes a join between F_1, allocated to VS_1, and F_2, allocated to VS_2. Its processing schedule consists of the restrictions S_1 and S_2, which are elements of the operations-sets of VS_1 and VS_2, respectively. The result of S_1, whose size is 200 bytes, is transmitted to VS_2, where the join (J) with the result of S_2 is computed. Finally, this result, 800 bytes in size, is sent to the physical site PhS_2.

Transaction 2 is an update, which is executed 6 times per unit of time. It updates (U) fragment F_1, which is allocated to VS_1.

Transaction 3 is again a query which retrieves (S_3) data from F_3, allocated to VS_3. It is executed 20 times per unit of time.

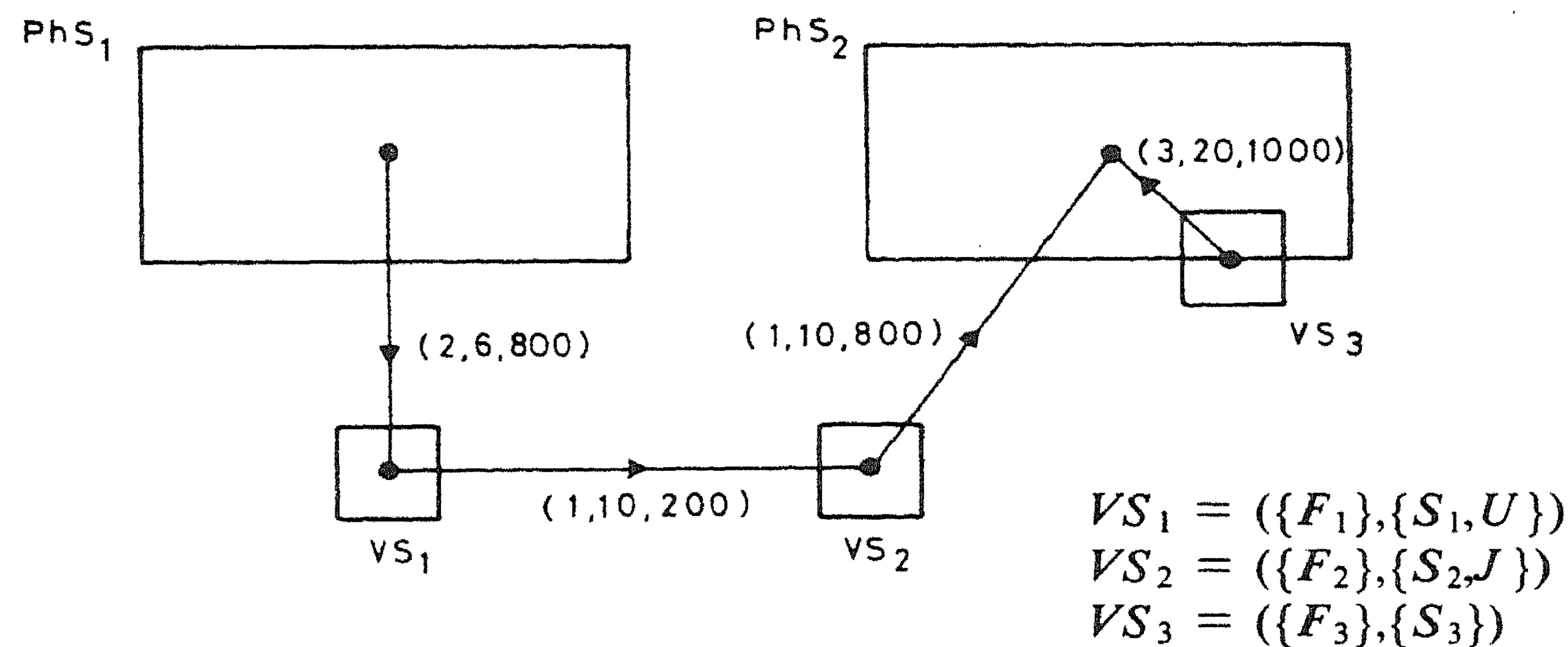


Figure 4.2. An example of a processing-schedule graph.

□

Given the processing-schedules graph we can compute the **cost of an allocation.** This allocation may be completely or partially specified. For each transaction we can construct a processing schedule from the processing-schedules graph, taking into account the assignment of certain virtual sites to nucleus-sites. This is done as follows: imagine that each physical site and each virtual site that is not assigned to another nucleus-site is a different site in a computer network. And furthermore, each virtual site that is assigned to another nucleus-site is identified with the site in the computer network that corresponds to the nucleus-site to which it is assigned. Hence, transmissions from a virtual site to the nucleus-site to which it is assigned are not counted in the cost of the schedule under construction; all other transmissions are. For each nucleus-site that is not assigned to another one the expected waiting time of its operations is computed as if the operation-sets of the virtual sites, which are assigned to it, were united. The expected waiting time of the operations in an operation-set of a virtual site that is assigned to a nucleus-site is computed as if the operation-sets of the virtual site and the nucleus-site to which it is assigned were united. Note, that although more than one virtual site may be assigned to a nucleus-site the expected waiting times of their operations are computed for each virtual site independently.

The expected waiting time of transmissions can be computed by giving to a routing algorithm the amount of data transmitted from one site to another and how frequently this is done.

The above should give enough information to compute whether constraints, such as for example the bandwidth of a communication channel, are met, and to construct the processing schedules and to compute their cost, such as total transmission cost, response transmission time, response time, etc..

Example 4.2

Given the processing-schedules graph of the partially specified allocation discussed in example 4.1. From it we will construct the processing schedules of the three transactions, taking into account the assignment of VS_3 to PhS_2. For simplicity we confine ourselves to the transmissions in the processing schedules.

Transaction 1:

$$F_1 \xrightarrow{200} F_2 \xrightarrow{800} \text{result site}$$

total transmission cost = 1000

Transaction 2:

$$\text{result site} \xrightarrow{800} F_1$$

total transmission cost = 800

Transaction 3:

no transmissions

This results in a total transmission cost of the allocation of $10 \times 1000 + 6 \times 800 = 14{,}800$.

□

In this section completely and partially specified allocations were introduced. To compute the cost of an allocation a processing-schedules graph must be determined. This can be done by giving the allocation to a query processing algorithm which returns a processing schedule for each query. The transmissions and operations in such a schedule are incorporated in the processing-schedules graph. In the rest of this chapter we will discuss various aspects of the data allocation problem within the model described above.

4.3.1. Forking Points and Forking Graphs

So far we 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 allocation such that a given cost function is minimized. Changing an allocation may have effects on the processing schedules of the transactions. Especially, the placement of forking points in a schedule depends on the allocation. Therefore, we will introduce a **forking graph**, which enables us to more efficiently handle forking points when changing partially specified allocations. The cases in which forking points 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 and the second case with 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. The representation of a forking graph is shown in fig. 4.3. 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** (The term notification is

used because it concerns a selective broadcast). All the nodes are VS-nodes. All edges in a forking graph are labeled with the same 3-tuple, because to each receiving site the same amount of data will be transmitted with the same frequency. Each receiving node is part of a schedule for a query that references the fragment allocated to that receiving node.

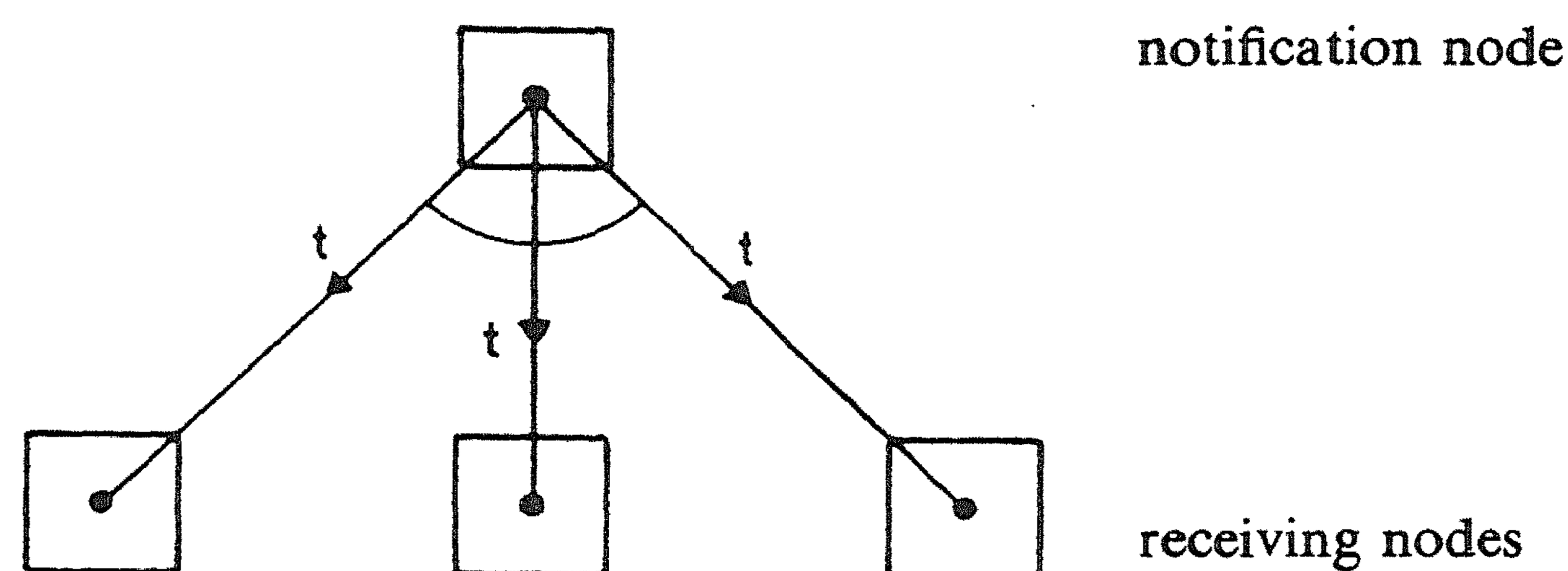


Figure 4.3. Forking-graph, with $t = (i, \lambda, d)$.

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; this is the forking point. 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, also operations directly involved with this transmission, and operations that worked on superfluous copies, are removed from the operation-sets.

The same forking-graph can be used for the notification of the processing schedule to the sites involved.

4.3.2. Unit of Allocation and Processing Schedules

Having explained how a processing-schedules graph for a given allocation can be constructed and how to compute the cost of that allocation we will now discuss how to determine the unit of allocation.

An obvious unit of allocation is a complete relation in the global conceptual schema. Data occur in the same relation because there is a logical relationship

between them. Such a relation may, however, contain tuples that are accessed by users of different sites. For example, a real-estate agent is mainly interested in properties that are for sale in his area. Allocating the relation containing tuples about these properties to one site would make the real-estate database a central one and most agents would have to retrieve data from a distant site. Splitting the relation such that tuples concerning specific areas form fragments of the relation and allocating these fragments to sites located in these areas would decrease the total transmission cost involved in query processing drastically. Mainly, because most of the data is locally available. In this example the splitting is rather simple. Determining the fragments to be allocated will be more formalized now.

Let us assume that we have a set of queries, updates and their 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 restrictions and projections.

First, we take a look at just one relation, say R. Assume we have a collection of queries $\{Q_k\}_{k \in I}$ and each Q_k uses only a fragment F_k of relation R, $F_k = R\{C_k\}[P_k]$, where C_k is a restriction and P_k a set of attributes of R. Every C_i defines a subset of the tuples of R. The corresponding subsets of some C_i may overlap. We are interested in these intersections, which can be uniquely identified by saying in which subsets of the C_i's they fall. Consider an n-bit number, one bit for every C_i. If the intersection, I, is part of C_i the ith bit is 1, otherwise it is 0. A unique identification of I is the decimal number that corresponds to the n-bit string. The number zero is not used because it corresponds to the empty set. In this way R is **split horizontally.**

Instead of the P_k's the singleton subsets A_j are used. A_j contains the jth attribute of R. in this way R is **split vertically.** The resulting fragments F_{ij}, where i is the decimal number corresponding to the bit string of the horizontal splitting, and j the index of the attribute, respectively, are considered as objects to be allocated. Fragments that do not contain a primary key are extended such that it is included.

Example 4.3

Assume we have a relation

$PARTS(PNO, PNAME, CITY)$

and the following queries:

$Q_1 = PARTS[PNO, PNAME]$
$Q_2 = PARTS\{CITY = \text{Paris OR } CITY = \text{Amsterdam}\}$
$Q_3 = PARTS\{CITY = \text{Amsterdam OR } CITY = \text{London}\}$.

From these queries we can determine the C_i's:

$C_1 = \textbf{true}$ $\quad A_1 = \{PNO\}$
$C_2 = (CITY = \text{Paris}) \textbf{ or } (CITY = \text{Amsterdam})$ $\quad A_2 = \{PNAME\}$
$C_3 = (CITY = \text{Amsterdam}) \textbf{ or } (CITY = \text{London})$ $\quad A_3 = \{CITY\}$

In general, there would be $(2^3 - 1) \times 3$ fragments F_{ij}. In this special case we see that the bit of C_1 is always 1, otherwise the corresponding set is empty. Hence, there are $(2^2 - 1) \times 3 = 9$ fragments.

$F_{51} = PARTS\{CITY =$ London$\}[PNO]$
$F_{52} = PARTS\{CITY =$ London$\}[PNAME]$
$F_{53} = PARTS\{CITY =$ London$\}[CITY]$

$F_{61} = PARTS\{CITY =$ Paris$\}[PNO]$
$F_{62} = PARTS\{CITY =$ Paris$\}[PNAME]$
$F_{63} = PARTS\{CITY =$ Paris$\}[CITY]$

$F_{71} = PARTS\{CITY =$ Amsterdam$\}[PNO]$
$F_{72} = PARTS\{CITY =$ Amsterdam$\}[PNAME]$
$F_{73} = PARTS\{CITY =$ Amsterdam$\}[CITY]$

Fragments with only *PNAME* and *CITY* are extended such that the primary key, *PNO*, is included as well.

□

Before discussing whether further splitting is necessary we show the extension for queries that reference more than one relation. The extension to queries that contain joins is simple. Again, the fragments required by the join can be described by projections and restrictions. The projection set contains the target attributes, the joining attributes and the attributes on which the restriction operates. The restriction includes the clauses that are applicable to the corresponding relation. It may also include clauses that are obtained by transitivity on clauses on the other relation and the joining clauses.

To investigate whether further splitting is necessary, we discuss the processing schedules of the operations restriction, projection and join. Each of these operations must be part of the set of queries $\{Q_k\}_{k \in I}$ on which the relations are split.

A **restriction** will be part of some query, so it is in some C_p. The result of the restriction is the fragment that can be composed of the fragments F_{ij} where i can take any value as long as the pth bit is set, and j can take any value. What is done with these fragments depends on the next operation. If there is none, all these fragments are sent to the result site.

The same for a **projection,** only now the fragments are determined by the attribute set. The result of a projection is computed in several phases. First, all fragments with the same first index are collected at one site, then local projections are computed (the same as a distributed projection in [Pelagatti1979]. Then the results are collected at one site where again a projection is computed.

Which fragments are involved in a **join** can be determined in the same way as was done for a restriction and a projection. The result of a join is computed as follows. For one relation all fragments with the same first index are collected at one site. All the fragments of the other relation are sent to these sites where the joins are computed. Note, the fact that a forking graph is used here; see also fig. 4.4.

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, and such that an edge coming from F is now replaced by an edge coming from 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.

A split of F into F' and F'' is called a **balanced split** if an outgoing edge of F labeled with (λ,d) is replaced by two edges labeled $(\lambda,d\,|F'|\,/\,|F|)$ and $(\lambda,d\,|F''|\,/\,|F|)$ leaving F' and F'', respectively. An incoming edge of F labeled (λ,d) is replaced by two edges both labeled (λ,d) going to F' and F''.

Theorem 4.1 A horizontal split of F into F' and F'', that is done randomly, is balanced, 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 size of the result produced by F' and F'' is $|F'|\,/\,|F|$ and $|F''|\,/\,|F|$ times the 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 balanced.

□

A vertical split is not necessarily balanced 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 projection. Assume that F is one of the fragments after a vertical split, and that in a schedule for a projection all transmissions are directed to F. If F is split vertically into F' and F'' the schedule will change drastically, because F no longer exists. One obvious change is to direct all the edges to F' including one from F''. But then the split is not balanced.

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 4.2 Further splitting the fragments obtained by horizontally splitting the relations based on clauses of queries and vertically splitting them on attributes 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 (λ,d) are replaced by λd. The fragments obtained can not be split vertically because they contain only one attribute. Therefore, let us consider a horizontal split of fragment F into F' and F''. And 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 site will not increase the total transmission cost. Fig. 4.4 shows part of the processing-schedules graph in the split form; remember, the horizontal split of F is balanced if the processing schedules are static under splitting. All virtual sites have been assigned to physical sites except VS' and VS'' which contain F' and F'', 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; r' stands for $|F'|\,/\,|F|$ and r'' for $|F''|\,/\,|F|$. In this partial processing-schedules graph there are six possible assignments. In all of them either VS' or VS'' is assigned to PhS_1 or PhS_2. Without loss of generality assume that $PhS_1 = \{VS'\}$.

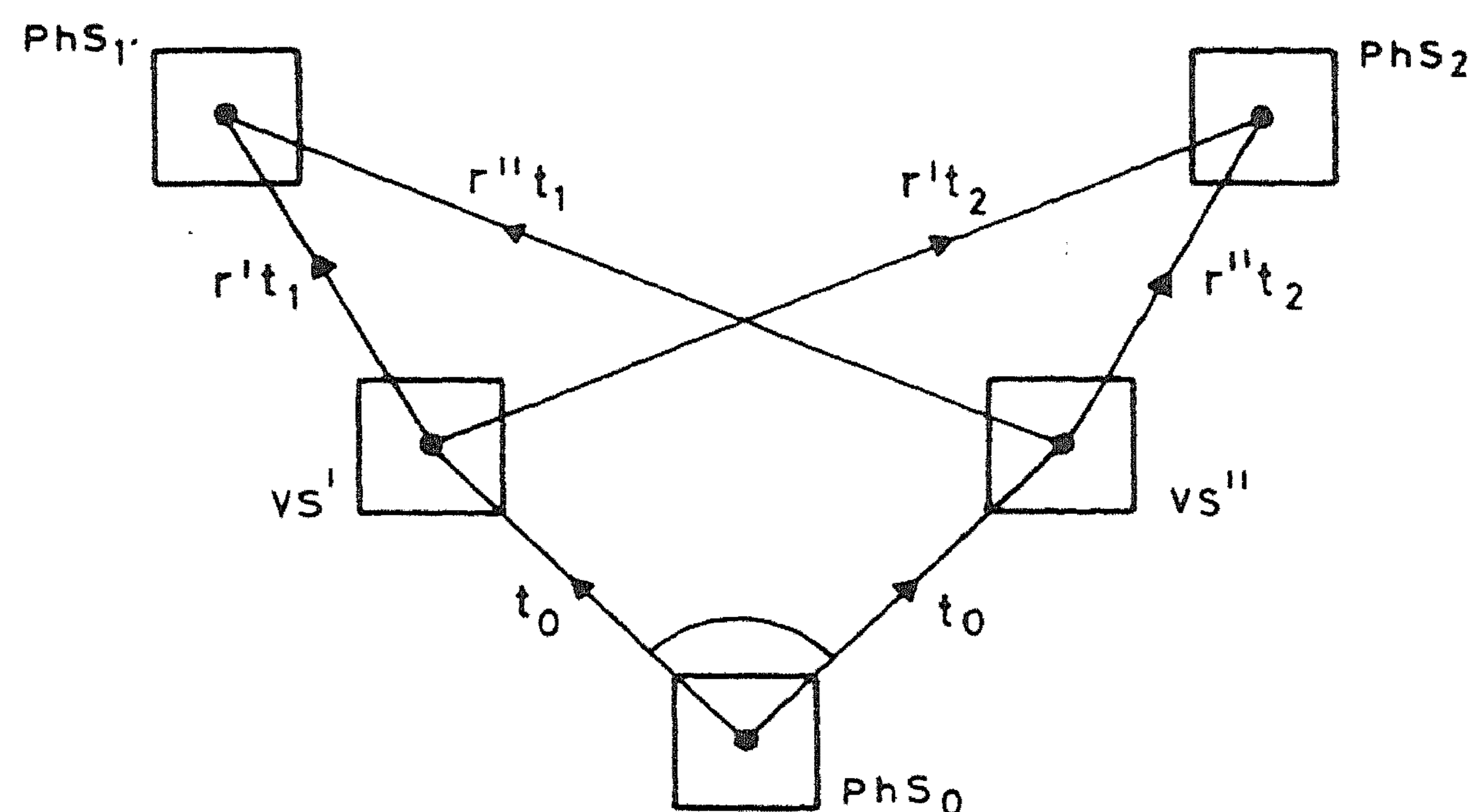


Figure 4.4. Partial processing-schedules graph.

This means that $r't_1 \geqslant r't_2$, which implies $r''t_1 \geqslant r''t_2$. 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.

□

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

When minimizing response time the problem of determining the objects to be allocated is a lot 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. Ideally every tuple is placed at a different site giving a maximum of parallelism. Due to overhead it is better to group tuples together. We assume that the minimum number of tuples per fragment is given as a system parameter. So, the fragments obtained for minimizing total transmission cost are horizontally split further but such that no newly created fragments contain too few tuples.

The fragments constructed in this subsection are the objects to be allocated. They are assigned to the virtual sites in the initial allocation. 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 information about attributes that are contained in fragments, restrictions 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 query processing unit. And, 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 is done with the relations.

To summarize this subsection we may conclude that just looking at the logical components of a database is not sufficient to determine the objects to be allocated. Therefore, a way to split relations horizontally and vertically based on the queries and attributes was proposed. The resulting fragments form one of the inputs of the algorithm that constructs the processing-schedules graph.

4.4. Centralized Data Allocation

In this section and the following one we will discuss two different approaches to data allocation, a centralized and a decentralized approach. 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 the database management system itself is allowed to change the existing allocation.

4.4.1. Construction of Processing-Schedules Graph

In subsection 4.3.2 a way of determining the fragments to be allocated, was discussed. All queries or updates will be used to compute these fragments. For each query (updates will be discussed later) a copy is made of the required fragments, which are assigned to newly created virtual sites. At this point the processing-schedules graph contains nodes for each site in the computer network (physical sites) and the above created nodes for virtual sites.

If the distributed query processing algorithm can handle an arbitrary allocation of fragments, the processing schedules for this initial redundant allocation can be computed as described in section 4.3. The operations that, according to the schedule, have to be computed at a virtual or physical site are added to the operation-set of the corresponding node. For the data transmissions, edges between the appropriate nodes are created, and labeled with the amount of data transmitted and the frequency with which the query is stated.

Not all distributed query processing algorithms are able to compute a processing schedule given an arbitrary allocation of fragments. For example, some of them assume that a whole relation is the unit of allocation. In that case the fragments of

each relation can be collected at a site, such that the query processing algorithm can be applied, or the schedules for the operations discussed in subsection 4.3.2 are used.

If there are no update transactions we are finished and from the processing-schedules graph a completely specified allocation can be determined minimizing some cost function. Here, we explicitly assume that each site has enough storage to contain the whole database.

In general, however, there will be updates. As an example we will show the processing-schedules graph for an update if centralized locking is used (Note, the fact that centralized locking is discussed, has got nothing to do with centralized data allocation). In **centralized locking** all requests for locks are funnelled through one site. That site decides whether a query or update can get the requested locks. Which of the physical sites should do this job is not known in advance. Therefore, all the involved operations are allocated to a virtual site, say VS_0. Finally, if a completely specified allocation is determined the site that controls the locks is known.

Assume an update transaction is initiated at PhS_1 that changes tuples in fragment F. All operations that have to be done by the site corresponding to PhS_1 are put in its operation-set. A schedule for an update consists of the following:

- request for an exclusive lock on F is sent to VS_0,
- an acknowledge is sent back to PhS_1 after the lock is set,
- the schedule or the query determining the tuples to be updated is transmitted to VS_1, where the tuples to be changed are determined,
- the tuples are sent back to PhS_1,
- PhS_1 computes what the changes are and notifies all the virtual sites (such as VS_1 and VS_2) where the copies of F are about the update.
- the final action of PhS_1 is to send a message to VS_0 to release the locks.

If in the second step the lock is rejected the request will be sent again. This can easily be included in the processing-schedules graph if statistics are available on how often this happens. Other ways of processing updates are possible, however, the above listed actions will somehow be part of it.

Although, initially every query has its own copies of the fragments, it may happen that some virtual sites containing a copy of the same fragment are united because it is too expensive to maintain too many copies. In this way the algorithm that determines a completely specified allocation not only decides where to allocate the fragments but also how many copies have to be maintained.

One may object against the fact that all queries and updates have to be known in advance to compute the fragments to be allocated and to compute the initial allocation. In some cases they are not known at all. Then 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 due to changes in the allocation should be estimated, based on the queries and updates that are known. A way of doing this is to look at operations most often used on the fragments and compute schedules such as was done in subsection 4.3.2.

4.4.2. Static versus Dynamic Schedules

In the processing-schedules graph corresponding to the initial allocation, the processing schedules were computed based on the assumption that between virtual sites transmissions are required if communication is necessary. During the search for an optimal allocation, the cost of other partially or completely specified allocations will be computed. To make this computation more efficient we may keep the schedules the same as in the initial processing-schedules graph. Only, if two virtual sites are united the edges between them disappear, or, if they are part of an update-graph, an edge in this graph disappears. An allocation algorithm that computes the cost of allocations in the above described way uses **static processing schedules.**

Although the computation of the cost of allocations is more efficient, finding a minimum total transmission cost allocation is still NP-complete [Cook1971, Aho1974, Garey1979].

Theorem 4.3 The problem whether there exists a completely specified allocation with total transmission cost less than or equal to a certain T using static processing schedules is NP-complete.

Proof 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 3-dimensional matching, a known NP-complete problem, to it.

> **3-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 transmission-strategy graph from a 3-dimensional matching problem is done as follows:

- the elements of the sets W, X and Y are the VS-nodes.
- for every triple $(w,x,y) \in M$ create a PhS-node and connect the VS-nodes corresponding to w, x and y to this PhS-node; label these edges with the number $d = 2|M| + 1$.
- create an edge between the VS-nodes 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 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 transformation. Assume this is the case. In this allocation each VS-node is united with a PhS-node with which it is connected by an edge in the processing-schedules graph. We will show that if it were not connected, the total transmission cost would be larger. For, in the processing-schedules graph there are $3|M|$ edges with label d, and there are $3q$ VS-nodes. Assume there

is a VS-node that is not united with a PhS-node with which it is connected by an edge. Then the total transmission cost is at least

$$\begin{aligned}(3|M| - 3q + 1)d &= 3(|M| - q)d + d \\ &> 3(|M| - q)d + (l - 2q),\end{aligned}$$

because $d > 2|M| \geq l$, so such a VS node does not exist.

Hence, the number of VS-nodes united with the same PhS-node is less than or equal to three. Also, a PhS-node will not be united with less than three VS-nodes. Assume that we have an allocation such that there are

q_1 PhS-nodes with 3 VS-nodes,
q_2 PhS-nodes with 2 VS-nodes,
q_3 PhS-nodes with 1 VS-node,

with $q_1 < q$.

Note: $3q_1 + 2q_2 + q_3 = 3q$, so $q_2 \leq 3/2\,(q - q_1)$.

The maximum number of edges, that are labeled 1, that will disappear because of this allocation is $2q_1 + q_2$. Because all VS-nodes are united with PhS-nodes with which they are connected by an edge the total transmission cost will be at least:

$$3(|M| - q)d + (l - 2q_1 - q_2) > 3(|M| - q)d + (l - 2q),$$

because

$$2q_1 + q_2 \leq 2q_1 + 3/2\,q - 3/2\,q_1 = 3/2\,q + 1/2\,q_1 < 2q,$$

which contradicts the first assumption.

Hence, either there are no VS-nodes united with a PhS-node or exactly three. It also shows that the three VS-nodes united with the same PhS-node are interconnected by two edges. Thus, the three VS-nodes, united with the same PhS-node, correspond to a triple in a matching M'. (All VS-nodes are used exactly once).

Now assume that there is no allocation with a total transmission cost less than $3(|M| - q)d + (l - 2q)$, and assume we have a matching $M' \subset M$. Construct the corresponding processing-schedules graph as was done above and unite the VS-nodes 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 VS-nodes corresponding to a triple of M' are united with the same site. Thus the total transmission cost is

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

□

Corollary 4.4 The problem of finding an allocation with minimum total transmission cost is NP-complete.

□

Clearly, using static processing schedules may lead to data allocations of which the real cost differs from the computed cost. We know that if the allocation changes the processing schedules change as well. A consequence is that the cost of a completely specified allocation obtained using static processing schedules might not reflect the real cost. The reason is that a query processing algorithm might come up with completely different schedules given the final allocation obtained using static schedules. Therefore, to compute the real cost of an allocation the schedules have to be recomputed. An allocation algorithm that does this uses **dynamic processing schedules.**

Whether the use of static or dynamic processing schedules is preferable depends on many aspects. For example, the dynamic case may be infeasible because it is time-consuming, and its effect on the cost of the allocation might be marginal. We come back to this point in sections 4.7.

4.5. Decentralized Data Allocation

Quite a different approach is taken in this section. Instead of assuming the existence of a database administrator that may make any change in the existing data allocation, we assume that the data is owned by the users, or that the distributed database is a collection of databases owned by different parties. Both cases have in common that there does not exists 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 is done by allowing the creation of copies of fragments by the DBMSs. This approach is called **decentralized data allocation.**

4.5.1. Private Copies and Processing-Schedules Graph

A group of users at a site that share the same view of a database can request their local DBMS to somehow change the allocation such that a certain cost function is minimized. Because the original relations are, in general, not owned by this group their DBMS may not change their location. Therefore, copies are made of the fragments of the relations in which the group is interested. These copies are allocated to virtual sites. The edges in the processing-schedules graph are obtained in the same way as in the centralized approach, however, here only the queries of the group are taken. Updates of users of this group and others are included as far as the fragments of interest are concerned. So, the fragments are only determined by queries of this group and not by all users as in the centralized data allocation. Therefore, the processing-schedules graph is less complex, and determining an allocation such that a cost function is minimized is a lot simpler. Each group of users will have its own processing-schedules graph and an allocation is determined for each one separately. This means that if copies are made they are solely used by one such group, and will therefore be called **private copies.**

The data allocation in the decentralized approach can change more or less continuously through time. A group of users starts using the database or changes its access pattern. To express this, a processing-schedules graph is constructed with

private copies allocated to virtual sites. Because there are many other users around, the fragment-sets, and operation-sets of the physical sites will, in general, not be empty. Depending on the cost function to be minimized this is taken into account. For example, a physical site may have a "large" operation-set, causing a high expected service time for an operation. When minimizing the response time of the queries stated by the group this physical site will probably be avoided.

Example 4.4

Assume that the relations R_1 and R_2 are allocated to PhS_1 and PhS_2, respectively, and that their locations are fixed. A group of users at the site that corresponds to PhS_3 wants to use fragments F_1 and F_2 of R_1 and R_2, respectively. The fragments are allocated to VS_1 and VS_2, respectively. Besides the transmissions and operations to process the queries of the users of site PhS_3, there will also be transmissions from PhS_1 to VS_1 and PhS_2 to VS_2 to keep the fragments F_1 and F_2 up-to-date.

□

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 deciding the update cost [Adiba1980b, Adiba1981]. Decreasing this cost will make it more likely that an allocation is chosen such that query processing 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. If this is not done it will affect *all* users. We come back to this point in section 4.6.

To investigate the feasibility of these ideas Sang Ajang and Spoor [Sang Ajang1981] made an initial design of a distributed DBMS, called STUFF, that allows for periodically updated fragments. The private copies are used by a group of users for query processing. Under the assumption that most of the transactions are queries the allocation of the private copies can be chosen such that the cost for query processing is minimized. Also, the queries will hardly be hindered by other concurrent transactions, because only periodic updates need to lock the private copies. The group of users must make some kind of arrangement with their local DBMS to keep the private copies up-to-date. For example, every hour or day their site, called Initiating Site, will send a message to the original relations to request for all the updates since the last periodic update. These updates are sent directly to the sites where the private copies reside. Note, because the allocation is determined to minimize (query) processing cost the sites of the private copies are not necessarily the Initiating Site. After arrival of the updates the Initiating Site is notified such that the updates are made effective on all copies in between queries to ensure a consistent database. Updates can not be done on out-of-date copies. Hence, the private copies are merely for retrieval and all updates should be done on the original relations. Interactive update sessions contradict the philosophy behind the idea of periodically updated fragments and were, therefore, not accounted for in STUFF. Updates can be done in batch mode, and the changes are available after the next periodic update of the private copies.

If every group of users is forced to have its own private copies that are periodically updated, the original relations are only accessed for updates. Periodic updates can be obtained from a log-file of the relations. The separation of the queries and updates will decrease the heavy load on the concurrency control mechanism. We expect that for many applications a system with private copies that are updated on a periodic basis, will suffice. This approach also emphasizes the fact that in practice a distributed database will not be just a database that is centralized at the logical level and distributed at the physical level, but that it is distributed at the logical as well at the physical level (see section 2.1). For example, a distributed database that is a collection of databases. If retrieving data from several databases is done frequently it might be worthwhile to create private copies of those parts of the databases in which the users are interested and allocate them to the users' sites. Query processing is now cheap in the sense that no data transmissions are required and if slightly out-of-date copies suffice the cost to keep them up-to-date is minimum.

As far as data allocation is concerned, however, creating private copies is attractive. For privacy and security reasons it might happen that no private copies may be created or that they may only be allocated to certain sites.

4.6. Comparison Centralized and Decentralized Approach

Several issues will be discussed to compare the centralized and the decentralized approach to data allocation [Apers1981b]. The first issue is the **applicability**. To determine the fragments to be allocated in the centralized case, all queries and updates have to be known to some central organization such as the database administrator or the DDBMS. Also, the frequencies with which they are executed, have to be known to construct the processing-schedules graph of the database. To install a new allocation the central organization should have the power to take decisions about data allocations at different sites. An example of such a database may be one that exists on a local network.

If the processing-schedules graph consists of a collection of processing-schedules subgraphs that are loosely connected, that is by means of edges whose labels hardly contribute to the cost function, then we are on the border between the centralized and the decentralized approach. The edges that connect the subgraphs can be deleted and a data allocation can be determined for each of them independently.

In the decentralized approach only the transactions of a group of users at the same site that share the same view have to be considered. Their local database administrator or their local DBMS must have the opportunity to create copies of relations and allocate them to arbitrary sites. The processing-schedules graphs in the decentralized approach are a lot smaller and, therefore, it will be easier to compute an optimal allocation.

The second issue is the **flexibility** with which it can incorporate changes in the users' access pattern. In the centralized approach the efficiency of the data allocation should be checked on a regular base. This means that based on currently available information a new data allocation should be determined and its cost be compared with the cost of the current allocation. If some threshold is passed a re-organization of the data allocation must be ordered. This re-organization itself will cost a lot and should be viewed as a sort of investment to get a more efficient allocation. This investment should of course not be too costly compared to the gain in efficiency. This problem can be overcome by requiring that the investment is earned back within a

certain time interval. What a reasonable time interval is can be determined in practice. To incorporate this in the processing-schedules graph we introduce the **re-organization graph** which is a subgraph of the processing-schedules graph. For every fragment in the fragment-set of a physical site a new virtual is created. Between the virtual site, to which it is allocated in the current allocation, and such a new virtual site an edge is placed. Its label is the cost if the virtual site were to be united with another physical site divided by the period in which the investment should be earned back. The other edges and their labels, and the operations involved are obtained the same way as discussed in subsection 4.5.1. An algorithm that determines the data allocation needs to know nothing about the re-organization and its cost.

Because the processing-schedules graph in the decentralized approach is a lot smaller it will be easier to reflect changes in the access pattern by the users.

For real-time applications or for the provision of interactive sessions the third issue is of interest, namely **constraints** queries may require on the response time. For instance, if during a conversation with customers a decision has to be taken based on data in the database a quick response is necessary. The problem we have here is that the cost function can not simply be expressed in transmission cost or response time. In the file allocation problem, which was stated as an integer programming problem, this was solved by just adding more constraints on the allocation. What this means is that the cost to satisfy the demands of a (small) group of users have to be paid by the rest of the users. Whether this is acceptable depends on the importance of the demands and whether it is applicable depends on the composition of the users. For example, in an airline reservation system selling tickets is important and needs of other users should be subject to this. As one can clearly see this is a non-quantifiable aspect and should be decided on by the management of the firm. Having an allocation of the data and the operations such that the response time constraints are met might be expensive. The question is, who will pay for this? If the database is centrally managed because it belongs to one company, this company can decide whether the cost is acceptable. If the database is a collection of databases owned by different companies the problem becomes more complicated. The most reasonable way to solve it is to let the group that wants its demands fulfilled, pay for it. The decentralized approach corresponds more closely with this last philosophy.

The problem discussed here is not just a database problem but a real-life one. For example, the government subsidizes public transportation because many people will benefit from it. But people that have special needs such as special destinations, have to pay more, because they either have to take their own car or a taxi. Problems of this kind are not easily solved and therefore in the case of distributed databases we merely provide the system with tools to handle these problems.

The fourth issue is the **cost** of the data allocation. In discussing the decentralized approach we touched upon this problem. The order in which groups of users construct their processing-schedules graphs and minimize their cost, may influence the total cost of the data allocation. Also, the use of private copies prohibit the sharing of data causing, on the one hand, a higher cost. On the other hand, the cost to update the private copies may be decreased by updating them only periodically. We will show this by an example.

Example 4.5

Take the relation *PARTS* of example 4.3 and two queries

$$Q_1 = PARTS\{CITY = \text{Paris OR } CITY = \text{Amsterdam}\}$$
$$Q_2 = PARTS\{CITY = \text{London OR } CITY = \text{Amsterdam}\},$$

and updates that change the quantity-on-hand of the parts.

In the centralized approach the relation is split horizontally into three fragments as follows. C_1 and C_2 are the two clauses of the queries. Vertically splitting is not considered. The set of attributes, P_1, contains all attributes of *PARTS*.

$$F_{11} = PARTS\{CITY = \text{London}\}$$
$$F_{21} = PARTS\{CITY = \text{Paris}\}$$
$$F_{31} = PARTS\{CITY = \text{Amsterdam}\}.$$

Just for the sake of the example assume that at a site in London (PhS_1) Q_1 is stated, in Paris (PhS_3) Q_2 and the updates come from Amsterdam (PhS_2). The corresponding processing-schedules graph is shown in fig. 4.5, where $VS_1 = (\{F_{11}\},O_1)$, $VS_2 = (\{F_{31}\},O_2)$, $VS_3 = (\{F_{31}\},O_3)$, and $VS_4 = (\{F_{21}\},O_4)$; We also assume that total transmission cost is the cost function.

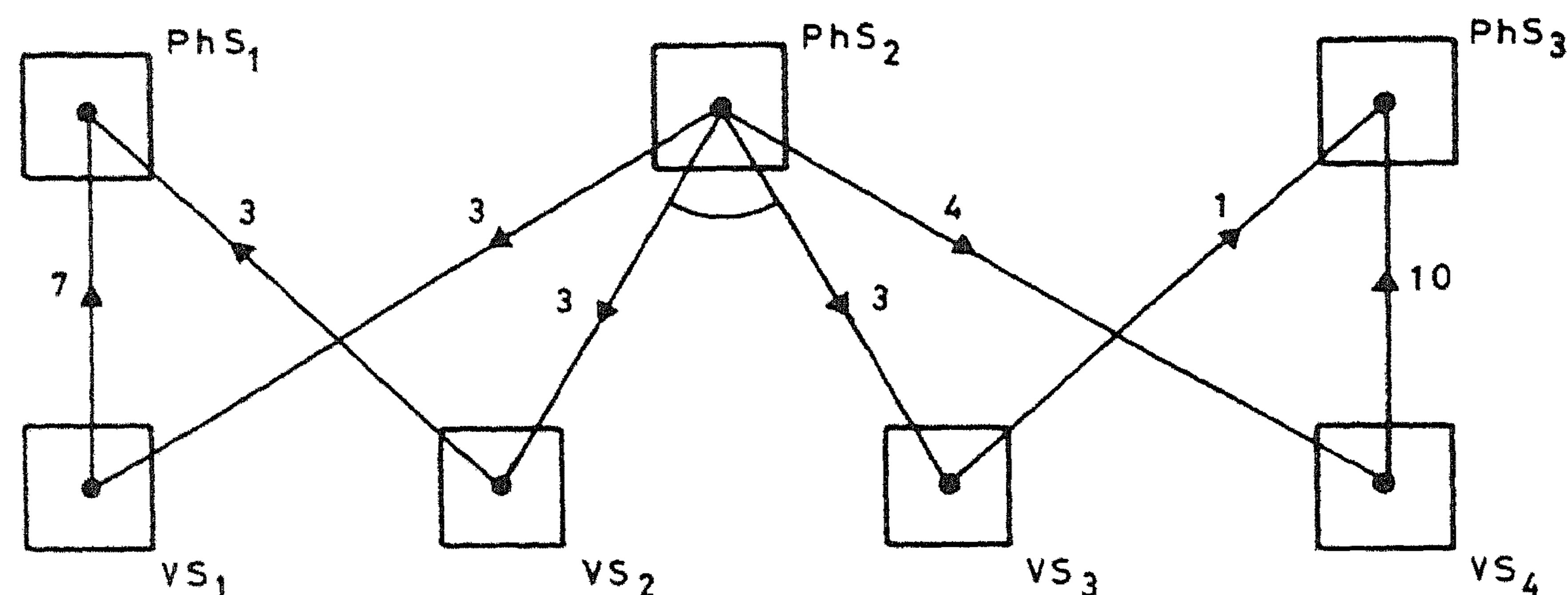


Figure 4.5. Centralized processing-schedules graph.

The optimal allocation is obtained by uniting VS_1, VS_2 and VS_3 and with PhS_1, and uniting VS_4 with PhS_3. Its cost is 11.

In the decentralized data allocation we start with an existing allocation; say relation *PARTS* is located in Amsterdam. In the processing-schedules graphs for the group of users in London and Paris this is shown by allocating *PARTS* to PhS_2.

Because the groups in London and in Paris determine their data allocation independently, the relation is not split. Instead, each group makes a private copy of

the fragment in which it is interested. We discuss the processing-schedules graph for London. The fragment they use is

$$F_L = PARTS\{CITY = \text{London OR } CITY = \text{Amsterdam}\}$$

The two processing-schedules graphs are shown in fig. 4.6; (a) shows the one for London and (b) the one for Paris.

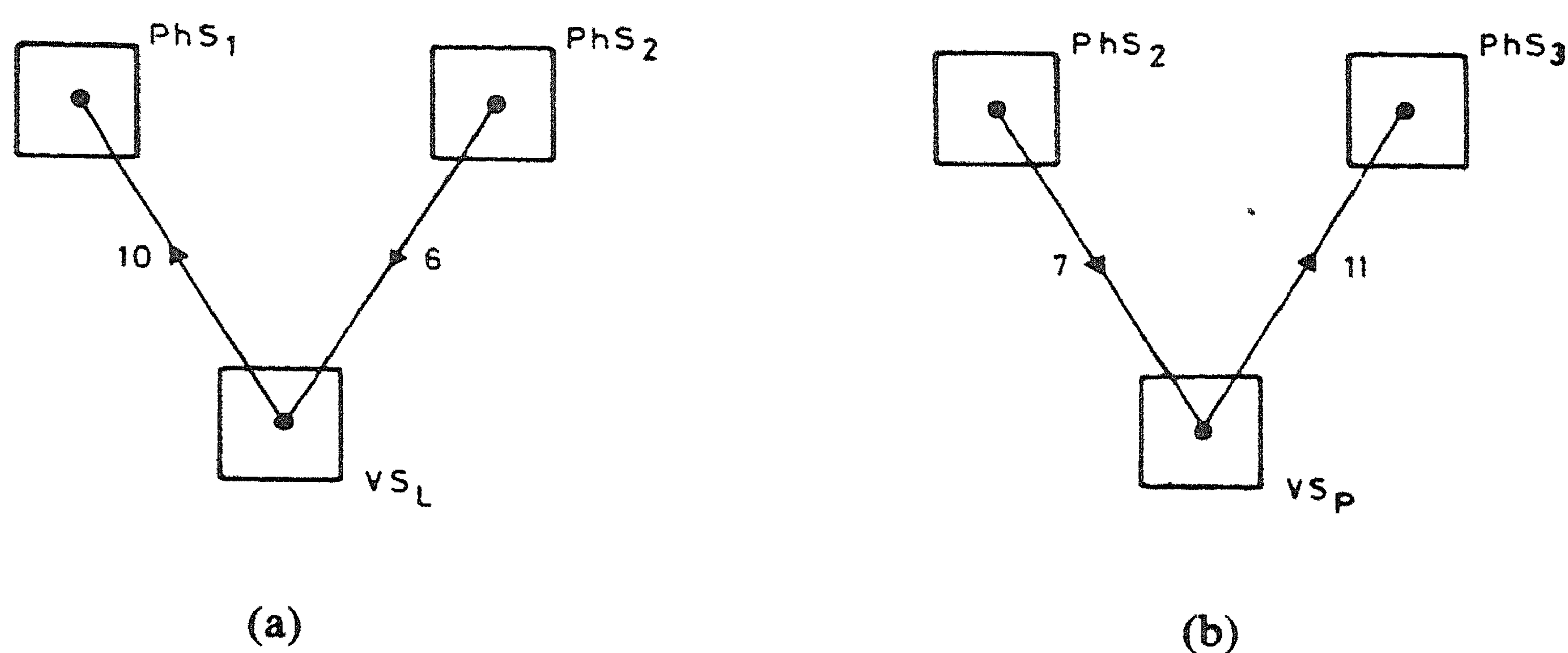


Figure 4.6. Decentralized processing-schedules graphs: (a) for London and (b) for Paris.

The order in which the groups determine their data allocation is irrelevant because of the cost function. The London-group will unit VS_L with PhS_1 and the Paris-group will unite VS_P with PhS_3. The resulting data allocation costs 13.

Comparing the two optimal allocations shows that sharing the fragment that contains parts that are located in Amsterdam, is more beneficial than allowing the two groups in London and Paris having private copies. However, if the group in Paris would only be interested in a periodically updated fragment of relation *PARTS*, the transmission cost from PhS_2 to PhS_3 might be less than 7, because not all updates have to be transmitted and there is less overhead when they are sent together. Hence, in that case the total transmission cost is less than 13.

□

To compute the cost of data allocations obtained by the centralized and the decentralized approach we will do some simulations. For simplicity we confine ourselves to minimizing total transmission cost.

First the way the transactions are generated is discussed. A processing schedule of a transaction will have one of the basic forms shown in fig. 4.7, with its probability that it is generated below it; *compl* is a complexity parameter with which a branch in the processing schedules is generated (except the branch to the result site) and may range between 0 and 1.

Because the relations are used by different transactions they are split into fragments as described in subsection 4.3.2; we assume that they are split into three

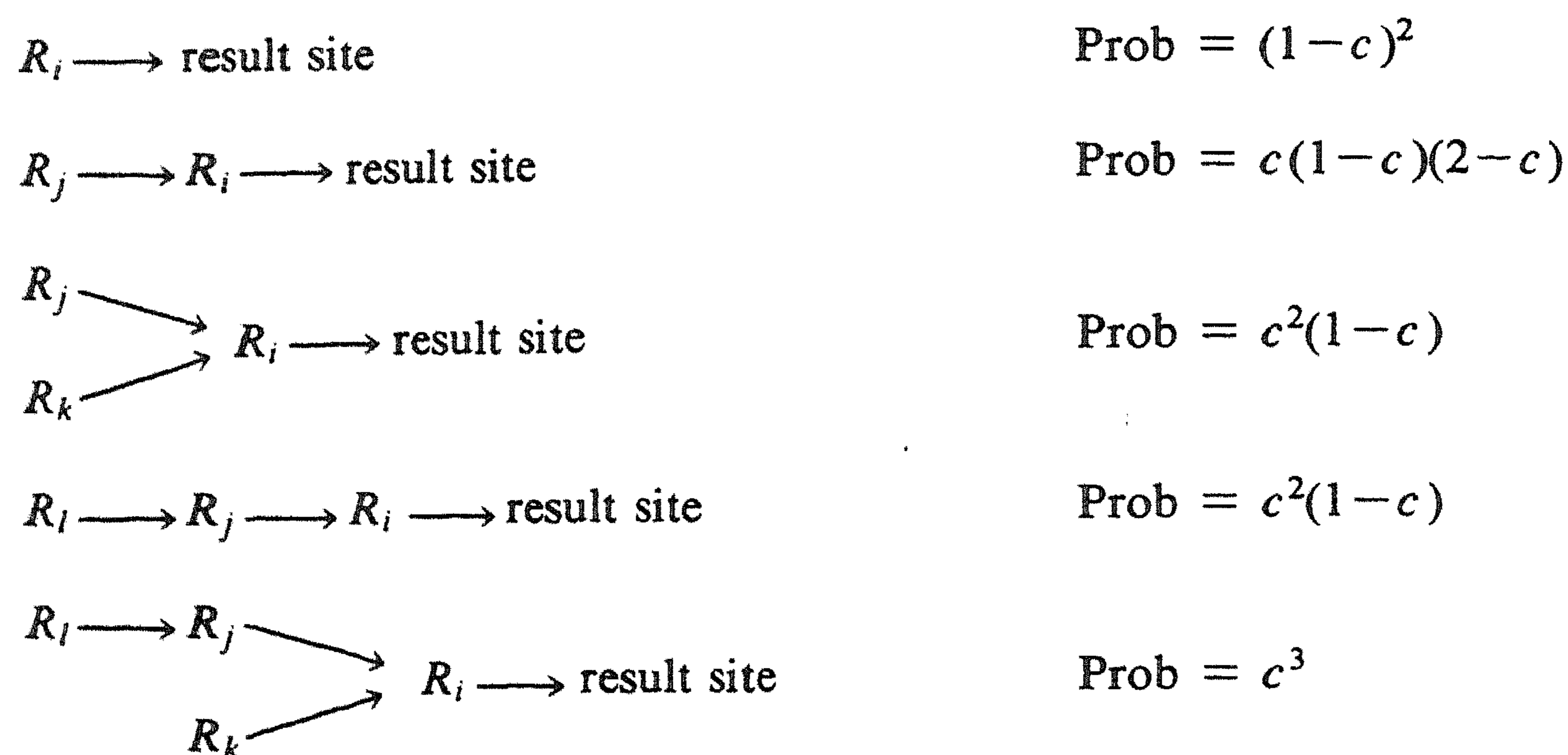


Figure 4.7. Five different processing schedules with the probability that they are generated; $c = compl$.

fragments. When generating the processing schedule for a transaction, for each relation it is decided which fragments are in fact used. Each of the 3 fragments of a relation is used in a transaction with probability *frag*, with a minimum of 1 fragment. During 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 the relevant fragments that were used in the query of the changes. A fragment that is used in the processing schedule of an update transaction is updated with probability *upd*.

In the centralized approach each query and update is given its own set of copies of fragments and for each fragment a virtual site is created. The processing schedules are represented by edges between the nucleus-sites and/or forking graphs in the processing-schedules graph. The virtual site of a fragment that is updated is the notification node in a forking graph and the receiving nodes are the virtual sites of other copies of that fragment. The label of the forking graph is the amount of data required to be transmitted to update a copy at another site times the frequency with which the update is executed.

In the decentralized approach the original relations are not split and all update transactions make use of these relations. The part of the processing-schedules graph that is concerned with queries is the same as in the centralized approach. If an original relation is changed by an update there will be edges from that relation to each of the virtual sites containing private copies used by the queries. The label of each edge is the total amount of data required to be transmitted to update a private copy, taking into account that updates may be done on a periodical basis. Note, that for these updates no forking graph is used because the private copies are not interchangeable, meaning that if two private copies are located at the same site they both have to be updated independently, because they are not necessarily consistent.

The following parameters are fixed:

- the number of sites in the computer network is 5,
- the total number of transactions, $q + u$, is 6,
- the number of relations in the database is 3, and each relation is split into 3 fragments,
- the number of fragments per relation used by a transaction is 1 ($frag = 0$),
- the complexity parameter, *compl* is set to 0.4,
- the edges in the processing schedules are labeled with a random number between 100 and 500,
- all fragments used in a update transaction are updated ($upd = 1$),

The reason that the first four parameters are kept so small is to be able to compute the optimal allocation in a reasonable amount of time.

The parameters that will vary are:

- the number of transactions that are updates (u), ranges from 0 to 6,
- the labels of the transmissions required to keep the private copies consistent in the decentralized approach are multiplied with the parameter *pud*, which denotes the relative cost of periodic updates compared to normal updates. *pud* will vary between 0 and 1.

Table 4.8 shows the total transmission cost for the centralized approach and the decentralized approach for different values of the parameter *pud*. Also the query/update transactions ratio is varied.

		centralized	decentralized			
			pud			
q	*u*		0.0	0.3	0.6	1.0
0	6	2565.3	3357.5	3357.5	3357.5	3357.5
1	5	1605.5	2099.7	2220.4	2282.7	2317.4
2	4	1280.3	1628.2	1831.7	1976.0	2123.2
3	3	1293.8	925.2	1225.2	1449.2	1585.8
4	2	813.1	444.5	793.5	1033.9	1167.2
5	1	291.4	0	160.9	321.8	519.4
6	0	0	0	0	0	0

Table 4.8. Total transmission cost for centralized and decentralized approach.

In the upper half of the table where the updates form a majority the centralized approach is clearly better. This is not so much caused by the decentralized approach itself as by the fact that the relations in the decentralized case were not split and that all update transactions are forced to use the original relations. We will come back to this point when discussing primary copies. In the lower half of the table, where the

queries form a majority, the decentralized approach becomes better if the cost to periodically update the private copies is not too high.

To summarize this section we may conclude that the decentralized approach is rather attractive in a system without a central management, which may change the whole data allocation. It is especially useful for meeting special needs such as response time constraints, and it makes rapidly changing access patterns easy to implement. Also, the usage of periodically updated private copies will decrease the total transmission cost. Compared to the centralized approach, it has the disadvantage that the data allocation can only be optimized locally, which will lead to a higher total transmission cost. Therefore, we expect that hybrid approaches to the data allocation problem will be used, which integrates the centralized and decentralized approach.

4.7. Minimizing Total Transmission Cost

In the previous sections a model was introduced to investigate the data and operation allocation problem. In this section algorithms for computing completely specified allocations, which minimize the total transmission cost, are presented. The total transmission cost of an allocation is the sum of the cost of the schedules in the processing-schedules graph belonging to the allocation multiplied by the frequencies of the corresponding queries. A schedule for a query can be reconstructed from a processing-schedules graph by assuming that physical sites and virtual sites are different sites in a computer network, except for virtual sites that are assigned to other nucleus-sites. Remember that two virtual sites that are assigned to the same physical site are not assigned to each other, implying that, as far as the schedules are concerned, they are different sites in a computer network. Given this reconstructed schedule its cost can be computed.

Both optimal and heuristic allocations are considered in this section when using static and dynamic schedules.

Topics, such as what are the constituents of the total transmission cost and what are the effects of the usage of primary copies, are investigated based on experimental data obtained from heuristic algorithms run on randomly generated processing-schedules graphs.

This section is organized as follows. In subsections 4.7.1 - 4.7.4 the total transmission cost is minimized using static schedules. In subsection 4.7.5 the same is done for semi-dynamic schedules. In subsections 4.7.6 - 4.7.8 dynamic schedules are used and the allocations obtained are compared with the ones obtained using dynamic schedules. In subsection 4.7.9 some extensions of the heuristic algorithm proposed in subsection 4.7.2 are considered.

The main line of research consists of finding admissible heuristic estimators for the Heuristic Path Algorithm to guarantee optimal solutions, developing heuristic algorithms that run in polynomial time, and making a comparison between the allocations obtained by the heuristic algorithm and the optimal ones. In subsection 4.7.10 the constituents of the total transmission cost are investigated. The usage of primary copies is discussed in subsection 4.7.11.

4.7.1. Optimal Allocations Using Static Schedules

When a completely specified allocation is determined using static schedules the processing schedules of all queries and updates are computed based on the initial allocation. Let us consider the processing-schedules graph of this initial allocation. From a graph-theoretical point of view the problem of minimizing total transmission cost boils down to the removal of certain edges in the processing-schedules graph such that there is no "path" left from one PhS-node to another. Then, the virtual sites in the subgraph belonging to a particular physical site are united with that site, resulting in a completely specified allocation. At that point possible constraints concerning bandwidths, etc. can be checked. A way to determine such a completely specified allocation, if total transmission cost is to be minimized, is to define the data allocation problem as an integer programming problem:

$$\text{minimize} \sum_{ij} \sum_{pq} c_{ijpq} x_{ij} x_{pq} + \sum_{ij} b_{ij}(1 - x_{ij})$$

$$\text{subject to } x_{ij} = 0 \text{ or } 1$$

$$\text{where } x_{ij} = \begin{cases} 1 \text{ if } VS_i \text{ is united with } PhS_j \\ 0 \text{ otherwise} \end{cases}$$

$$c_{ijpq} = \begin{cases} \text{total amount of data transmitted between } VS_i \text{ and } VS_p \text{ if } j \neq q \\ 0 \text{ if } j = q \end{cases}$$

and b_{ij} = total amount of data transmitted between VS_i and PhS_j.

The first term of the formula to be minimized is the sum of all the data transmissions between virtual sites that are united with different physical sites and the second term represents the data transmissions between virtual sites and physical sites with which they are not united. This problem looks like the **quadratic assignment problem** only there the b_{ij}'s are zero, and also an additional constraint is added to ensure that only one virtual site is united with one physical site [Lawler1962].

Applications of standard integer programming techniques has not been done because applications to similar problems in the area of physical database design [Hoffer1975], where a large set of objects is manipulated, have shown that these techniques are rather time consuming. For a real database this may not be a drawback as large as in our case, here where we are more interested to investigate the characteristics of the problem.

Other techniques such as **branch-and-bound** [Lawler1966] or the **Heuristic Path Algorithm** [Nilsson1971] can and will be used to search large solution spaces efficiently. In [Pohl1972] it was shown that these techniques are basically the same. These search techniques construct decision trees. A node in such a tree is identified with 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: unite VS_i with

PhS_j. During the search for an optimal data allocation the decision tree constructed so far divides the space of completely specified allocations into subsets which 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 the 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 an **estimate-cost**. Such an estimate-cost is the sum of two components, namely the cost caused by the decisions taken so far and 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.

The search proceeds as follows. At each iteration a leaf with the least estimate-cost is expanded. This means that, given the unions decided on so far to reach that leaf, consider the union of a not yet considered virtual site with any of the physical sites. For each of the physical sites an edge leaving that leaf is created, rendering the leaf into an internal node. For each of the newly created leaves the estimate-cost of the corresponding subset is computed. Then, the algorithm goes through the next iteration, until a leaf whose corresponding subset contains only one completely specified allocation, is expanded. And this allocation is chosen as result.

If the estimate-cost of the subsets are computed such that they underestimate the real cost, then the Heuristic Path Algorithm will eventually find the optimal completely specified allocation [Nilsson1971]. So, this 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 estimate-cost of a partially specified allocation, we will take a look at the basic ideas behind it.

If all virtual sites in a partially specified allocation are directly or indirectly connected with only one physical site the estimate-cost 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, and for each successive pair of nucleus-sites that are united, data transmissions disappear. To underestimate the cost that will be caused by decisions about the virtual sites on such a path, this path is cut in two at the edge that forms the cheapest connection.

The estimate-cost 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 one 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 the edges between the successive nucleus-sites in the sequence defining the path and the removal of the complete forking graphs in which successive nucleus-sites are part, from the processing-schedules graph.

If all paths are removed each virtual site is connected directly or indirectly with only one physical site.

To compute the estimate-cost of the partially specified allocation the algorithm *psa static cost* shown in fig. 4.9 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. The fact that a forking graph is replaced by an edge between the notification node and one of the receiving nodes is quite arbitrarily, because it could be between any of the nodes in the forking graph. Therefore, it should be interpreted as that a forking graph may be used only once in a path.

```
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
    do
        sum +:= cost of that path;
        remove that path
    od;
    unite virtual sites with their physical sites;
    sum
end
```

Figure 4.9. Algorithm *psa static cost*.

Example 4.6

To show how *psa static cost* computes the estimate-cost of a partially specified allocation we will apply it to a simple allocation, shown in fig. 4.10.

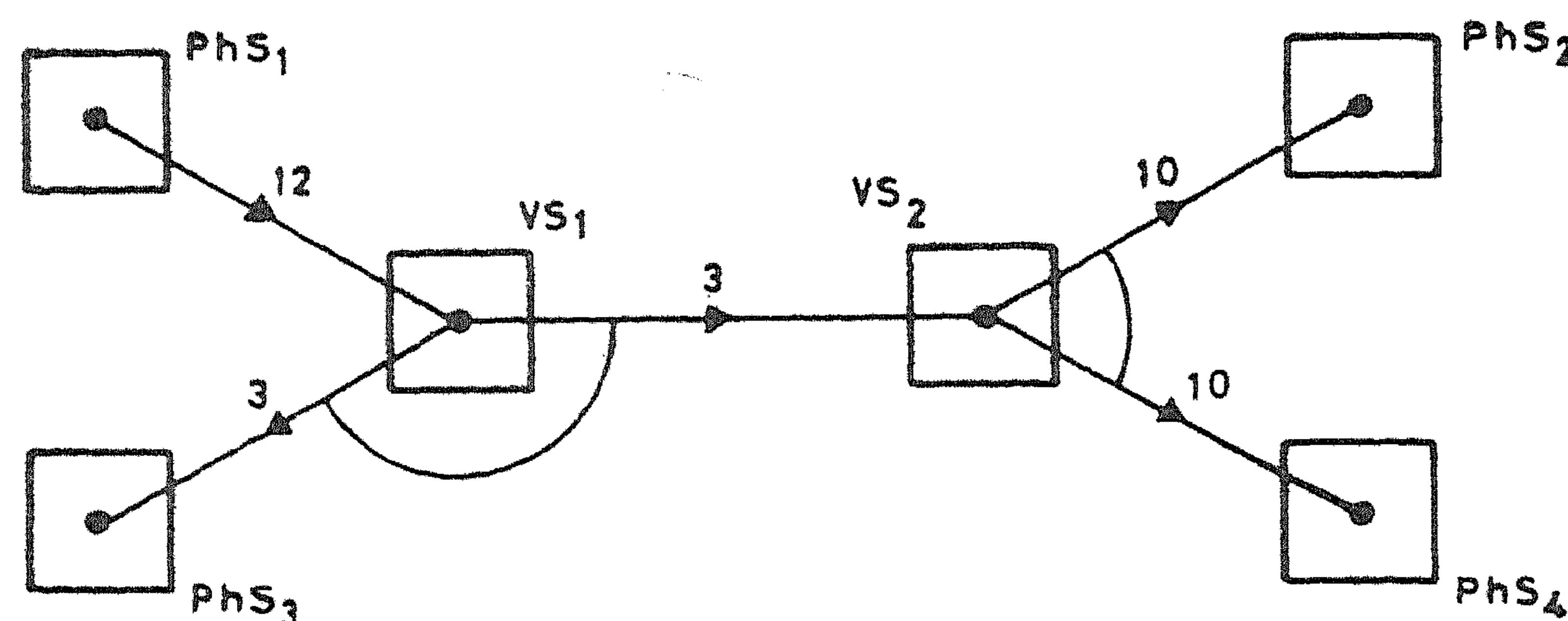


Figure 4.10. Processing-schedules graph.

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 it is 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 estimate-cost is $10 + 3 = 13$.

ne 4 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. An estimator with this property is called **admissible**.

Theorem 4.5 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 can not increase the cost of the partially specified allocation.

Now paths of greater length are considered. Say, NS_0, NS_1, . . . , 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 4.6 If the Heuristic Path Algorithm uses *psa static cost* the completely specified allocations produced have minimum total transmission cost if static schedules are used.

4.7.2. Heuristic Allocations Using Static Schedules

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** [Horowitz1978].

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

- □ virtual sites can not be united with physical sites independently of each other,
- □ 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.

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 data allocation algorithm starts with a partially specified allocation in which every virtual site is assigned 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-value. 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 non-positive. Before VS_u can be assigned, first its LINK-values with other nucleus-sites have to be determined, of course.

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.

In the resulting allocation no two virtual sites will be assigned to the same physical sites. For, 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. Fig. 4.11 shows the procedural form of algorithm *total data allocation*, which minimizes total transmission cost.

We will show by an example how the algorithm works.

Example 4.7

Consider again the real-estate database and the following two queries and two updates:

Q_1 : ((*SELLER*{*CITY* = PARIS})(*PROP#* = *PROP#* AND
LOC = CITY)*PROP*)[*SNAME*,*ADDRESS*,*CITY*,*TYPE*,*LOC*]

Q_2 : (*SELLER*(*PROP#* = *PROP#*)(*PROP*{*LOC* = AMSTERDAM}))
[*SNAME*,*PROP#*,*TYPE*]

U_1 : add new properties to *PROP* located outside Amsterdam

U_2 : delete and add information about sellers.

Because relation *PROP* is accessed in two queries it will be split according to the procedure of subsection 4.3.2. In this case it is split into two, P' and P'', where

P' = *PROP*{*LOC* = AMSTERDAM}
P'' = *PROP*{*LOC* $\neq$ AMSTERDAM}.

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

The processing schedule of query Q_1 will consist of the following data transmissions: the reduced relation *SELLER* after the restriction *CITY* = PARIS, denoted by S_1, is sent to the two fragments P' and P''. The results of the joins, J_1 and J_2, are sent to PhS_1. And the schedule for Q_2: after the projection [*SNAME*,*PROP#*] the relation *SELLER*, denoted by P_1, is sent to fragment P'' where the join, J_3, is computed and the result is sent to PhS_2.

Users at the sites corresponding to PhS_2 and PhS_3 update the fragment P'' and relation *SELLER*.

```
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
```

Figure 4.11. Algorithm *total data allocation*.

For each query and each update, copies of the fragments involved allocated to virtual sites, are interconnected in the update graphs. The resulting processing-schedules graph is shown in fig. 4.12.

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 is assigned to PhS_1, VS_2 and VS_3 to PhS_2, and VS_3 to PhS_3.

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

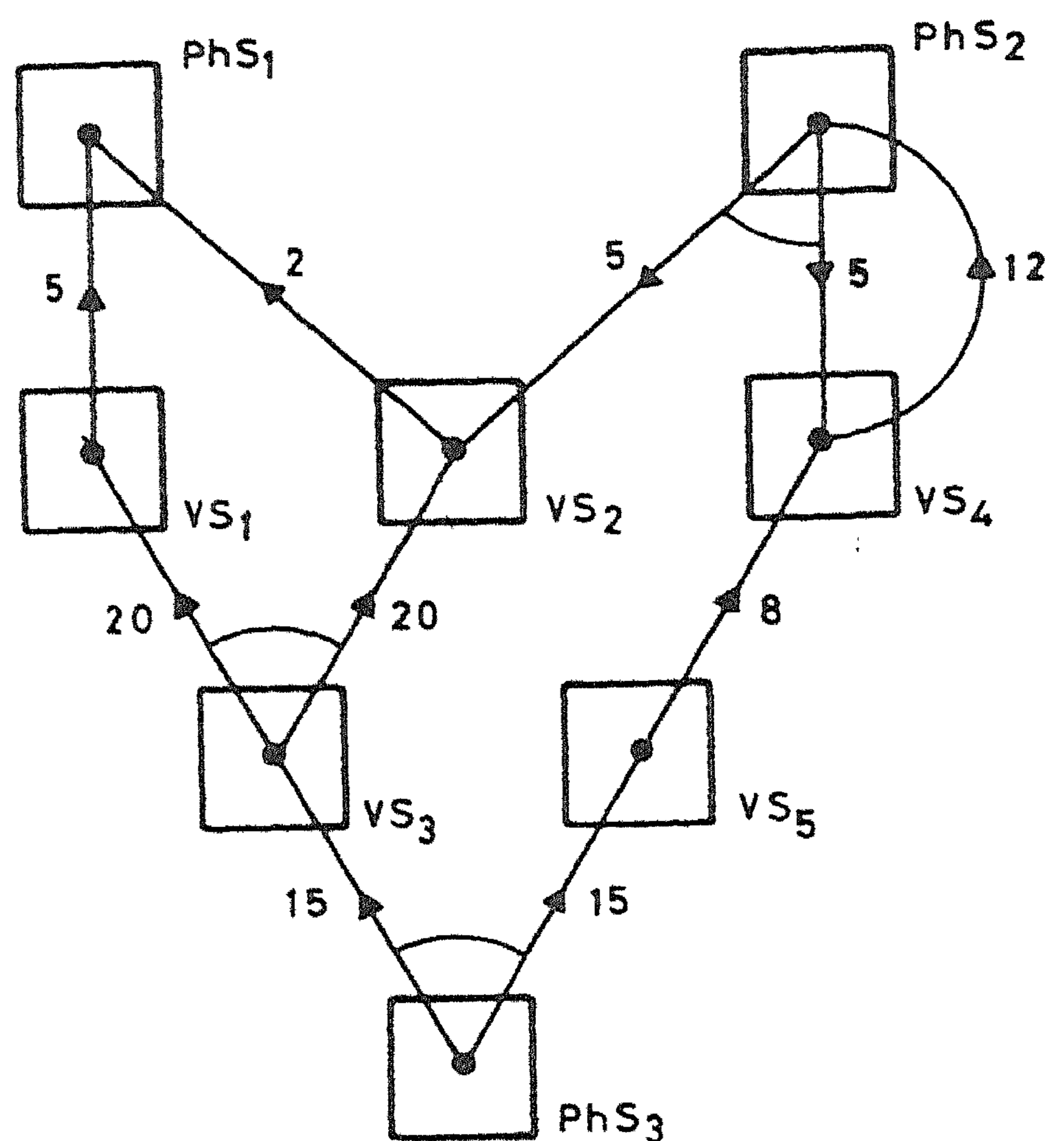


Figure 4.12. Processing-schedules graph of example 4.7.

(VS_1,VS_3)	20
(VS_2,VS_3)	20
(VS_2,VS_4)	5
(VS_3,VS_5)	15
(VS_4,VS_5)	8

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

$$5 + 15.$$

Uniting them and assigning the union, VS_x, to PhS_3 decreases the total transmission cost with

$$20 + 15.$$

The net result, the difference between the two changes, is not positive and, therefore, they are united (see fig. 4.13).

The next pair to be considered is (VS_x,VS_2) whose LINK-value is 20. The net result of uniting them is:

$$5 + 15 - (20 + 15) = - 15 \leq 0.$$

Again, the union, VS_y, decreases the total transmission cost.

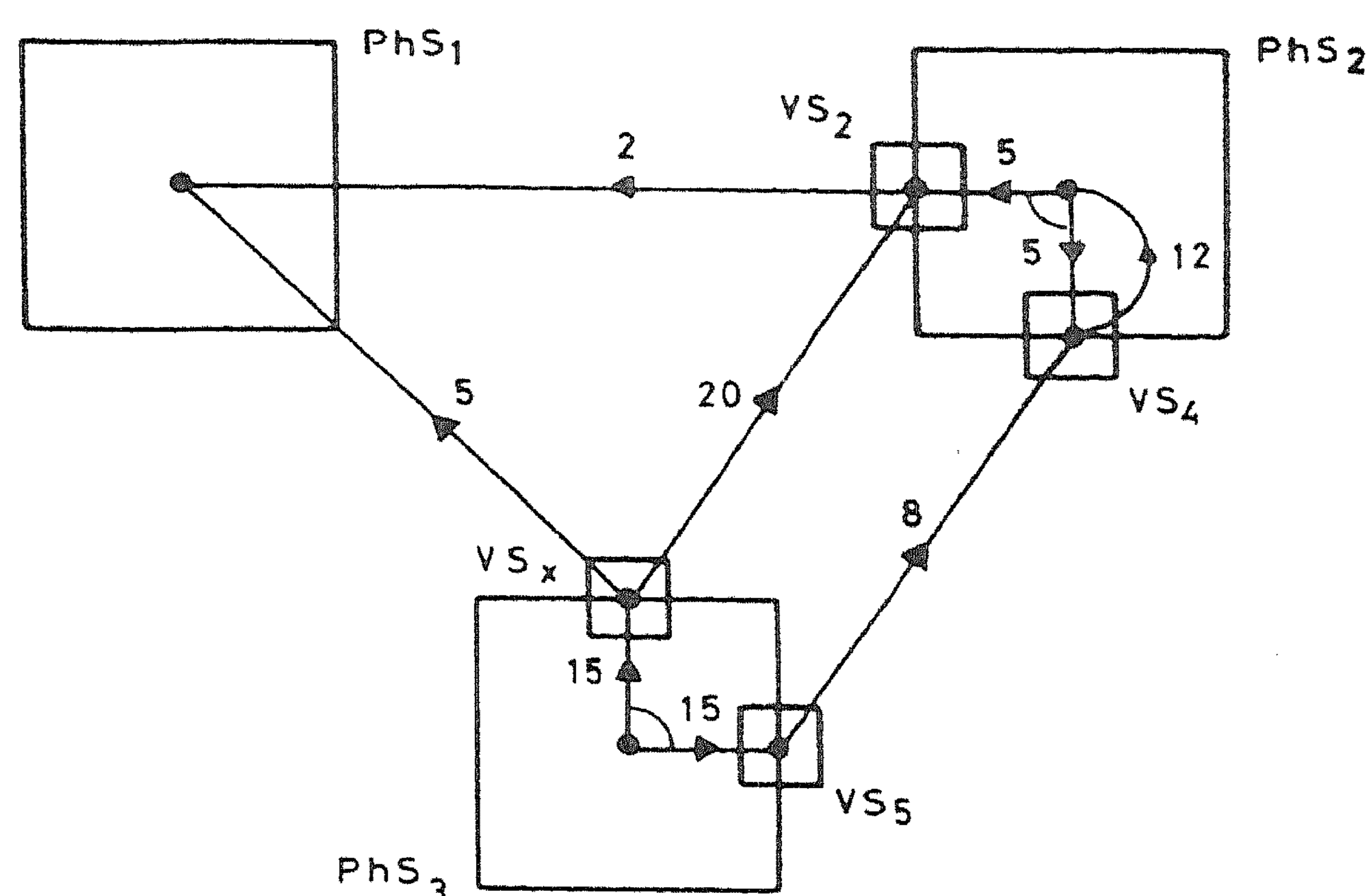


Figure 4.13. Processing-schedules graph after the union of VS_1 and VS_3.

The next pair to be considered is (VS_y, VS_5) whose LINK-value is 15. The net result of uniting them is:

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

which is non-positive and, therefore, the two virtual sites, which contain copies of the same relation *SELLER* are united, VS_z. This means that only one copy will be maintained in the system.

The LINK-value between the two virtual sites VS_4 and VS_z is 13, which is the sum of updates coming from PhS_2 (5) and the data transmissions from relation *SELLER* to P'' (8). Uniting them increases the total transmission cost with:

$$17 + 15 - (13 + 17) = 2 > 0.$$

So, the allocation does not change.

Note, the fact 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_z to PhS_3 and of VS_4 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 is distributed over two sites: the relation *SELLER* is located to PhS_3 and the fragment P'' 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.

□

4.7.3. Theoretical Results Concerning Algorithm *total data allocation*

As was mentioned before the algorithm *total data allocation* is greedy and does, therefore, not necessarily obtain the completely specified allocation with the absolute minimum total transmission cost. However, it is interesting to know how well the algorithm performs. We will show that for a special class of processing-schedules graphs the algorithm computes minimum total transmission cost allocations. 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 be no longer 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.

Theorem 4.7 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 either the algorithm decides to unite two virtual sites that occur in different subsets or the algorithm 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 $\max_k LINK_{jk} \leqslant \max_k LINK_{uk}$

   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:

$$LINK_{jl} - LINK_{ij} \leqslant \max_k LINK_{jk} - LINK_{ij} =$$

$$\max_k LINK_{ik} + \max_k LINK_{jk} - \max_k LINK_{uk} - LINK_{ij} \leqslant 0$$

   b) $\max_k LINK_{ik} \leqslant \max_k LINK_{uk}$ 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 when 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.

□

4.7.4. Comparison 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 in exactly the same way as described in section 4.6. The parameters that will vary are:

- □ the number of update transactions u, which varies from 0 to 4,
- □ the complexity parameter *compl*, 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.

Each of the above parameters will vary while the others are kept fixed at the following values:

$$u = 2 \qquad compl = 0.5 \qquad frag = 0.5.$$

The results are show in table 4.14.

To still be able to compute the optimal allocations, the parameters were chosen rather small. For the processing-schedules graphs generated it took about 5 times longer to compute the optimal allocations compared to the heuristic ones. This may not seem too bad, however, further 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% above the optimal values.

If *compl* 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 *compl* 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

	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	1093.5	1.225	1133.5	1.25
1 3	1336.6	1.25	1380.2	1.225
0 4	1708.8	1.275	1763.6	1.325
compl				
0	162.6	1.05	162.6	1.05
0.25	231.1	1.125	233.1	1.125
0.5	1093.5	1.225	1133.4	1.25
0.75	-	-	1249.9	1.175
1	1474.0	1.2	1491.5	1.125
frag				
0	667.5	1.025	706.8	1.1
0.25	731.1	1.075	761.7	1.225
0.5	1093.5	1.225	1133.4	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

Table 4.14. Comparison results of *total data allocation* and optimal solution.

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 on the whole *total data allocation* computes allocation that have on the average a 3% higher total transmission cost than the optimal one. Also, the number of sites over which the data is distributed per transaction is just a bit more than in the optimal solution. This shows that merely considering pairs of virtual sites to be united is not always enough. We come back to this in section 4.7.9.

4.7.5. Semi-Dynamic Schedules

The advantage of using static schedules is that the computation of the cost of an allocation can be computed efficiently. The main disadvantage is that the cost of an allocation obtained using static schedules might differ considerably from the real cost. By this we mean that if the processing schedules were recomputed given the obtained allocation they could differ from the ones given the initial allocation. In the next subsection dynamic schedules will be discussed. Here, a compromise is made between efficient computation of the cost of allocations and changes in schedules based on changes in the allocation.

How does a schedule change if the allocation changes? For example suppose

that, joins have to be computed between R_1, R_2 and R_3. If the relations are located at different sites, a query processing algorithm may determine a schedule that says: send R_1 to R_2, compute the join between R_1 and R_2, send the result to R_3, compute the join between the previously computed result and R_3, and, finally, send the result to the result site. If static schedules are used, allocating R_1 and R_3 to the same site would not change the order in the computation. So, although, a join between R_1 and R_3 could be computed immediately it is not done. In the **semi-dynamic schedules** that we propose here we will not fix the order in which the joins are computed, in advance. During the computation of an allocation, however, we assume that if two relations referenced in the same query are allocated to the same virtual or physical site, the join between them is computed before joins with relations located at other virtual or physical sites. This may not always be optimal. A consequence is that if R_1, R_2 and R_3 are located at different sites the only thing we know is that 3 transmissions are required to process the joins. And, if R_1 and R_3 are located at the same site only two transmissions are required. In both cases we assume a final transmission to the result site. By not fixing a processing schedule it is not known between which sites data is transmitted and how much; only the number of transmissions can be counted. So, to add more flexibility we have to lower the accuracy of the cost computation. Only counting the number of data transmissions is in some cases acceptable. For example, if transmission time is mainly determined by queueing delays a reasonable objective is to minimize the number of transmissions.

The representation of the semi-dynamic schedules is done by a graph that looks like a forking-graph only the arrows are directed to the nucleus-site, where the result of the joins is required. This may be the result physical sites of the query but may also be just another virtual site. Also, the other nodes in a semi-dynamic schedule have no special function.

The reason why the same type of graph is chosen, is because its behavior under a change in the allocation is the same. For example, if NS_2 and NS_3 are united only one edge to NS_1 remains, and if NS_4 is united with NS_1 the edge between them disappears. The labels of the edges are all the same $(\lambda, 1)$, where λ is the frequency with which the query is stated and 1 stands for 1 transmission.

Optimal solutions can be obtained by letting the Heuristic Path Algorithm use the admissible heuristic estimator *psa static cost*. To compute efficient allocations, algorithm *total data allocation* of subsection 4.7.2 can be used. Also, the optimality proof of theorem 4.7 holds for processing-schedules graphs in which semi-dynamic schedules are used.

4.7.6. Optimal Allocations Using Dynamic Schedules

Having considered static and semi-dynamic 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. As shown in subsection 4.7.1 an estimator, which can easily be computed, can be found 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 can not 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 an estimate-cost of a partially specified allocation can not 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 which 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 estimate-cost of the partially specified allocation.

The procedural form of *psa dynamic cost* is shown in fig. 4.15. An example is given to show how it works.

Example 4.8

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 *PROP* and *SELLER*. There are five one query allocations:

1) $PhS_1 = (\{\})$
   $VS_1 = (\{PROP\})$
   $VS_2 = (\{SELLER\})$,

2) $PhS_1 = (\{\})$
   $VS_1 = (\{PROP, SELLER\})$,

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

Figure 4.15. Algorithm *psa dynamic cost.*

3) $PhS_1 = (\{PROP\})$
   $VS_1 = (\{SELLER\})$,

4) $PhS_1 = (\{SELLER\})$
   $VS_1 = (\{PROP\})$,

5) $PhS_1 = (\{PROP, SELLER\})$.

Because the operations do not matter they are not included in the 2-tuple of the nucleus-sites. 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 $PROP$ to PhS_2, and that no decision has been taken yet about $SELLER$. The one-query-allocations that satisfy this partially specified allocation are 1, 2 and 4. The one with the least cost is taken.

□

Proposition 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 4.8 If the Heuristic Path Algorithm uses *psa dynamic cost* then the completely specified allocations obtained have minimum total transmission cost.

4.7.7. 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; and only after the change the schedules corresponding to the new allocation are computed.

This approach deals with the disadvantage of static schedules 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 non-positive 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.

4.7.8. Comparison Static and Dynamic Schedules

In this subsection a comparison will be made between allocations that are obtained using static schedules and using dynamic schedules. To use dynamic schedules a query processing algorithm is needed. We will take a simple one, namely a variation of algorithm *SERIAL* ([Hevner1979a] and subsection 3.4.2). The

modification is that relations located at the result site of a query do not participate in the schedule. This was mainly done to make the algorithm as efficient as possible. Because computing the optimal allocations is rather time consuming the heuristic algorithms described in the subsections 4.7.2 and 4.7.7 were used. For the static approach the schedules were computed once under the assumption that each relation is located at a different site other than the result site.

Three parameters were varied:

- □ q, the number of queries,
- □ r, the average number of relations,
- □ p, the selectivity of a relation.

The results are shown in table 4.16; each entry is an average of 50 test runs.

q	2	4	6	8	10
dyn	17.2	39.7	63.3	109.1	134.5
stat	1.1	15.6	75.6	56.9	148.6

$r = 3 \quad 0 \leqslant p \leqslant 1$

r	1	2	3	4	5
dyn	296.9	94.3	65.3	5.32	0.34
stat	123.9	96.1	77.6	16.5	26.1

$q = 5 \quad 0 \leqslant p \leqslant 1$

p	0.0-0.33	0.33-0.66	0.66-1.0
dyn	20.1	68.2	128.8
stat	23.0	70.6	137.2

$q = 5 \quad r = 3$

Table 4.16. *TTT* using static and dynamic schedules.

For a small number of queries and a small number of fragments the effect of the modification is noticeable. The dynamic approach is punished because most of the fragments referenced in a query will end up at the result site, and are, therefore, not used in the processing schedules. On the whole, a part from the effect caused by the modification, we may conclude that the total transmission cost of the allocations obtained using dynamic schedules are less than of the ones obtained using static schedules. The efficiency is quite essential because the query processing algorithm must be executed every time a change is made in the allocation for all queries that reference fragments whose location has changed.

4.7.9. Extensions of Algorithm *total data allocation*

Algorithm *total_data_allocation* can be viewed as the most simple version, which can be adapted or extended when applied to more sophisticated models of data allocation. One such generalization has already been discussed in the subsection on dynamic schedules.

Another simple extension is to allow groups of nucleus-sites to be united. In its simple version the algorithm only considers unions of two virtual sites. It can easily be seen that if the processing-schedules graph is not a simple processing-schedules graph there is a chance that uniting three or more virtual sites at once will decrease the total transmission cost, and that pairwise uniting does not [Apers1980a]. By first considering the union of pairs, then triples, etc. the result will definitely improve. However, continuing this until a union of all virtual sites is considered does not necessarily lead to an optimal solution.

Checking constraints is another extension. *Total data allocation* gives no consideration to constraints such as the bandwidth of communication channels or the utilization of CPUs. A straightforward way to implement this is to compute the utilization factor of the resulting nucleus-site whenever two nucleus-sites are united. This can be computed because all operations in that nucleus-site are known. If it is too high, greater than or equal to 1, the site corresponding to the nucleus-site would become saturated, giving an infinite response time. The same can be done for communication channels. After the assignment of a virtual site to a physical site the amount of data transmitted per unit of time to the other nucleus-sites may not violate the bandwidths of the communication channels. If the network has an arbitrary topology the routing of the transmissions need to be determined. Obviously, the algorithm may end up with a non-feasible allocation because of these constraints.

4.7.10. Constituents of Total Transmission Cost

To get a better insight in what effect an allocation has on query processing we take a closer look at the total transmission cost. It is the sum of all transmission costs that remain after the virtual sites are united with physical sites and can be divided into:

- ☐ transmissions required in processing queries,
- ☐ transmissions required in processing updates,
- ☐ transmissions required to keep the copies consistent, denoted by cc.

The first two can again be divided into transmissions that were transmissions between virtual sites and that were transmissions between virtual sites and physical sites in the processing-schedules graph of the initial allocation. They will be denoted by vv and vph and will be suffixed with a q, or u, for a query transaction, or an update transaction, respectively.

So, there are five constituents of the total transmission cost

$$TTC = vphq + vvq + vphu + vvu + cc.$$

We will investigate the changes in these quantities when changing the query/update ratio. Therefore, all the other parameters concerning the structure of the transactions are kept constant:

$$frag = 0.1 \quad compl = 0.6 \quad upd = 0.6.$$

The *upd* parameter may seem high, but it is merely to ensure that an update transaction changes at least one relation. Table 4.17 shows the absolute values of the transmission costs when the number of queries varies from 0 to 10; the total number of transactions is kept constant as well, namely 10. The values are averages of 50 test runs.

q u	*TTC*	*vphq*	*vvq*	*vphu*	*vvu*	*cc*
0 10	3110.4	0	0	2481.9	324.1	304.4
1 9	3048.3	141.9	24.1	2159.1	303.3	420.0
2 8	2764.2	276.7	77.9	1736.8	301.7	371.0
3 7	2505.1	342.9	65.1	1476.0	209.2	411.9
4 6	2022.7	337.4	109.8	1096.6	189.6	289.2
5 5	1814.1	441.3	113.3	843.2	94.6	321.7
6 4	1478.9	399.8	104.7	574.0	76.4	324.0
7 3	1276.1	496.9	68.8	267.4	76.6	368.3
8 2	741.3	297.5	103.2	82.5	13.5	244.7
9 1	397.4	197.5	59.5	17.3	2.6	120.4
10 0	0	0	0	0	0	0

Table 4.17. Constituents of *TTC*.

As one would expect if there are no updates ($u = 0$) every physical site can be united with the virtual sites of all required fragments, and, therefore, no transmissions are required for query processing. Also, because none of the fragments gets updated no transmissions are needed to keep the copies consistent. In the other extreme, when there are no queries, there will hardly be any copies, and, therefore, the total transmission cost consists mainly of the transmissions needed for updates.

When increasing the number of update transactions the amount of data transmitted in update processing grows as well, until it constitutes the whole total transmission cost, together with the transmissions to keep the copies consistent. When increasing the number of query transactions, initially the cost to process queries increases, but then, contrary to when the number of updates increases, it decreases until zero. The reason for the latter is that because there are less updates the cost to maintain copies becomes less and uniting the virtual sites accessed in one query with the result physical site decreases the cost to process queries.

The transmissions required for query and update processing are split into transmissions between virtual sites and between virtual and physical sites to show that even in a completely specified allocation the schedules are not comparable to the schedules used in the file allocation problem.

To get an indication how distributed query processing is after an allocation has been determined, the number of sites over which the fragments required in processing one transaction are distributed, were counted; the results are shown in table 4.18. Sites containing copies of the accessed fragments were not counted; neither were the result sites if they did not contain any referenced fragment. For the given values of the parameter we may conclude that most of the time all the required fragments were located at one site, and in more than 99% of the transactions no more than two sites were involved. We have to keep in mind that for processing-schedules graphs with a different structure the results may be less striking. In [Apers1981a], for example, where transactions were generated in a completely different way a similar phenomenon was observed; there in more than 70% of the transactions no more than two sites were involved. So, we may conclude that, on the average, the fragments accessed are located at no more than 3 sites. Furthermore, query processing at a single site is still an important issue in a distributed database system.

q	u	1 site	2 sites	3 sites	4 sites	av. sites
0	10	0.85	0.142	0.006	0.002	1.16
1	9	0.854	0.136	0.01		1.16
2	8	0.838	0.154	0.008		1.17
3	7	0.874	0.122	0.004		1.13
4	6	0.868	0.122	0.01		1.14
5	5	0.892	0.104	0.004		1.11
6	4	0.904	0.096			1.1
7	3	0.914	0.084	0.002		1.09
8	2	0.932	0.068			1.07
9	1	0.964	0.036			1.04
10	0	1				1

Table 4.18. Number of sites per transaction.

The averages shown in table 4.17 conceal what is really going on. Therefore, we computed cc as percentage of the total transmission cost. The range from 0 to 100% is divided into 10 intervals and for each test run with a non-zero total transmission cost the percentage of the cc is computed and added to the appropriate interval. The resulting frequency diagrams are shown in fig. 4.19 for varying numbers of update transactions. For large q one can clearly see the bi-stable character; for almost all test runs the cc is either 0% or 100% of the total transmission cost. For decreasing q the 100% column starts to diminish and a bulge forms in the middle ($q = 7$). For even smaller q it loses its bi-stable character which can be seen from the disappearance of the bulge and the fact that for almost all test runs the cc is less than 20%. So, for decreasing q the left top moves to the right one, and, finally, they merge. So, in fact the averages of cc shown in table 4.17 are the average of two completely different classes of solutions.

A similar behavior can be found in $vvq + vphq$. The cost of the transmissions to process the updates are rather stable.

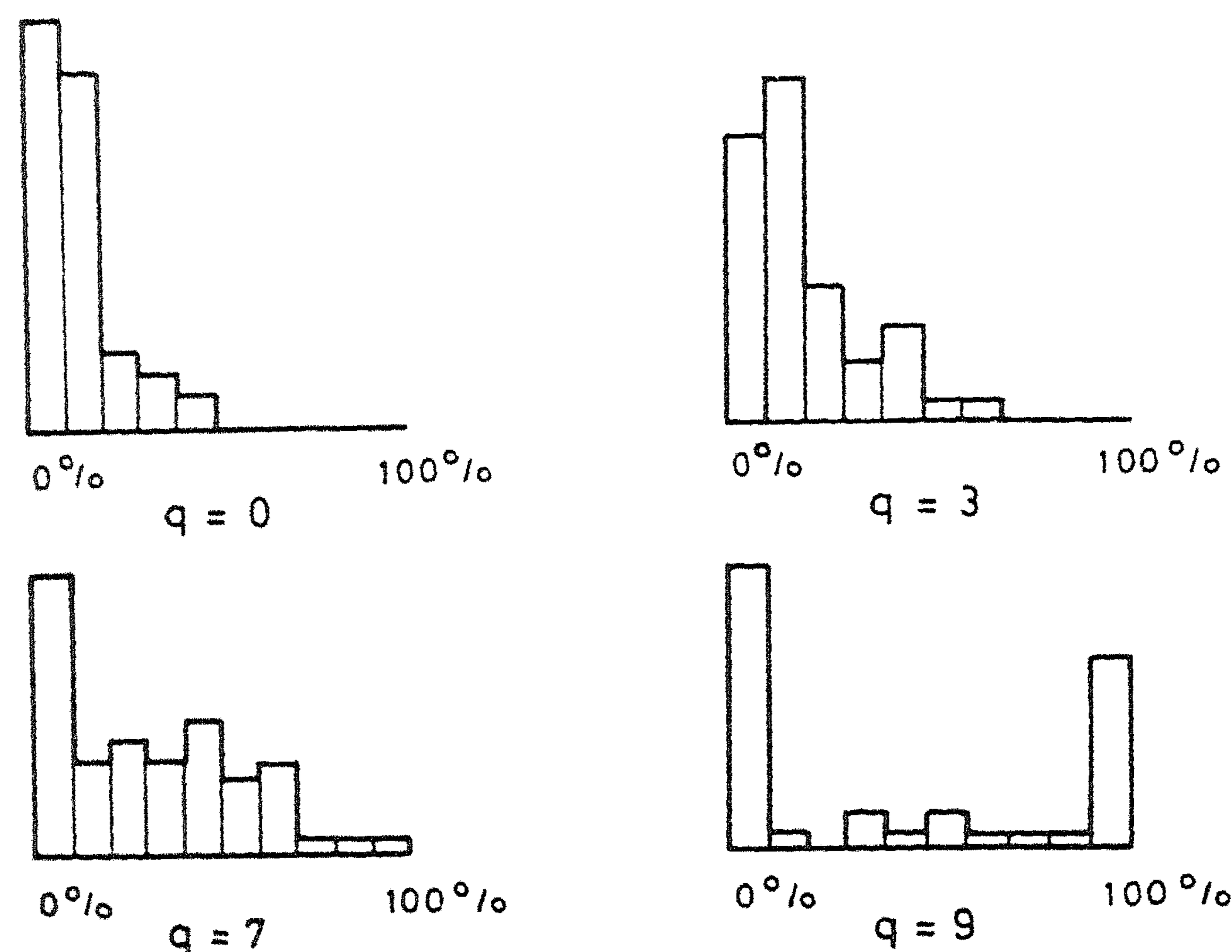


Figure 4.19. Frequency diagrams for varying q.

The two extremes of the bi-stable character can easily be explained. Assume that there are only queries, then there are no transmissions, because every site contains all the required copies. Now replace one query by an update. This update transaction may update a few fragments and transmissions are required to keep the copies all around the network consistent. Two situations can occur. Either nothing is changed in the allocation, implying that there are no transmissions for queries. This explains the fact that query transmission cost goes to 0%. The other situation is when all copies of the updated fragments are discarded except one, implying that there are no transmissions to keep the copies consistent. Hence, the query transmission cost is 100%.

4.7.11. Primary Copies

One of the advantages of maintaining several copies of a fragment is that these copies can be allocated such that transmission and/or response time is minimized. However, if fragments are volatile, meaning that they are updated frequently, the cost of keeping the copies **mutually consistent** might be too high. Whether all copies must be identical all the time (**strong consistency** or whether they will eventually become identical when there are no more updates (**weak consistency**), is not important for the data allocation problem. Important is that the transmissions required for mutual consistency are represented correctly in the processing-schedules graph.

Two ways of representing update transactions are discussed. A simple way is to use two kind of copies for each fragment, namely one **primary copy** and several **secondary copies** [Stonebraker1977]. The idea is to let *all* update transactions use only the primary copy of each fragment. Secondary copies are only used for retrieval

purposes. In the processing-schedules graph of the initial allocation each query transaction will be given its own secondary copy of a fragment.

Another less restrictive way of representing an update transaction is to handle an update exactly the same way as a query when constructing the processing-schedules graph. This means that an update transaction also gets its own set of copies of the accessed fragments. If one of the fragments is updated it becomes a notification node in a forking graph and all other copies, used by both queries and other updates, are the receiving nodes.

The second alternative leaves more freedom to a data allocation algorithm to let update transactions use different copies. Obviously, other design aspects may influence the decision whether primary copies are used or not. Here, we will limit ourselves merely to the effect on the total transmission cost.

To be able to calculate the consequences of the use of primary copies the effect of it on the total transmission cost is compared with the case that no primary copies are used. Table 4.20 shows the results computed by *total_data_allocation* for varying number of update transactions. The fact that update transactions are forced to use the same set of copies clearly has a negative effect on the total transmission cost; if there are only updates this amounts to 8.5%. The negative effect disappears when fewer updates make use of the primary copies and, finally, when there are no updates, it completely disappears. As far as the data allocation problem is concerned there is no need to specify in advance that certain transactions need to use the same set of copies. If it is too expensive to let each transaction have its own copy the data allocation produced will force some of them to share copies.

		total transmission cost	
q	u	no-primary	primary
0	10	3110.4	3374.2
1	9	3048.3	3319.9
2	8	2764.2	3004.6
3	7	2505.1	2748.3
4	6	2022.7	2220.6
5	5	1814.1	1967.9
6	4	1478.9	1633.7
7	3	1278.1	1368.9
8	2	741.3	789.3
9	1	397.4	396.9
10	0	0	0

compl = 0.6 *frag* = 0.1 *upd* = 0.6

Table 4.20. Comparison primary and no-primary.

4.7.12. Summary

The problem of determining completely specified allocations such that the total transmission cost is minimized, was investigated. First, a static approach was considered. The processing schedules of all the queries are computed once, and they will be used throughout the computation of a completely specified allocation. The advantage is that the cost of an allocation can efficiently be determined. A disadvantage is that the computed cost of a completely specified allocation may differ from the real cost. This is caused by the fact that the query processing algorithm might come up with a different processing schedule than the one that was computed at the start. An admissible heuristic estimator, called *psa static cost*, was given to compute optimal allocations with the Heuristic Path Algorithm. Also, a heuristic algorithm, called *total data allocation*, was given, which produces optimal allocations for simple processing-schedules graphs. The basic idea behind it is to unite virtual sites in decreasing order of their LINK-value if the total transmission cost decreases. The solutions obtained by the heuristic algorithm were compared with the optimal ones; the heuristic solutions had, on the average, only a 3% higher total transmission cost than the optimal ones.

Secondly, a semi-dynamic approach was treated. If only the number of data transmissions are counted the schedules can be represented by a subgraph that looks like a forking graph. On the one hand, this gives a more flexible approach towards the changing schedules, and, on the other hand, the same results hold as for the static approach.

Thirdly, a dynamic approach was discussed. To determine the real cost of a partially or completely specified allocation the processing schedules of the queries have to be computed given this allocation. A consequence is that to determine the optimal completely specified allocation, all possible allocations of the relations have to be considered. An admissible heuristic estimator was given. We expect, however, that the computation of an optimal solution using dynamic schedules is too time consuming. Therefore, a heuristic algorithm was proposed, which is a variation of *total data allocation*. It does not use a processing-schedules graph but a LINK-graph. The labels of the edges denote the cost if the two adjacent nucleus-sites are not united. The static and the dynamic approach were compared. We may conclude that the allocations obtained using dynamic schedules have a lower total transmission cost; however, the cost to obtain these allocations highly depends on the efficiency of the query processing algorithm used.

To get a better insight in the allocations obtained the total transmission cost was investigated for varying the query/update ratio. The *TTC* starts from zero, if the are no updates, and gradually grows, if the number of updates increases. If there are mainly queries there is either a non-redundant allocation or a fully redundant allocation for the fragments that are updated. Also, part of the *TTC* is constituted by the cost for transmissions between virtual sites in the initial allocation, showing that even in a completely specified allocation the schedules still differ from the ones used in the file allocation problem. From this we may conclude that solutions obtained from the file allocation problem do not characterize solutions of the data allocation problem in a distributed database. Furthermore, the usage of primary copies was investigated. As far as data allocation is concerned there is no need to specify in advance that an update must access a certain copy of a fragment. On the contrary, forcing all updates to use the same copy will increase the total transmission cost.

4.8. Minimizing Average Response Time

In the previous section we considered the total transmission cost as the cost function to be minimized. Here we will discuss the average response time of the queries. Let λ_i be the frequency with which the ith query is stated and let RT_i be its response time then the average response time is:

$$ART = \frac{\lambda_1 RT_1 + \lambda_2 RT_2 + \cdots + \lambda_m RT_m}{\lambda_1 + \lambda_2 + \cdots + \lambda_m}.$$

The response time of a query can be determined by reconstructing its schedule from the processing-schedules graph of an allocation and computing the queueing delays at the CPUs and communication channels.

In the subsections to come, both the response transmission time and the response processing time are investigated, as well as the response time, which includes both transmissions and processing. When discussing processing time it is not only important to know where the fragments are located, but also at which sites the operations, required for processing the queries, are executed. Consider a query that computes the join between two restricted relations. The restrictions are, of course, executed at the sites where the fragments are located, however, the join may be computed at either site or at the result site. Fixing an operation allocation is similar to determining processing schedules. On the one hand, one may consider it as a form of precompiling the schedules, on the other hand, it can also be used to see whether a particular data allocation is feasible, i.e., having an acceptable average response time. In the latter case, the schedules can be determined at run time, with the advantage that utilization factors, etc. can be taken into account.

The advantages of allowing for a redundant data allocation have been discussed extensively in the previous section. Therefore, and for reasons of simplicity, we confine ourselves in this section to a non-redundant data allocation. Also, only static schedules are used.

In this section both minimizing response processing time and response transmission time are considered. In subsection 4.8.1 a queueing model is introduced to be able to compute queueing times given an operation allocation. In the subsections 4.8.2 - 4.8.10 simple processing schedules will be considered. First, in subsections 4.8.3 - 4.8.5 minimizing average response transmission time for these simple processing schedules is treated. Then, in subsections 4.8.6 - 4.8.9 average response processing time is minimized. In subsection 4.8.10 the two cost functions are combined. In the subsections 4.8.11 - 4.8.14 the average response transmission time is minimized for arbitrary processing schedules. In subsection 4.8.13 an algorithm is given, which determines an operation allocation given a data allocation. In subsection 4.8.14 this algorithm is used in an algorithm, which computes both a data and an operation allocation to minimize average response transmission time.

The main line of research consists of establishing, where possible, the complexity of the problem, finding an admissible heuristic estimator to guarantee that the Heuristic Path Algorithm produces optimal solutions, and developing heuristic algorithms that run in polynomial time. In some cases a comparison is given between solutions obtained by the heuristic algorithm and the optimal solutions.

4.8.1. Queueing Model and Response Time

A nucleus-site consists of two sets: one for fragments and one for operations. The latter contains operations that will be executed at a site of the computer network. To compute the response time of a query we have to know the expected queueing times of the different sites and the different communication channels involved. Straightforward application of queueing theory to compute the transmission time of messages [Kleinrock1975a, Kleinrock1975b, Jackson1957] can not be done, because of the forking and synchronization points in processing schedules, which cause dependencies between transmissions and operations in one query. A partial solution to this problem was discussed in subsection 3.1.6. Based on that, the response time of a query will be computed algorithmically, by serializing the operations and transmissions that share the same resources. In between queries we will assume independence. It should be noted that none of the algorithms presented here makes explicit use of the way queueing times are computed and, therefore, any queueing model can be used as well. However, some of the algorithms need a fast computation of the expected queueing times, which prohibits the use of a simulator.

Lacking a queueing model for the whole network that takes into account the dependencies caused by the schedules, we assume that Jackson's Independence Theorem [Jackson1957] can be applied, implying that a queueing model can be taken for each server, and that the queueing times can be computed for each one separately.

The model used for a CPU is M/G/1 with bulk arrivals. This means that there is a single server with Poisson arrivals and arbitrary service time distribution. Bulk arrival means that groups of jobs are placed in the queue at the same time. The reason why we need this is that more than one operation may be initiated by a query at one site. If, for example, two fragments that are accessed in the same query are located at the same site, the operations performed on both of them will be initiated at the same time.

No attempt has been made to better implement the special characteristics of database operations in the queueing model. For example, no distinction is made between the use of a CPU and of IO devices. Again, this is only done to focus our attention on the algorithms; the formulas to compute queueing times are merely tools.

We assume that we know how frequently a query is executed; this will be denoted by λ_i for query Q_i. Each operation that is part of a query is executed with the same frequency. Also, the execution time (service time) of an operation is assumed to be known; the execution time of operation O_j is denoted by x_j. So, for every nucleus-site we have a set of operations to be executed and for each operation O_j of each query Q_i we know its λ_i, its x_j, and the query of which it is a part. From this we will compute the expected queueing times of the sites.

Three distributions are involved: the **bulk size distribution**, the **arrival distribution** and the **service distribution.** The **coefficient of variation** of an arbitrary distribution X is defined as

$$C_X^2 = \frac{\overline{X^2} - \overline{X}^2}{\overline{X}^2},$$

where $\overline{X}$ and $\overline{X^2}$ are the first and second moment of the distribution X. Because nothing is known about the distribution of the bulk sizes and service times we assume that the operations that are executed at a site form a sample and, therefore, the first

and second moment can be estimated by the sample mean and variance. The arrival distribution is really a whole set of distributions, namely one for each group of arrivals. If we assume that the arrival of each group can be described by a Poisson distribution, then we can use the property that the sum of Poisson distributions is again a Poisson distribution, only now with varying group size.

The formula for the expected queueing time of the first operation in a group is

$$W_g = \frac{\rho \bar{x} \bar{g}}{2(1-\rho)} \left[1 + \frac{C_b^2}{\bar{g}} + C_g^2 \right],$$

where

ρ is the utilization factor,
$\bar{g}$ is the average bulk size,
C_g^2 is the squared coefficient of variation of the bulk size distribution,
$\bar{x}$ is the average service time
C_b^2 is the squared coefficient of variation of the service time distribution [Kleinrock1975a].

Other servers for which a queueing model can be used are the communication channels. A simple M/M/1 model seems adequate to compute queueing delays [Kleinrock1975a]. Quite often one can observe that these delays are rather constant until a certain threshold is passed by the utilization factor. Then they rapidly grow to infinity. In the context of this research we will confine ourselves to this simple model to represent transmissions, although there are no real limitations to compute the delays based on the real transmissions. Therefore, the transmission time of each communication channel is described by

$$TT(X) = T_0 + TC(X),$$

where T_0 is a delay constant and TC is the transmission cost function.

Given a completely specified data and operation allocation the queueing delays of the physical sites can be computed. The response time of a query can now be determined by reconstructing its schedule from the processing-schedules graph and serializing the transmissions and operations that share the same resource. The average response time of all queries and updates can then be computed by weighing all the response times.

For a partially specified allocation only the queueing times of a physical site can be computed based on its operation-set and the virtual sites assigned to it. The response time of a query is determined based on these queueing times and the transmissions between physical sites and between physical sites and virtual sites assigned to other physical sites.

4.8.2. Simple Processing Schedules

Most research in the area of the file allocation problem has concentrated on minimizing the total transmission cost. To establish the complexity of minimizing *ART* and to find general techniques which can also be applied to more complex processing schedules, we will first consider simple schedules.

The **simple schedules** consist of local processing at the sites where the data are located, followed by transmissions to the result site where, for example, a join is

computed. These simple schedules imply that operations within one query ar executed either at a site that contains data referenced by the query or at the result site. In subsection 4.8.11 arbitrary schedules are considered which makes it possible to offload some of the operations to other sites.

Our goal is to unite the virtual sites with the physical sites such that the average response time is minimized. Because the operations have already been allocated we will speak of the **data allocation**. First the complexity of the problem will be established.

Imagine that transmission delay and processing time are zero. So, the only thing that counts is the queueing delay before transmission. The edges for query Q_i in the processing-schedules graph will be labeled with $(i,1,1)$. This means that the response time of a query is either 0 or 1 depending on whether all the accessed data are local or not.

Theorem 4.9 The problem of minimizing the average response time is NP-complete.

Proof We will show this by translating the set packing problem to it.

> **Set Packing Problem:** given a collection of sets $\{S_1, S_2, \ldots, S_n\}$ determine whether there exist l disjoint sets S_i.

This problem is known to be NP-complete [Karp1972, Garey1979] It is translated to the response time problem as follows. Define for each element in each of the S_i's a virtual site, and for each set S_i a physical site PhS_i. In the processing-schedule graph all elements in one set S_i are connected by an edge with physical site PhS_i; the label of the edge is $(i,1,1)$. If S_i and S_j are disjoint we know that the virtual sites corresponding to the elements of S_i can be assigned to PhS_i and those corresponding to the elements of S_j to PhS_j resulting in a zero response time for both the query stated at PhS_i and PhS_j.

Asking for l disjoint sets S_i is the same as asking for an allocation such that l queries have zero response time, which is the same as $n - l$ queries with response time equal to 1.

Assume we are given an assignment of virtual sites to physical sites such that the $ARTT$ exactly equals $(n - l)/n$. Because a response time of a query can only be 0 or 1, there must be l queries with zero response time. Each of these l queries corresponds to a set S_i, and these sets must, therefore, be disjoint.

The other way around goes similarly.

□

4.8.3. Optimal Allocations to Minimize Average Response Transmission Time

We will again use the Heuristic Path Algorithm to compute the allocation that has minimum $ARTT$ for simple processing schedules; for simplicity, we assume that all λ_i's are 1. So, minimizing $ARTT$ is the same as minimizing the sum of the RTT_i of the queries.

The heuristic estimator that we will describe now is called *psa resp data*. Assume we are given a partially specified allocation. For each query we will use a RTT'_i, which is initially set to the maximum of transmission times of data coming from other physical sites. Later we will prove that the RTT'_i of a query is an upper bound on its RTT_i, given the partially specified allocation. Now for a virtual site VS

its **contribution** to the sum of the response transmission times is computed. First, the increase of each RTT'_i is computed if *VS* were not assigned to any of the physical sites to which it transmits data. Take the sum of these increases and subtract the maximum. This is the contribution of *VS*. Before considering the next virtual site, *VS* and its adjacent edges are removed from the processing-schedules graph, and the RTT'_i are updated by adding their respective increases. This continues until no virtual sites are left, and the sum of the contributions of the virtual sites plus the sum of the initial values of the RTT'_i is the estimate-cost of the partially specified allocation computed by *psa resp data*.

```
proc psa resp data=(schedules graph psg)real:
begin
   [1:n]int RTT fm;
   real sum := 0;
   set RTT'i to maximum of transmission times of data coming from
   another physical site than i;
   while there is a virtual site
   do
      say VS is such a virtual site;
      foreach RTT'i
      do
         compute its increase if VS were not assigned to the correspond-
         ing physical site and update it by adding this increase
      od;
      contribution of VS is the sum of the increases of the RTT'i minus
      the maximum of these increases;
      sum +:= contribution of VS
   od;
   sum
end
```

Figure 4.21. Algorithm *psa resp data*.

Example 4.9

Assume we have a processing-schedules graph and in the search tree the decision has been taken to unite VS_1 with PhS_1. Then RTT'_2 equals 3 and both other RTT'_i are zero. This situation is shown in fig. 4.22.

The increases of the RTT'_i caused by VS_2 are:

$RTT'_1 : 2 - 0 = 2$,
$RTT'_2 : 6 - 3 = 3$, and
$RTT'_3 : 4 - 0 = 4$.

So, the contribution of VS_2 to the sum of the RTT_i is $2 + 3 + 4 - 4 = 5$. The RTT'_i are updated as follows: $RTT'_1 = 2$, $RTT'_2 = 6$, and $RTT'_3 = 4$.

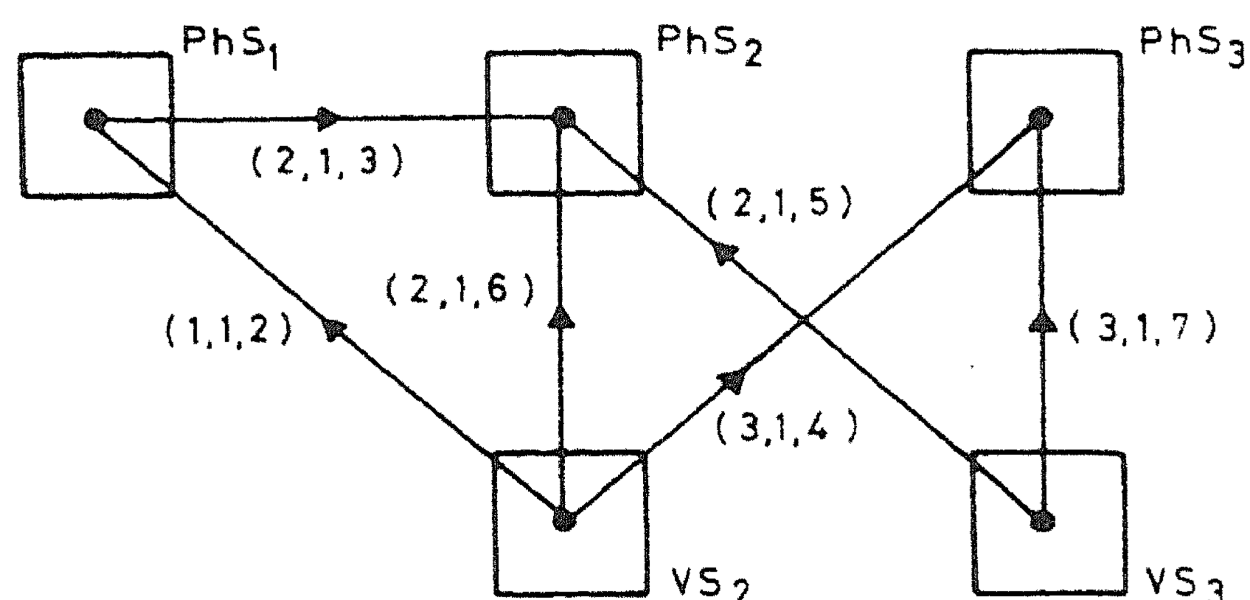


Figure 4.22. Processing-schedules graph belonging to a partially specified allocation.

The contribution of VS_3 is $\max(0,5 - 6) + (7 - 4) - (7 - 4) = 0$. So, the sum of the contributions equals 5, and if we add the cost caused by previous decisions (3) we end up with an estimate-cost of 8. This also happens to be the cost corresponding to the optimal solution, satisfying the constraint that VS_1 is united with PhS_1. In this optimal solution both VS_2 and VS_3 are united with PhS_3.

□

Theorem 4.10 The heuristic estimator *psa resp data* is admissible if the simple processing schedules are not serialized.

Proof Consider a partially specified allocation. What we will do is consider the virtual sites in the order done by *psa resp data* and unite them with the physical site according to the optimal solution satisfying the previously taken decisions, and compute their contribution to the sum of the response transmission times of this optimal solution. Let $RTT_j^{(l)}$ and $RTT_j'^{(l)}$ be the values of the RTT_j and RTT_j' after l virtual sites have been united with physical sites. To start with, set both $RTT_j^{(0)}$ and $RTT_j'^{(0)}$ to the maximum of the transmissions of data accessed by Q_j coming from other physical sites. Because we are given a partially specified allocation their values may be non zero. Let $R(VS_i,PhS_j)$ denote the result after local processing at VS_i, which is transmitted to PhS_j. Let VS_i be a $l + 1$-th virtual site for which the contribution is computed by the estimator. Assume VS_i is united with PhS_k in the optimal solution then its contribution to the optimal sum of response transmission times is:

$$\sum_{\substack{j=1 \\ j \neq k}}^{n} \max(TT(|R(VS_i,PhS_j)|) - RTT_j^{(l)},0).$$

The contribution computed by the estimator is:

$$\sum_{j=1}^{n} \max(TT(|R(VS_i,PhS_j)|) - RTT_j'^{(l)},0) - \max_j(\max(TT(|R(VS_i,PhS_j)|) - RTT_j'^{(l)},0))$$

If $RTT_j'^{(l)} \geqslant RTT_j^{(l)}$ then the terms in the summation in the first formula are all greater than the corresponding terms in the second formula, and the kth term, which is not counted in the first formula, is less than or equal to the term that is subtracted in the second formula. Hence, the contribution computed by the estimator is less than or equal to the real contribution.

So, we only have to show that the $RTT_j'^{(l)}$s are always greater than or equal to the RTT_js. Before the first iteration $RTT_j'^{(0)} = RTT_j^{(0)}$ for all j. Now assume the induction hypothesis holds after $l-1$ virtual sites have been assigned, i.e., $RTT_j'^{(l-1)} \geqslant RTT_j^{(l-1)}$. After the assignment VS_{i_l} to PhS_k both $RTT_j^{(l)}$ and $RTT_j'^{(l)}$ are computed.

$$RTT_j^{(l)} = \begin{cases} RTT_k^{(l-1)} & j = k \\ \max(TT(|R(VS_{i_l}, PhS_j)|), RTT_j^{(l-1)}) & \text{otherwise} \end{cases}$$

and,

$$RTT_j'^{(l)} = \max(TT(|R(VS_{i_l}, PhS_j)|), RTT_j'^{(l-1)}) \qquad \text{for all } j$$

From this we can conclude that $RTT_j'^{(l)} \geqslant RTT_j^{(l)}$ for all j.

Because $RTT_j'^{(l-1)} \geqslant RTT_j^{(l-1)}$ for all j the estimator will underestimate the real cost. Hence, *psa resp data* is admissible.

□

Corollary 4.11 Algorithm *psa resp data* is an admissible estimator if the simple processing schedules are serialized.

Proof This is true because the response transmission time of a serialized simple processing schedule is larger than a non-serialized one.

□

Corollary 4.12 If the Heuristic Path Algorithm uses *psa resp data* the completely specified allocations produced for simple processing schedules have minimum average response transmission time.

4.8.4. Heuristic Allocations to Minimize Average Response Transmission Time

In case the response transmission time of a query is completely determined by the data transmissions involved, the response time of a query with a simple processing schedule is determined by its largest data transmission. If two files that are accessed in one query are located at the same site, the results obtained after local processing on both files have to be transmitted along the same communication channel to the result site. Because this can not be done in parallel one result will have to wait until the transmission of the other result has been completed. So, to compute the response transmission time of a query the schedule must first be serialized. Here, this serialization is simple because the order in which the results are transmitted is irrelevant as far as the response transmission time is concerned.

In the following we assume that per physical site only one query is stated. This is not a severe limitation, it is merely to make the figures and examples simpler.

The fact that even the simplest imaginable problem of minimizing average response transmission time is NP-complete, does not give us much hope of finding an algorithm that produces optimal allocations for a special class of processing-schedules graphs. Therefore, we will settle with a simple heuristic algorithm.

If we look at the set packing problem, an obvious algorithm to find suboptimal solutions is to start with the collection of the sets and at each iteration selecting a set S_i with the smallest number of sets that have a non-empty intersection with it, and throwing S_i and all the intersecting sets out of the collection.

A similar approach can be taken for our allocation problem if no attention is given to transmissions in one schedule that share the same communication channel. If all edges are labeled with $(i,1,1)$, where i stands for the corresponding query, then the algorithm runs as follows. At each iteration, take a query whose RTT_k has not been determined yet and that shares its fragments with the fewest number of other queries. Set its RTT_k to zero, and the RTT_j of the other queries that use the same fragments to 1.

In general, the labels are of the form (i,λ_i,x), where λ_i and x are arbitrary numbers. The algorithm to be described now is called *simple resp data allocation*. At each iteration, it decides to unite one virtual site with one physical site. Which virtual site and which physical site is determined by considering all possible alternatives. To start with, it sets all RTT_j equal to zero. The increase of the $ARTT$ caused by the union of VS and PhS is computed by assigning VS to PhS and determining the increases of the response transmission times of the other physical sites. From these increases of the RTT_j the increase of the $ARTT$ can be computed.

At first glance, it looks better to consider the union of several virtual sites to one physical site. The advantage would be to take several virtual sites that are accessed in the same query, say Q; under the Parallelism Assumption the virtual site that transmits the least amount of data to the result site of Q could be united with the other physical site at no cost. However, because the schedules are serialized, considering more than one virtual site to be united with one physical site will always increase the $ARTT$ more than if only one virtual site was considered.

We will show how the algorithm works by an example.

Example 4.10

There are three queries; let Q_i be stated by users of PhS_i. We assume that the λ_i's are all 1. Hence, minimizing the average response transmission time is the same as minimizing the sum of the response transmission times. The processing-schedules graph is shown in fig. 4.23. The transmission time is assumed to be $TT(X) = 10 + X$.

In the table below we give the possible assignments that are investigated by algorithm *simple resp data allocation* and their increase of the sum of the response transmission times.

assign	increase sum
VS_1 to PhS_1	$70 + 90 = 160$
VS_1 to PhS_2	$60 + 90 = 150$
VS_1 to PhS_3	$60 + 90 = 150$

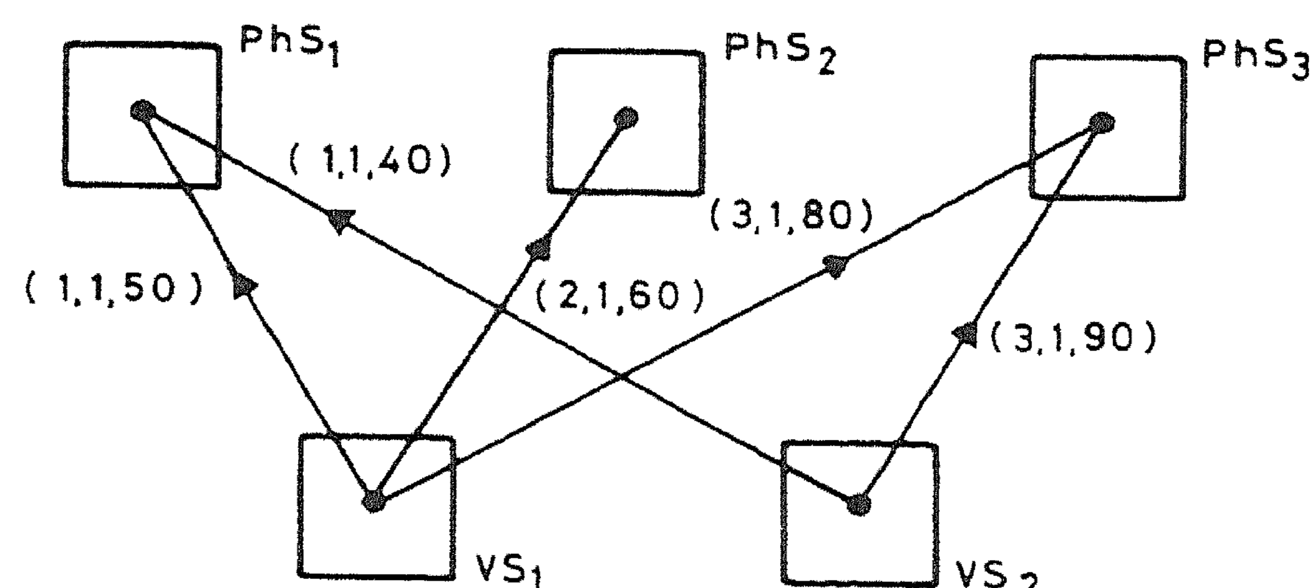


Figure 4.23. Processing-schedules graph.

assign	increase sum
VS_2 to PhS_1	100
VS_2 to PhS_2	50 + 100 = 150
VS_2 to PhS_3	50

The assignment of VS_2 to PhS_3 increases the sum the least and, therefore, VS_2 is united with PhS_3. It remains to assign VS_1.

assign	increase sum
VS_1 to PhS_1	70 + 90 = 160
VS_1 to PhS_2	10 + 90 = 100
VS_1 to PhS_3	50 + 70 = 120

Because the assignment of VS_1 to PhS_2 increases the sum the least, it is turned into a union. So, the final allocation is $PhS_2 = (\{F_1\})$ and $PhS_3 = (\{F_2\})$ (operation-sets are omitted), and the sum of the response transmission times is $60 + 90 = 150$.

□

4.8.5. Comparison Heuristic and Optimal *ARTT* Allocations

To get an impression what the quality of the allocations produced by algorithm *simple resp data allocation* a comparison is made between the average response transmission times of these schedules and of the optimal ones. For each site in the computer network one query is generated, which references a number of fragments. The schedule of a query consists of transmissions from the sites where the fragments reside to the result site. If several fragments that are referenced in the same query, their transmissions to the result site of the query are serialized.

To be able to compute optimal allocations that minimize the average response

transmission time, the experiments are confined to a small computer network and a small database. The number of physical sites is four and the number of fragments is six. Because only non-redundant allocations are considered, this implies that there are six virtual sites in the processing-schedules graphs generated. In the experiments only one parameter was varied, namely the average number of fragment referenced in a query (f); it will vary from 1 to 5. The amount of data transmitted from a fragment to the result site of a query in which it participates, is randomly drawn between 0 and 500. The results are shown in table 4.24; each entry is the average of 50 test runs.

f	1	2	3	4	5
opt	49.4	148.1	251.2	348.6	419.3
heur	50.0	153.4	262.1	364.5	442.3
%	1.2	3.6	4.3	4.6	5.5

Table 4.24. *ARTT* for varying the number of fragments.

For small f most of the queries will reference different fragments, which cause a small *ARTT*. Because a virtual site is connected with only a few physical sites, algorithm *simple resp data allocation* will have no difficulty finding near optimal allocations. If f is increased, the *ARTT* increases as well, both because a virtual site can not be united with all physical sites to which it is connected and because the processing schedules have to be serialized. From the table we can see that the heuristic algorithm starts to produce allocations that have a higher *ARTT* than the optimal ones. This is because several fragments accessed in one query will have to be allocated to the same physical site, which may cause that considering one fragment at the time leads to non-optimal solutions.

Overall we may conclude that if queries reference a number of fragments which is less than the number of sites in the computer network, the heuristic algorithm *simple resp data allocation* computes near optimal allocations.

4.8.6. Minimizing Average Response Processing Time.

For simple processing schedules we now consider the other extreme: transmission cost is negligible compared to processing cost. This means that the edges in the processing-schedules graph have no meaning. Because the data allocation is determined based on the response processing time we will speak of the **operation allocation**. However, this does not mean that operations can be freely allocated to any of the physical sites. Only the virtual sites with their complete operation-sets can be assigned to or united with a physical site. Two strategies to determine a data allocation are distinguished, one from the system's point of view and one from the users'.

Let us first consider the system's point of view. The system seems to be better off if the maximum queueing time over the physical sites is minimized. The utilization factor ρ of a physical site is an important parameter in the formula of the queueing time. The idea of minimizing the maximum queueing time, is more or less similar to minimizing the maximum ρ among all physical sites. Assume that the arrival times

of operations with execution time x_i are described by a Poisson distribution with λ_i as mean. Then,

$$\rho = \lambda_1 x_1 + \lambda_2 x_2 + \cdots + \lambda_n x_n,$$

showing that each operation that is offered to a physical site contributes a little piece to its ρ. The problem of minimizing the maximum ρ, denoted by ρ_{max}, means keeping the largest sum of $\lambda_i x_i$'s among the physical sites as small as possible. To get an idea how difficult this problem is we will discuss a similar problem, which is known to be NP-complete.

This problem is known as the **multiprocessor scheduling problem** [Garey1979]. Imagine a set of tasks $T = \{t_i\}$ each having their own execution time $l(t_i) \in \mathbf{Z}^+$, and a number of physical sites m $(m \geqslant 1)$. A physical site can execute only one task at a time, and immediately after a task is completed another task can be started. So, there is no loss of time between tasks. The question is: is there an assignment of the tasks to the different physical sites such that an overall deadline $D \in \mathbf{Z}^+$ is met. Meaning that all physical sites should be finished before D units of time have passed after the execution has started.

Theorem 4.13 The problem of minimizing ρ_{max} is NP-complete.

Proof We will show this by translating the multiprocessor scheduling problem to the problem of minimizing ρ_{max}.

Create for every $t_i \in T$ an operation O_i with service time $l(t_i)/D$. The question whether there is an assignment that meets the overall deadline D, is translated to whether there exists an operation allocation such that $\rho_{max} \leqslant 1$.

Assume we have an operation allocation with $\rho_{max} \leqslant 1$, then the corresponding assignment can be obtained by assigning the tasks t_i to the same physical site as operation O_i is allocated to. Because $\rho_{max} \leqslant 1$ all sites will be finished at a time less than or equal to D.

Also, if there is an assignment which meets D, then the corresponding operation allocation can be determined in a similar way.

□

Another approach to the operation allocation is to look at it from the **users'** point of view. If we assume a completely empty system then the response processing time of one query will be minimum if all the operations are executed in parallel and their results sent to the result site, where the final result is produced. So, maximizing parallelism is a good way to minimize the response processing time. To compute a suitable operation allocation we construct a **not-at-same-site graph**. A node N_i in such a graph corresponds to virtual site VS_i and there is an edge between N_i and N_j iff VS_i and VS_j are both accessed in at least one query.

To establish the complexity of this problem we will look at a well-known graph problem. A graph $G(V,E)$ contains a **clique** if there is a subset V' of V such that every two nodes of V' are connected by an edge of E. The size of V' is the size of the clique. To maximize parallelism in a processing-schedules graph virtual sites accessed in the same query should be united with different physical sites. We can see this immediately from the not-at-same-site graph. None of the virtual sites should be united with a physical site with which one of the other virtual sites in the clique is

united. To unite the virtual sites with the physical sites such that the conditions described by the not-at-same-site graph are not violated, we need at least as many physical sites as the size of the largest clique.

Theorem 4.14 The problem of finding an allocation with maximum parallelism is NP-complete.

Proof Follows directly from the NP-completeness of the clique problem

□

4.8.7. Optimal Allocations to Minimize Average Response Processing Time

The operation allocation with minimum average response processing time ($ARPT$) for simple processing schedules will again be computed by the Heuristic Path Algorithm. The heuristic estimator, called *psa resp operation*, is a straightforward extension of *psa resp data* (see fig. 4.20). Assume we are given a partially specified allocation and the corresponding processing-schedules graph. For simplicity we assume again that all λ_i are equal to 1. To start, the RPT_i^r of the queries are computed solely based on the operations that are allocated to the physical sites. For every physical site we can compute the expected queueing time, and so, for each operation its expected time in the system can be obtained by adding its execution time. Also, if several operations of one query are to be executed at one physical site they have to be serialized. This gives the initial value of $ARPT$.

Now consider a virtual site VS which will contain one or more operations. Assign VS to each of the physical sites PhS_j and compute the increases of the RPT_i^r and the increase of the $ARPT$. For every value of j there is an increase of the RPT_i^r. Update the RPT_i^rs by adding the maximum of their increases over j. The contribution of VS is the minimum increase of the $ARPT$. The estimate-cost of a partially specified allocation is the sum of the contributions of the virtual sites plus the initial value of $ARPT$.

When computing the increase of the different response processing times we may take into account the serialization of the operations in VS and in the physical sites to which it is assigned.

Example 4.11

The same data will be used as in the previous example. So, to start with $RPT_1^r = 0.075$ and $RPT_2^r = 0.013$, and, therefore, the initial value of $ARPT = 0.031$. Consider VS_1 first.

1) Assign VS_1 to PhS_1.
   Then $W_1^r = 0.7 \times 0.023 \ / \ 0.3 = 0.054$.

   The consequences for the RPT_i^r are:
   $RPT_1^r = 0.054 + 0.01 + 0.04 + 0.02 = 0.124$, an increase of 0.049,
   RPT_2^r is unchanged.
   This means an increase of 0.014 for $ARPT$.

2) Assign VS_1 to PhS_2.
   Then $W_2^r = 0.45 \times 0.013 \ / \ 0.55 = 0.011$.

   $RPT_1^r = \max(0.075, 0.011 + 0.02) = 0.075$, a zero increase,
   $RPT_2^r = 0.011 + 0.01 = 0.021$, an increase of 0.008.
   This means an increase of 0.006 for $ARPT$.

Now the RPT'_i are updated by adding their maximum increase

$$RPT'_1 = 0.124 \text{ and } RPT'_2 = 0.021.$$

The contribution of VS_1 is the minimum increase of the $ARPT$, which is 0.006.

Finally, consider VS_2.

1) Assign VS_2 to PhS_1.
   Then $W'_1 = 0.75 \times 0.017 \ / \ 0.25$.
   The consequences for the RPT'_i are:
   $RPT'_1 = 0.051 + 0.01 + 0.04 = 1.01$, a zero increase,
   $RPT'_2 = \max(0.021, 0.051 + 0.01) = 0.061$, an increase of 0.04.
   This means an increase of 0.028 for $ARPT$.
2) Assign VS_2 to PhS_2.
   Then $W'_2 = 0.5 \times 0.01 \ / \ 0.5 = 0.01$.
   The consequences for the RPT'_i are:
   RPT'_1 is unchanged,
   $RPT'_2 = 0.01 + 0.01 + 0.01 = 0.03$, an increase of 0.009.
   This means an increase of 0.006 for $ARPT$.

The minimum increase of the $ARPT$ is 0.006. The estimate-cost is the sum of the increases plus the initial value of $ARPT$: $0.031 + 0.006 + 0.006 = 0.043$.

□

Theorem 4.15 Algorithm *psa resp operation* is an admissible estimator if serialization of operations in simple processing schedules allocated to the virtual sites is not considered.

Proof The proof goes along the same lines as the one of theorem 4.10

□

Corollary 4.16 Algorithm *psa resp operation* is an admissible estimator if the simple processing schedules are serialized.

Corollary 4.17 If the Heuristic Path Algorithm uses *psa resp operation* the completely specified allocations produced for simple processing schedules have minimum average response processing time.

4.8.8. Heuristic Allocations to Minimize Average Response Processing Time

Because of the NP-completeness of the problem to maximize parallelism, we will again give a heuristic algorithm to unite the virtual sites with physical sites for simple processing schedules. Algorithm *max parallelism* is shown in fig. 4.25.

During each iteration algorithm *max parallelism* selects a new virtual site, say *VS*, which is accessed in the most number of queries among the virtual sites that have not been selected yet. Then it tries to assign *VS* to a physical sire, say *PhS*, such that no other virtual site in the assigned set of *PhS* is accessed by a query in which *VS* is referenced. If this is impossible *VS* is assigned to an arbitrary physical site.

The main drawback of algorithm *max parallelism* is that it does not take into account the execution times of the operations and the utilization factors of the different physical sites. To overcome this, the algorithm will be extended by computing queueing times, and by serializing operations that are executed at the same

```
proc max parallelism=(graph G)allocation:
begin
        {all nodes are unmarked at the beginning}
  while there is an unmarked node
  do
    take an unmarked node with the most adjacent edges and mark it;
    assign the corresponding virtual site with a physical site to which
    none of the virtual sites corresponding to the adjacent marked
    nodes are assigned;
    if this is impossible assign it to an arbitrary physical site
  od;
  unite the virtual sites with their physical sites
end
```

Figure 4.25. Algorithm *max parallelism.*

physical site and that are part of the same query. This extended version will be called *simple resp operation allocation*.

To start with, *simple resp operation allocation* computes the utilization factor of all virtual sites. At each iteration, a virtual site is united with a physical site. Take the virtual site VS with the largest ρ and assign it successively to each of the physical sites and compute the corresponding increase of the $ARPT$. Unite VS with that physical site that corresponds to the least increase of the $ARPT$. Then continue with the next virtual site.

In the next example we will show how algorithm *simple resp operation allocation* works.

Example 4.12

For simplicity we assume that the execution times are all drawn from the same distribution; this implies that $C_b^2 = 1$. Furthermore, we will not take bulk arrivals into account. This gives a simpler formula for the expected queueing time, namely

$$W = \frac{\rho\bar{x}}{(1-\rho)}.$$

Assume we are given

$PhS_1 = (\{(1,10,0.01),(1,10,0.04)\}),$
$PhS_2 = (\{(2,25,0.01)\}),$
$VS_1 = (\{(1,10,0.02)\}),$
$VS_2 = (\{(2,25,0.01)\}).$

The fragment-sets are omitted. There are two queries with frequencies 10 and 25.

Two operations have already been allocated to PhS_1 and one to PhS_2. Given this partially specified allocation the queueing times of the two physical sites are:

$W_1 = 0.5 \times 0.025 \ / \ 0.5 = 0.025$, and
$W_2 = 0.25 \times 0.01 \ / \ 0.75 = 0.003$,

and the response processing times of the queries are:

$RPT_1 = 0.025 + 0.01 + 0.04 = 0.075$,
$RPT_2 = 0.003 + 0.01 = 0.013$.

The ρ of VS_1 is 0.2 and of VS_2 is 0.25, so, VS_2 is considered first.

1) Assign VS_2 to PhS_1.
Then, $W_1 = 0.75 \times 0.167 \ / \ 0.25 = 0.5$.
The consequences for the RPT_i are:
$RPT_1 = 0.5 + 0.01 + 0.04 = 0.55$, an increase of 0.475
$RPT_2 = \max(0.013, 0.5 + 0.01) = 0.51$, an increase of 0.497

2) Assign VS_2 to PhS_2.
Then, $W_2 = 0.5 \times 0.01 \ / \ 0.5 = 0.01$.
The consequences for the RPT_i are:
RPT_1 is unchanged, and
$RPT_2 = 0.01 + 0.01 + 0.01 = 0.03$, an increase of 0.017.

The increase of the latter assignment is clearly the least and, therefore, VS_2 is united with PhS_2. So, $RPT_1 = 0.075$ and $RPT_2 = 0.03$, and, therefore, $ARPT$ is 0.043.

Now consider VS_1.

1) Assign VS_1 to PhS_1.
Then, $W_1 = 0.7 \times 0.023 \ / \ 0.3 = 0.054$.
The consequences for the RPT_i are:
$RPT_1 = 0.054 + 0.01 + 0.04 + 0.02 = 0.124$, an increase of 0.049,
RPT_2 is unchanged.

2) Assign VS_1 to PhS_2.
Then, $W_2 = 0.7 \times 0.012 \ / \ 0.3 = 0.028$
The consequences for the RPT_i are:
$RPT_1 = \max(0.075, 0.028 + 0.02) = 0.075$, a zero increase,
$RPT_2 = 0.028 + 0.01 + 0.01 = 0.048$, an increase of 0.018.

If VS_1 is assigned to PhS_1 then its contribution is 0.014 and if assigned to PhS_2 to 0.013. Therefore, VS_1 is united with PhS_2. The resulting $ARPT = 0.056$.

□

4.8.9. Comparison Heuristic and Optimal *ARTT* Allocations

To get an idea how the allocations produced by *simple resp operation allocation* compare with the optimal allocations for minimizing the average response processing time, we did some experiments. The database consists of seven fragments, which are assigned to different virtual sites. Each query consists of operations that are executed at the site where the fragments referenced, are located, and of an operation at the result site. For simplicity we assume that all queries are stated with the same

frequency. The number of physical sites four. The goal is to unite virtual sites with physical sites such that the average response processing time is minimized. The queueing delays are computed by means of the formula for W_g discussed in subsection 4.8.1. If several operations within a query are executed at one site they are serialized.

Two parameters are varied:

- ☐ the average number of fragments referenced per query (f),
- ☐ the average number of operations serviced per unit of time (μ).

The f varies from 1 to 7, and the operation size is drawn from a negative exponential distribution. The results are shown in table 4.26; each entry is the average of 50 test runs.

f	1	3	5	7
opt	0.27	0.48	0.92	2.38
heur	0.28	0.49	0.97	2.72
%	3.7	2.1	5.4	14.3

$\mu = 5$

(a)

μ	5	10	50
opt	2.18	0.48	0.16
heur	2.27	0.49	0.16
%	4.1	2.1	0.0

$f = 3$

(b)

Table 4.26. *ARTT* for varying f and μ.

From the table we may conclude that the heuristic algorithm *simple resp operation algorithm* performs well if the execution times of the operations times the frequencies with which they are executed are small and if the number of fragments referenced per query is less than the number of sites in the computer network. If the utilization factors of the physical sites are getting close 1 the heuristic algorithm *simple resp operation allocation* starts to produce allocations that have a much higher *ARPT* than the optimal allocations. Also, the percentage of cases in which the heuristic algorithm can not find an allocation with all utilization factors less than 1, while there exists one, starts to increase. For $f = 7$ and $\mu = 5$, this was 6%.

Overall we may conclude that the heuristic algorithm performs best if the execution times of the operations are small and the number of fragments referenced compared to the size of the computer network is small.

4.8.10. Minimizing Average Response Time

Finally, we consider the most realistic model. The response time of a query is determined by both its transmissions and processing of operations.

A heuristic algorithm can be constructed from either *simple resp data allocation* or *simple resp operation allocation*. For example, if transmission cost weighs more than processing cost, algorithm *simple resp data allocation* should be used, however, to compute the effects on the response times of queries the operations should be taken into account as well. If, on the other hand, processing cost weighs more, one should use algorithm *simple resp operation allocation*; however, to compute the effects on the response times of the queries the transmissions should be taken into account.

A heuristic estimator for the Heuristic Path Algorithm for this more general model can be obtained by combining the estimators *psa resp data* and *psa resp operation*. Given a partially specified allocation, the RT'_i of the queries are computed based on the transmissions between physical sites and the operations that are allocated to the physical sites. The contribution of a virtual site VS is determined by considering all possible assignments of this virtual site to physical sites. For every assignment we can compute the increases of the RT'_i of the queries. There is one assignment such that the contribution of VS to ART is minimum. This increase is taken as the contribution of VS. The RT'_i are updated by adding the maximum increase of the different assignments of VS. By adding up the contributions of the virtual sites we will obtain the estimate-cost of a partially specified allocation.

4.8.11. Arbitrary Processing Schedules

In subsections 4.8.2-4.8.10 simple processing schedules were investigated. These schedules imply that the allocation of the operations is completely determined by the data allocation. Here, we will allow for both a **data and operation allocation**. To do so, in the processing-schedules graph belonging to the initial allocation, virtual sites are created for both operations and fragments. Only the operations that implicitly read data in the fragments from secondary storage are put in the operation-set of the virtual site to which the fragment is allocated. An example of such an operation is the restriction. For other operations such as a join a new virtual site is created. If a virtual site has an empty fragment-set it is called an **operation virtual site**, otherwise it is called a **fragment virtual site**. The final operation in the processing schedule of a query is permanently allocated to the physical site that corresponds to the result site in the computer network. An example of a final operation is one that presents the result of the query to the user.

For these arbitrary processing-schedules graphs we will confine ourselves to minimizing response transmission time. As far as minimizing response processing time is concerned we expect that similar techniques as proposed in subsections 4.8.7 and 4.8.8 can be used.

4.8.12. Optimal Operation and Data Allocations to Minimize $ARTT$

The next step is to compute both an operation *and* data allocation. The optimal **data and operation allocation** will be obtained by using the heuristic estimator for simple processing schedules, *psa resp data*. Assume we are given a partially

specified allocation and the corresponding processing-schedules graph. To apply the same techniques as was done for the simple processing schedules, a new processing-schedules graph is constructed.

From each nucleus-site, NS_i, there are sequences of transmissions part of the schedule of one query leading to the result physical site of that query; they will be called **paths**. On each path there is a **nearest physical site**, which is the first physical site encountered when going through the sequence of transmissions and considering the nucleus-sites in the processing-schedules graph that receive the transmissions. The nearest physical site is not necessarily the result physical site of the query.

For example, during the search process a partially specified allocation can have been obtained where some of the virtual sites in the initial allocation may have been united with physical sites.

For each query and each nucleus-site accessed in the query, consider the paths from the nucleus-site to the nearest physical site on that path. Each edge on these paths represents the transmission of an amount of data. For each nearest physical site place an edge in the new processing-schedules graph between the nucleus-site and that physical site, directed to the latter with the same label as the edge on the path to the nearest physical site which transmitted the least amount.

The estimate-cost based on this newly constructed processing-schedules graph is done as follows. The initial values of the RTT'_i are set equal to the RTT_is, which are solely based on transmissions between physical sites. Take an arbitrary virtual site, VS, and assign it in turn to each of the physical sites with which it is connected, and compute the increase of the $ARTT$. The minimum increase among the different assignments is taken as the contribution of VS. Then the RTT'_i are updated with their maximum increase among the different assignments. Before selecting the next virtual site, VS and its adjacent edges are removed from the processing-schedules graph. This continues until no virtual sites are left over.

Theorem 4.18 The heuristic estimator *psa resp data* is admissible if the schedules are not serialized.

First, we will prove that the estimate-cost of a given partially specified allocation based on the newly constructed processing-schedules graph, called N, is less than or equal to the cost of any completely specified allocation that satisfies the given partially specified allocation.

Assume we are given a completely specified allocation that satisfies the given partially specified allocation. The corresponding processing-schedules graph, called P, can be obtained by uniting the virtual sites with physical sites. Let us assume that VS_i is united with PhS_j. Then at least one edge on each path to the other physical sites in the original processing-schedules graph, called O, is present in P. If the same union would have been done in N, VS_i would be connected with exactly one edge on each path to other physical sites in O. Because the edges are chosen such that they represent the smallest transmission, the cost of a completely specified allocation based on N is less than or equal to the cost on O.

The rest of the proof, showing that *psa resp data* underestimates the cost of completely specified allocation based on N, goes along the same lines as the proof of theorem 4.10

□

Corollary 4.19 The heuristic estimator *psa resp data* is admissible if the schedules are serialized.

Corollary 4.20 The Heuristic Path Algorithm using *psa resp data* as a heuristic estimator computes completely specified allocations with minimum average response transmission time.

Example 4.13

Assume we are given the processing-schedules graph shown in fig. 4.27(a), and assume that $TT(X) = X$.

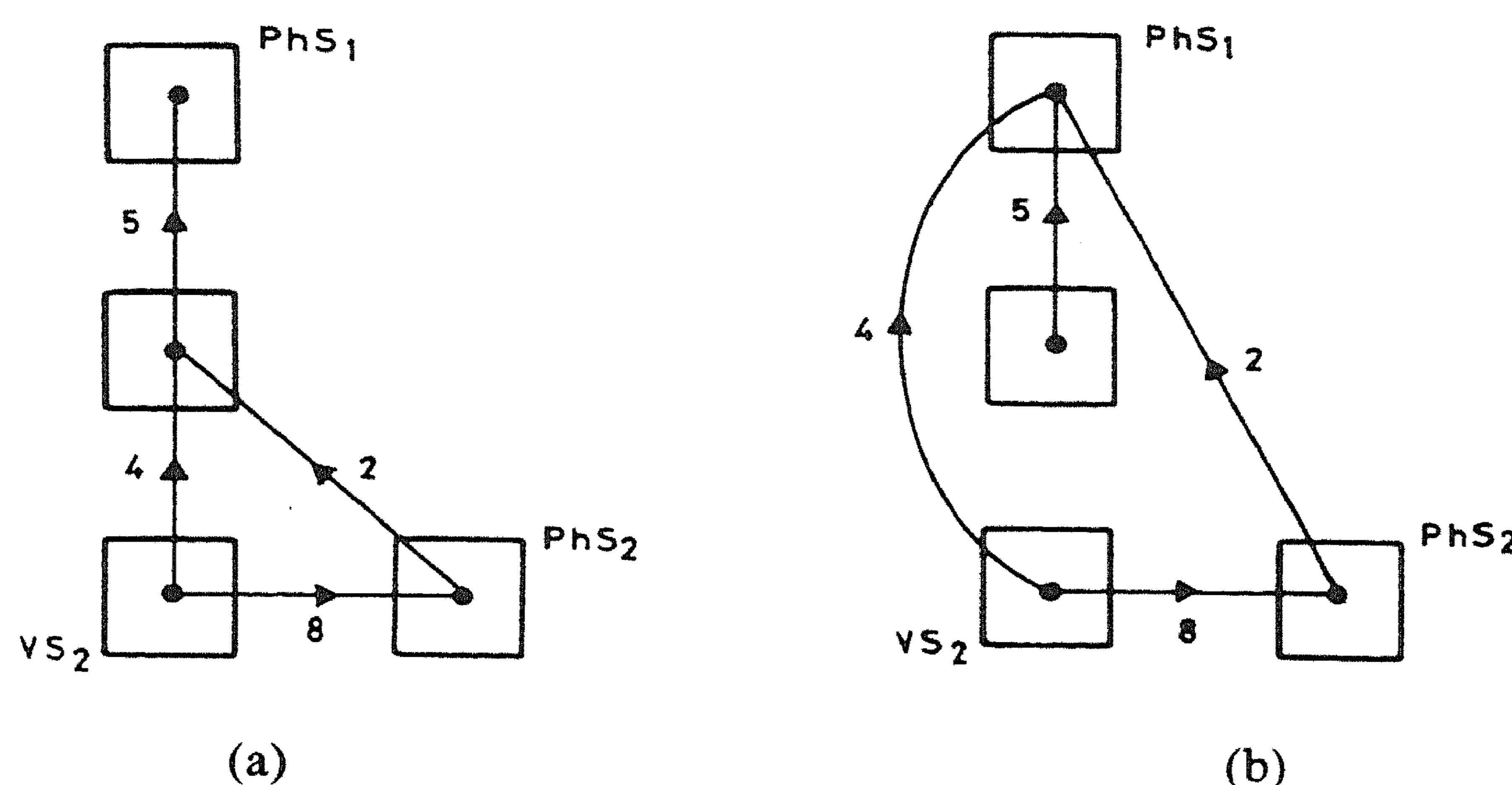


Figure 4.27. Processing-schedules graph of example 4.13.

For each nucleus-site on all paths to the result physical site there is a nearest physical site. For example, for path VS_2, VS_1, PhS_1, the nearest physical site is PhS_1, and for VS_2, PhS_2, VS_1, PhS_1, it is PhS_2. For each path the edge that represents the smallest transmission is used to connect the nucleus-site with the nearest physical site on that path in the newly constructed processing-schedules graph. The result is shown in fig. 4.27(b). Initially $RTT'_1 = 2$. Now consider VS_2. If it is assigned to PhS_2 the RTT'_1 is increased by 2, and if assigned to PhS_1 by 8. Hence, the contribution of VS_2 equals 2, and RTT'_1 is set to 10. The contribution of VS_1 is zero. Hence, the estimate-cost of the partially specified allocation, which is the sum of the contributions plus the initial value of RTT'_1, is $2 + 2 = 4$.

□

4.8.13. Operation Allocation Given a Data Allocation

We will now discuss an algorithm that computes an operation allocation that minimizes the average response transmission time given the allocation of the fragments. The operation allocation can be computed separately for every query because each operation belongs to only one query. Before we present the algorithm we will introduce some notions.

If a nucleus-site contains an operation whose result is transmitted to another nucleus-site then we will call this operation an **export-operation.** A schedule is called

tree-structured if every nucleus-site contains at most one export-operation, which transmits its result to exactly one other nucleus-site (this means no forking points). If nucleus-site NS_i contains an export-operation whose result is transmitted to NS_j then we call NS_i an **input nucleus-site** of NS_j. If all the input nucleus-sites of nucleus-site NS are either physical sites or virtual sites that are assigned to physical sites, and if NS is a virtual site or if NS is the result physical site then NS is called a **candidate nucleus-site.** The amount of data transmitted from an export-operation EO of a nucleus-site is denoted by $|R(EO)|$.

If a query has a tree-structured schedule the export operation of a nucleus-site is uniquely determined by the nucleus-site itself. Therefore, we will use $R(NS_k)$ when we mean the result of the export operation of NS_k.

The response transmission time of a nucleus-site NS, denoted by $RTT(NS)$, is defined if all the nucleus-sites from which NS receives data directly or indirectly are either physical sites or virtual sites assigned to physical sites. It can be computed by reconstructing the schedule of the transaction up to and including the operations allocated to NS.

Note that the RTT of a nucleus-site NS depends on the allocation of NS if it is a virtual site and on the allocation of the nucleus-sites from which it receives data directly or indirectly. We define the **response transmission time of NS at physical site PhS_j**, denoted by $RTT_j(NS)$, as $RTT(NS) + TT_{ij}(|R(NS)|)$, where NS is either PhS_i or is assigned to it, and TT_{ij} stands for the transmission time from PhS_i to PhS_j.

Assume we are given a completely specified allocation of the fragments, and the processing-schedules graph that belongs to this allocation where each operation is allocated to its own virtual site. Algorithm *resp operation allocation* will compute a completely specified operation allocation given the data allocation such that the response transmission time of a transaction is minimized. At each iteration the algorithm takes a candidate nucleus-site, say CNS. The input nucleus-sites of CNS that are virtual sites assigned to physical sites are united with CNS; call the resulting nucleus-site CNS_u. If CNS_u is a physical site then we have computed a completely specified operation allocation, because CNS_u is the result physical site. Otherwise, CNS_u is a virtual site that will be either assigned to or united with one of its input nucleus-sites. Note, that all inputs of CNS_u are physical sites. To determine whether to assign it to or unite it with one of its input nucleus-sites we do the following. Assign CNS_u to an arbitrary physical site PhS_x that is not one of its inputs. Compute the response transmission time of CNS_u given this allocation, say its value is V. Let $|R(EO_k)| \geq \max_i |R(EO_i)|$, where EO_k is an export operation of PhS_z. Now assign CNS_u to PhS_z to compute $RTT_x(CNS_u)$; if $RTT_x(CNS_u) \leq V$ then CNS_u is united with PhS_z, otherwise it will stay assigned to it.

The procedural form of algorithm *resp operation allocation* is given in fig. 4.28.

Before considering the quality of the completely specified operation allocation produced we give an example.

Example 4.14

Assume we are given the processing-schedules graph shown in fig. 4.29(a). The transmission time is $TT(X) = X$. The fragments accessed are already allocated to different physical sites and for each of the two joins a virtual site was created.

```
proc resp operation allocation=(schedules graph psg)allocation:
begin
   while there is a candidate nucleus-site
   do
      let NS be such a nucleus-site;
      unite its input virtual sites with it;
      if NS is a virtual site
         then
            determine whether to assign it to or unite it with one of its
            input physical sites
      fi
   od;
   extract operation allocation from psg
end
```

Figure 4.28. Algorithm *resp operation allocation*.

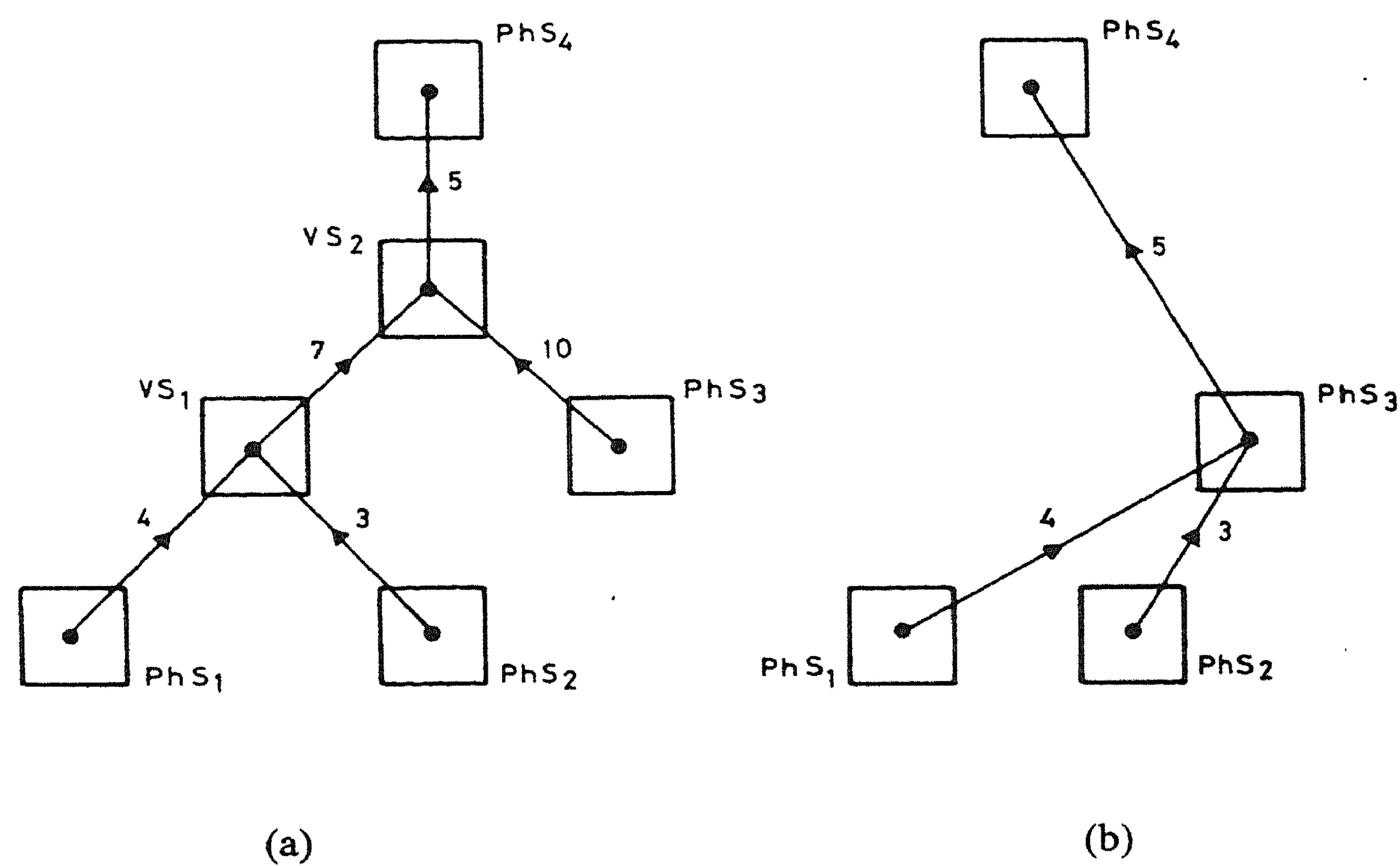


Figure 4.29. Processing-schedules graphs of example 4.14.

Virtual site VS_1 is a candidate nucleus-site because both its inputs nucleus-sites are physical sites. Because none of its inputs is a virtual site we continue with the assignment to or union with either PhS_1 or PhS_2. Because $|R(PhS_1)| > |R(PhS_2)|$, it will be PhS_1. Assigning VS_1 to an arbitrary physical site PhS_x ($x \neq 1, 2$) results in $RTT(VS_1) = 4$. On the other hand assigning VS_1 to PhS_1 will give $RTT_x(VS_1) = 10$. Therefore, VS_1 is not united with PhS_1 but assigned to it.

Now VS_2 is the candidate nucleus-site. One of the inputs of VS_2, namely VS_1 is a virtual site and, therefore, they will be united with each other; the result is denoted by VS_u. VS_u has three input nucleus-sites: PhS_1, PhS_2 and PhS_3. For the latter holds that $|R(PhS_3)|$ is maximum. Assigning VS_u to an arbitrary physical site PhS_x ($x \neq 1, 2, 3$) gives $RTT_x(VS_u) = 4 + 5 = 9$. Because the latter is smaller VS_u is united with PhS_3.

The resulting processing-schedules graph is shown in fig. 4.29; the response transmission time is 9 which is optimal.

□

Now we will discuss the quality of the completely specified operation allocation obtained.

Theorem 4.21 Algorithm *resp operation allocation* computes completely specified operation allocations with minimum response transmission time for tree-structured schedules.

Proof We will prove the following property by induction on the number of candidate nucleus-sites considered.

Property M:

M_1) a physical site has at most one virtual site assigned to it,

M_2) $RTT(PhS_i) + TT_{ix}(|R(PhS_i)|)$ is minimum, over all allocations,

M_3) the minimum response transmission time of VS at physical site PhS_x over all allocations, where VS is already assigned to PhS_y can be obtained by re-assigning VS to PhS_x (x is not necessarily different from y).

If no candidate nucleus-sites have been considered M_1 and M_2 are true and M_3 is not applicable.

Assume that M is true after n candidate nucleus-site have been handled. Now consider the $n + 1$-st candidate nucleus-site, CNS. Let CNS_u be the result after the union. If CNS_u is *assigned to* a physical site, M_2 still holds simply because of the induction hypothesis; remember that nothing has changed with the physical sites. Also, because CNS_u is assigned to one of its input physical sites M_1 holds. Now we have to prove that the minimum $RTT_x(CNS_u)$ for arbitrary x is obtained by assigning CNS_u to PhS_x. Uniting CNS with its input virtual sites guarantees that the inputs of CNS_u have minimum response transmission time. The latter is true because the inputs are physical sites for which M_2 holds. CNS_u is merely assigned to a physical site because $RTT_x(CNS_u)$ is smaller if CNS_u is assigned to PhS_x. But because CNS_u's inputs have minimum response transmission time $RTT_x(CNS_u)$ is minimum if CNS_u is assigned to PhS_x.

If CNS_u is *united with* one of its inputs we only have to prove M_2. M_1 and M_3 are not applicable because there are no newly assigned virtual sites. Again CNS_u's inputs have minimum response transmission time over all allocations. The reason that CNS_u is united with one of its inputs is that the $RTT_x(CNS_u)$ for arbitrary x is smaller if CNS_u is united with one of its inputs. Hence, M_2 holds.

□

In this subsection we have considered the computation of an operation allocation, given a data allocation, such that the response transmission time is minimized. For tree-structured processing schedules algorithm *resp operation allocation* computes optimal operation allocations. This algorithms will be used in the next subsection where the allocation of both the data and the operations will be discussed.

4.8.14. Heuristic Operation and Data Allocation to Minimize $ARTT$

A heuristic algorithm called *resp data operation allocation* will be described now. The initial assignment is determined as follows. If in the processing-schedules graph corresponding to the initial allocation there is a transmission from a fragment virtual site to a physical site this virtual site is assigned to the physical site to which it transmits most of the data. Other virtual sites are assigned arbitrarily.

Given the assignment of the fragment virtual sites to the physical sites, we would like to be able to determine the operation allocation with algorithm *resp operation allocation*. The problem is that this algorithm requires a completely specified data allocation to do so. However, this can be solved quite easily by letting algorithm *resp operation allocation* consider a fragment virtual site as a physical site.

Now for every query a set of proposals for changing the data allocation and the corresponding decreases in the response transmission time of the query is computed. These proposals are based on the current data and operation allocation. In the basic version of algorithm *resp data operation allocation*, we only allow for simple changes in the data allocation. Such a simple change is defined as follows. In a schedule for a query there will be a sequence of transmissions that determines the response transmission time. To decrease this, one of the transmissions will have to disappear. If it is a transmission between two fragment virtual sites, these two can be united and assigned to either physical site to which they were originally assigned. If it is a transmission from a fragment virtual site to a physical site, the former can be assigned to the latter.

After all proposed changes in the allocation and their expected decrease in the response transmission times of the transactions are collected, the expected decrease in the average response transmission time is computed for each change. We go through the list of proposed changes in descending order of the expected decrease of the $ARTT$ until a change is found that actually does decrease the $ARTT$. This actual change is computed as follows. Take the processing-schedules graph belonging to the initial allocation and assign the fragment virtual sites just like the initial assignment only with the proposed changes incorporated. So, again we have an assignment of fragment virtual sites and algorithm *resp operation allocation* is applied to compute the RTT_i of the queries. Because some of the proposals consist of two alternatives of assigning the fragment virtual site obtained from uniting two virtual sites, they are both considered separately. The one with the smallest $ARTT$ is taken. If this $ARTT$ is less than the previous $ARTT$ we go through the next iteration. If not, the next

proposed change on the list is taken until one is found that decreases the $ARTT$. If no such proposed change can be found the algorithm terminates.

We will give an example to show how the algorithm works.

Example 4.15

Assume a database consists of three fragments F_1, F_2 and F_3 which are allocated to FVS_1, FVS_2 and FVS_3, respectively. There are two queries, one computes the join between F_1 and F_2, and the other the join between all three fragments. The processing-schedules graph corresponding to the initial allocation is shown in fig. 4.30.

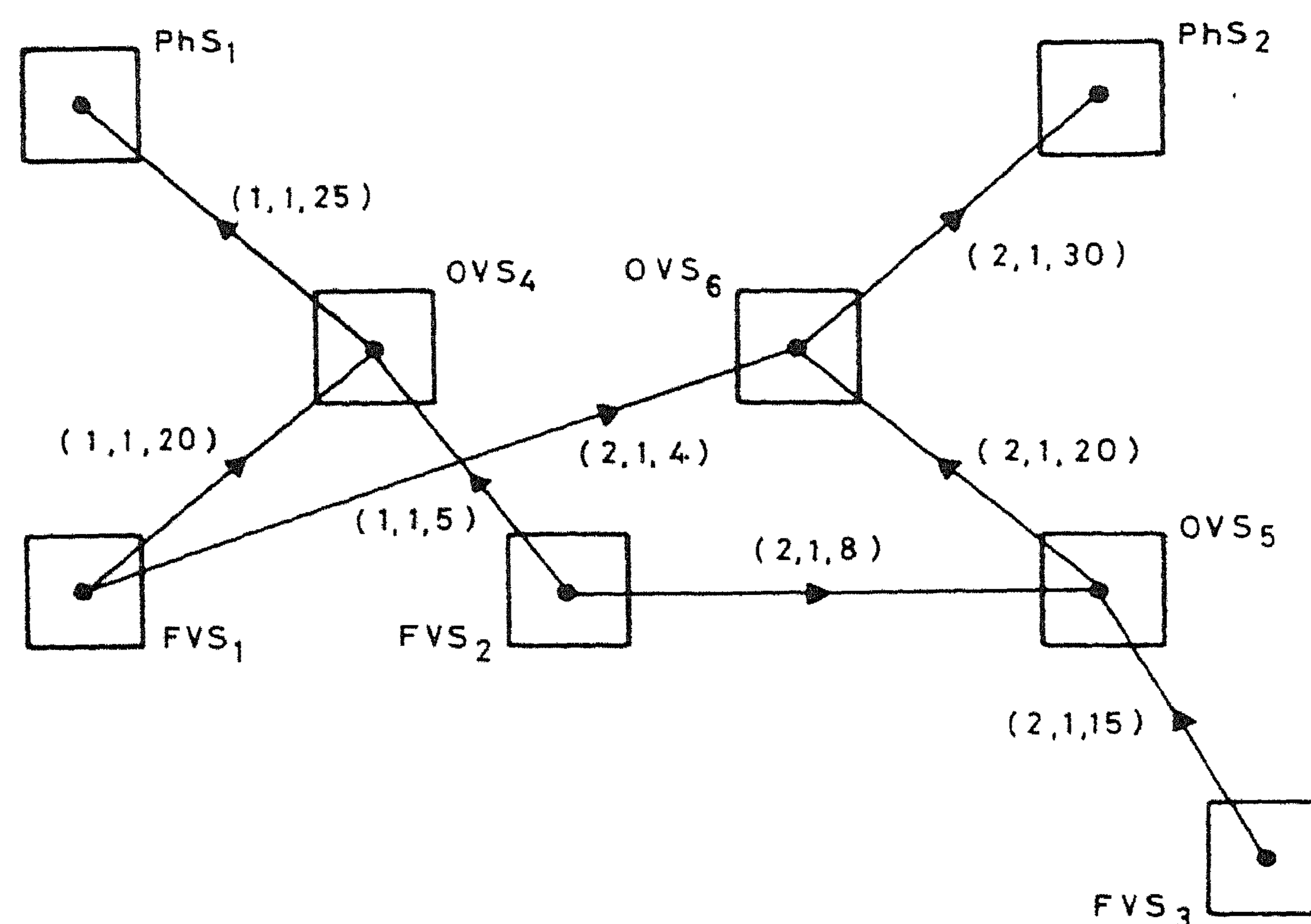


Figure 4.30. Processing-schedules graph of example 4.15.

Our goal is to unite both the virtual sites containing fragments as well as the ones containing operations with the physical sites. Algorithm *resp operation allocation* starts with an initial assignment. Because none of the fragment virtual sites transmits data directly to a physical site they all are assigned arbitrarily:

assign FVS_1 to PhS_2
assign FVS_2 to PhS_1
assign FVS_3 to PhS_1

Now algorithm *resp operation allocation* is applied for both queries to compute their RTTs. $RTT(Q_1) = 20$, which is obtained by uniting OVS_4 with PhS_1, and $RTT(Q_2) = 15$, by uniting OVS_5 and OVS_6 with PhS_2. The initial assignment is shown in fig. 4.31.

For both queries changes in the data allocation will be proposed to decrease their RTTs. The RTT of Q_1 is determined by the transmission from FVS_1 to PhS_1 and can be decreased by assigning FVS_1 to PhS_1. Similarly for Q_2 by assigning

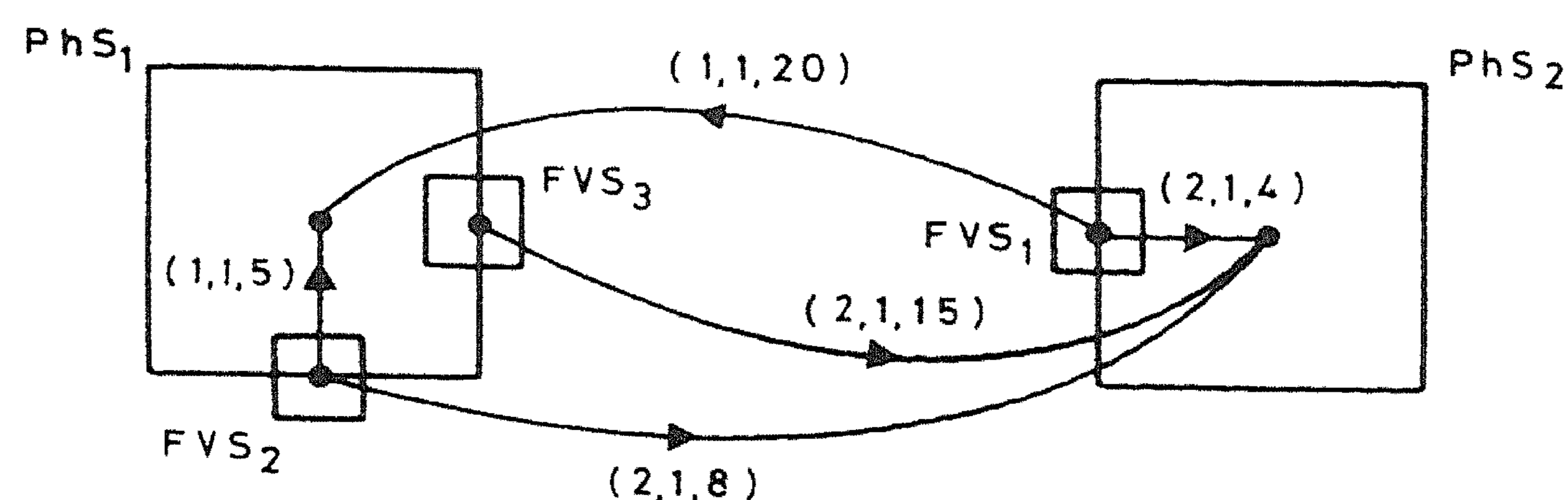


Figure 4.31. Initial assignment.

FVS_3 to PhS_2. The proposals are considered in order of decreasing potential effect on the $ARTT$ until a real decrease of the $ARTT$ is obtained. Therefore, the proposal of Q_1,assign FVS_1 to PhS_1, is considered first.

Because both FVS_1 and FVS_2 are assigned to PhS_1 algorithm *resp operation allocation* will unite OVS_4 with PhS_1, resulting in $RTT(Q_1) = 0$. The operation allocation of Q_2 stays the same and, therefore, there is no change in $RTT(Q_2)$. The $ARTT$ decreases from 17.5 to 7.5. The corresponding change in the assignment of the fragment virtual sites is adopted.

In the next iteration there will only be a proposal for Q_2, because the $RTT(Q_1) = 0$. The proposal is: assign FVS_3 to PhS_2. The consequences for the RTT of Q_2 are again computed by algorithm *resp operation allocation*. Again OVS_5 and OVS_6 are united with PhS_2, giving a $RTT(T_2) = 8$. Hence, the $ARTT$ drops from 7.5 to 4, and, therefore, the change is adopted. The result is shown in fig. 4.32.

In the next iteration there will again only be a proposal for Q_2: assign FVS_2 to PhS_2. The consequences of this change are that $RTT(Q_1) = 5$ and $RTT(Q_2) = 4$, which means an increase of the $ARTT$ from 4 to 4.5, and, therefore, the proposal is rejected. Because there are no more proposals the algorithm terminates.

The final data and operation allocation is obtained by uniting FVS_1, FVS_2 and OVS_4 with PhS_1 and FVS_3, OVS_5 and OVS_6 with PhS_2 giving an $ARTT$ of $(0 + 8) / 2 = 4$.

□

For the simple processing schedules it was already time consuming to compute optimal allocations for small databases and small computer networks. Because the differences between simple and arbitrary processing schedules can only be shown for large schedules, a comparison between the allocation produced by the heuristic algorithm *resp data operation allocation* and the optimal ones is omitted.

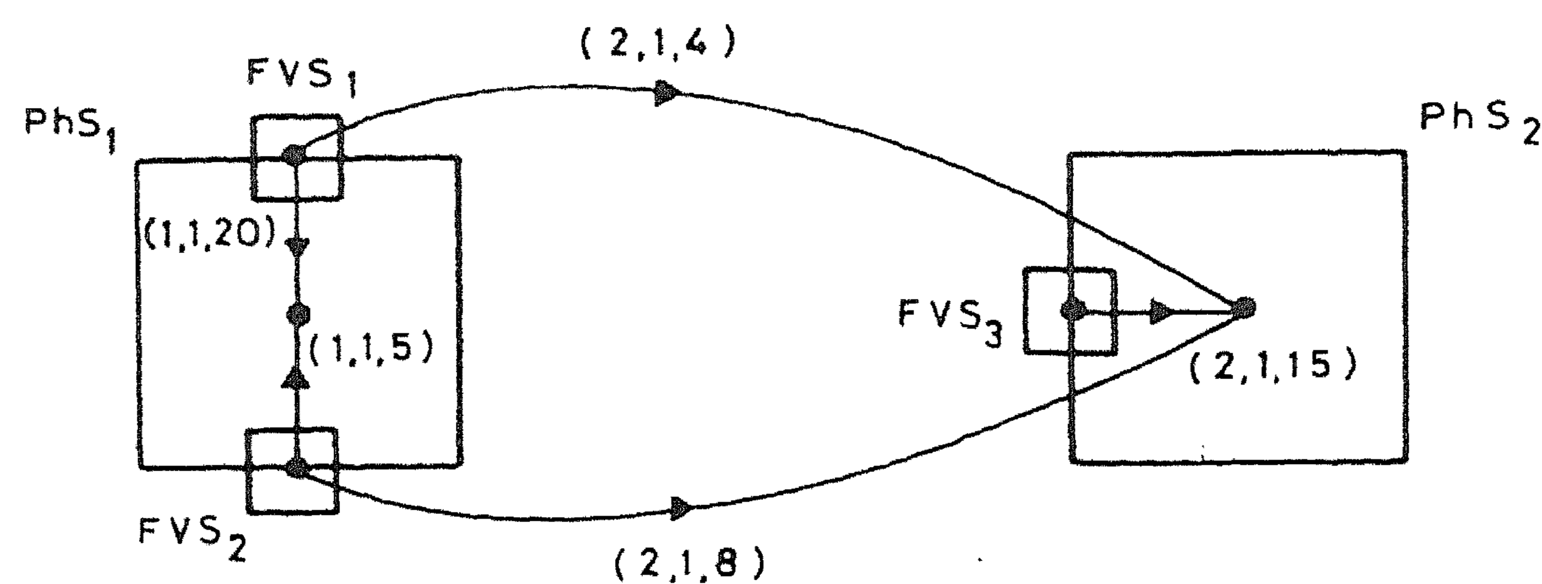


Figure 4.32. Assignment of FVS_3 to PhS_2.

4.8.15. Summary

Most research on the file allocation problem deals with minimizing total transmission cost. Sometimes, response time constraints are added. In this section the problem of determining allocations such that the average response time of the queries is minimized, was investigated.

In the first half of the section, only simple processing schedules were considered. Such schedules are characterized by the fact that operations are executed at either the result site or at the sites where the fragments referenced reside and the fact that only transmissions to the result site are allowed. These simple processing schedules are exactly the schedules used in the file allocation problem. For these simple schedules both minimizing response transmission time and response processing time were treated. It was shown that minimizing average response time is NP-complete. To be able to compute optimal allocations that minimize average response transmission time a heuristic estimator, called *psa resp data*, was given. For the same problem a heuristic algorithm, called *simple resp data allocation*, was developed and the solutions obtained by it were compared with the optimal ones. The allocations produced by *simple resp data allocation* have near optimal average response transmission time if the number of fragments referenced per query does not exceed the number of sites in the computer network.

For minimizing the average response processing time both the system's point of view and the users' were considered. For both it was shown that minimizing $ARPT$ is NP-complete. To compute the processing time of a query the queueing delays of the physical sites have to be known. Lacking a queueing model for the whole network, which can handle processing schedules with forking and synchronization points, we assume that Jackson's Independence Theorem can be applied, i.e., that for each physical site the queueing delays can be computed based on the operations allocated to it. Based on the techniques used in the heuristic estimator *psa resp data* an estimator *psa resp operation* was developed. Also, a heuristic algorithm, called *simple resp operation allocation*, which emphasizes the maximization of parallelism, was proposed, and the solutions were compared with the optimal ones. Again, the average response

processing times of the allocations produced by the heuristic algorithm were close to the optimal solutions, if the number of fragments referenced per query does not exceed the number of sites in the computer network and the execution times of the operations times the frequencies with which they are stated, is small.

Minimization of the response time can be obtained by combining the algorithms or heuristic estimators used for minimizing response transmission and processing time.

In the second half of the section arbitrary processing schedules were discussed. These arbitrary schedules imply that both the data and the operations have to be allocated. To start with, an algorithm, called *resp operation allocation*, was presented, which, based on a completely specified data allocation, computes an operation allocation. Because operations are not shared among queries an operation allocation can be determined for each query separately. It was shown that for tree-structured processing schedules these operation allocations have minimum response transmission time. This algorithm is used in *resp data operation allocation*, which determines both an allocation for the fragments of a database and the operations that work on them. For the optimal allocations the heuristic estimator *psa resp data* of the simple processing schedules was used on a transformed processing-schedules graph.

5. SUMMARY AND CONCLUSIONS

Our goal was to study the dual problems query processing and data allocation in distributed database systems. Together they will, to a great extent, determine the efficiency of a database. Query processing, on the one hand, deals with the problem of determining efficient processing schedules given an allocation, and data allocation, on the other hand, with the problem of determining efficient allocations given the queries and updates and a query processing algorithm.

In section 5.1 the results on query processing will be discussed and section 5.2 the results on data allocation. In section 5.3 we will end with some remarks about future research.

5.1. Query Processing

In section 3.1 the three phases in query processing were discussed: parsing, determining a schedule and executing it. Cost functions were defined to measure the efficiency of schedules. Especially, the problem of computing the response time of a schedule was treated in detail. Both the forking and synchronization points in a schedule cause dependencies among transmissions and operations in one processing schedule. A partial solution to this problem was proposed: parallel schedules that end at the same synchronization point and which can not be treated independently because they share the same resource, have to be serialized. This serialization both determines the "real" response time and the order in which transmissions and operations that compete for the same resource have to be served. Because it is not likely that a query processing algorithm will directly deal with the serialization itself, it was proposed to determine a schedule in two phases. In the first phase, it is assumed that no resources are shared by parallel schedules, and in the second phase the schedules obtained, which may contain certain parallel schedules that share resources, are serialized.

Another aspect of query processing is the type of basic operations. Two extremes were discussed. In the one the basic operations manipulated storage structures and in the other they were subqueries in the relational data model. The advantage of the first one is that efficient schedules can be computed and of the second one that site autonomy can be maintained.

In section 3.2 an overview was given of current research on query processing in distributed database systems. A qualitative comparison was made based on three aspects: the choice of materialization, whether schedules are determined before or during execution and whether the joins are computed at other sites than the result site. Most of the research deals with minimizing total transmission cost. Furthermore, it was striking to note the variety of models. Our conclusion was that minimizing response time and the use of low level basic operations had hardly been researched.

In section 3.3 the problem of minimizing the response time of a query for the inverted file organization, was treated. The basic operations were the set operations and the transmission of data. It was proposed to minimize response time at two levels. First, only data transmissions are considered, which gives a macro-schedule. This macro-schedule fixes the duties of the sites involved. Secondly, for each of these sites a micro-schedule must be computed. All these micro-schedules are integrated into the macro-schedule, to obtain a schedule for the query to be processed. Under

the Transmission Assumption, the Parallelism Assumption and the Intersection Assumption and optimal macro-schedule can be determined. The original query is replaced by its disjunctive normal form and the terms are broken into small expressions containing no more than three A-lists and no more than one B-list. All the union operations are executed at the result site. Because the schedule is serialized at different stages and the micro-schedules are integrated the resulting schedule is not necessarily optimal. It was shown that the serializations of macro-schedules by the heuristic serialization algorithm of section 3.1.6 are close to the optimal serializations. The difference between the response time before and after serialization will mainly be determined by the degree in which parallel schedules share the same resources.

For minimizing total time it was shown that forking points can not be removed, as was done for minimizing response time. As a consequence, the terms in the disjunctive normal form can not be treated independently if optimal schedules must be obtained. A heuristic approach was proposed: first schedules for the terms are computed independent of each other and then integrated, by removing superfluous transmissions, to get a schedule for the query to be processed. In general, a schedule for a term consists of the transmission of the smallest A-list along the sites where the other A- and B-lists in the term reside and intersecting at each site the intermediate result obtained so far with the residing lists. The micro-schedules for minimizing total processing time are computed by a variation of Liu's algorithm for minimizing total time in a centralized database.

In section 3.4 only macro-schedules were considered. The basic operations were the semi-join and the transmission of data. A similar approach was taken as for inverted lists. Because the join operation may produce a large result it is processed at the result site. Before the relations are transmitted to the result site they are reduced in size by applying semi-joins on the joining attributes. Under the Transmission Assumption, the Parallelism Assumption and the Selectivity Assumption optimal response transmission time schedules can be obtained. The schedules are very much like the ones for the inverted lists.

For minimizing total transmission cost two approaches were investigated. First, an approach similar to the one for inverted lists was followed. For each relation a schedule is determined, and these schedules are integrated to get one for the query to be processed. Redundant transmissions are removed. From experimental results it was observed that optimal total transmission cost schedules for individual relations had very few transmissions in common. Hence, few redundant schedules could be removed. Therefore, another approach was adopted that allows for as few parallel schedules as possible, which were used for all relations.

Finally, insection 3.5 two types of query processing algorithms were compared. On the one hand, query processing algorithms that decompose the query at the logical level, which means that joins may be executed outside the result site. And, on the other hand, the algorithms proposed in section 3.4. Overall we may conclude that if only transmission cost is taken into account, the application of the semi-join operation decreases the cost considerably. If the domain sizes are small, the selectivities not too close to one or the same attribute is used in several clauses the algorithms that only use the semi-join operation produce with a smaller total transmission cost than the ones produced by a decomposition algorithm.

Now we will compare the results obtained for the inverted file organization and the relational data model. At first glance the assumption that the size of a result after

an intersection is neglectably small compared to the lists seems rather restrictive. In the relational data model a more sophisticated technique was used to estimate result sizes, but if we compare the schedules we notice that they are basically the same. The exact meaning of a basic operation is not important, what counts, when minimizing total transmission cost, is whether its result is greater than or equal to the sum of its operands minus the cost to collect the operands at one site. Under the assumption that the union belongs to this class we have shown that it should be computed at the result site. In the relational model a similar, however intuitive, approach has been taken for the join. Another approach is to assume that the join does not fall into this class and its computation should not be postponed. We also showed that operations whose results are less than or equal to the sum of its operands, such as intersection and semi-join, should be applied to large inverted lists or fragments before transmitting them to the result site. Hence, theorems that hold for certain basic operations, in general, also hold for other basic operations that fall in the same class.

When minimizing response transmission time under the Parallelism Assumption we may say that an operation should not be computed at another site than the result site if the cost to transmit the result of the operation to the result site is larger than the transmission cost of the largest operand to the result site minus the cost to collect the operands at one site. The union operation is an example of such an operation, and, depending on the size if the result, the join too.

The problems when minimizing response time and total time are fundamentally different. When minimizing response time, we like to enhance as much parallelism as possible, however, certain operations may use the same resource, affecting each other's response time. We do not think that an overall optimization that takes this sharing of resources into account is feasible and, therefore, an optimization in two stages was proposed. First, obtain the schedules that minimize response time under the Parallelism Assumption and then serialize these schedules such that no two operations use the same resource at the same time and such that again the response time is minimized.

Minimizing the total time is of a completely different character. The goal is to share as many basic operations as possible in the schedule. This can be achieved by removing redundant transmissions and operations and, hence, moving the forking points to the right on the time scale. One way of guaranteeing this is to compute prefabricated schedules that can be used in the schedule of all the objects to be transmitted.

5.2. Data Allocation

The problem of allocating the data of a database to the sites of a computer network has two aspects:

- ☐ the objects to be allocated,
- ☐ the final allocation should reflect the processing schedules for the queries produced by the query processing algorithm.

In section 4.1 the differences between the file allocation problem and the data allocation problem, were discussed. The file allocation problem deals with objects, which are determined prior to the allocation process, and these objects may only be accessed by simple processing schedules. For example, only transmissions from a site where a file resides to a result site are allowed. Therefore, the solutions of the file allocation problem are not suited for a distributed database.

In section 4.2 a brief overview was given of the many variations of the file allocation problem. Although not discussed in this monograph, all these variations can be applied to the data allocation problem as well.

In section 4.3 a model was presented, which makes it possible to discuss allocations under construction and their costs. Besides the notion of a completely specified allocation also a partially specified allocation was introduced. These are allocations where not all fragments or operations have been put in the fragment- and operation-sets of physical sites. This allows for feasible allocations under construction. To compute the cost of a completely or partially specified allocation a processing-schedules graph is used. Such a graph consists of physical and virtual sites (nucleus-sites); a physical site represents a site in a computer network and a virtual site is used for fragments and operations for which an allocation has not yet been determined. Between these nucleus-sites there are edges which represent transmissions. A processing-schedules graph is constructed by assuming that every physical site and every virtual site is a different site in a computer network and computing the processing schedules under this assumption by a query processing algorithm. These schedules are represented by transmissions between nucleus-sites and operations, which are put in the operation-sets of the nucleus-sites. From this processing-schedules graph the cost of the allocation can be computed by taking into account the assignments of virtual sites to other nucleus-sites.

To represent forking points in a processing schedules a forking graph was introduced, which has the property that if two receiving nodes are identified with the same nucleus-site the graph adjusts itself such that only one notification is sent to that nucleus-site.

Also, the objects to be allocated were discussed. The following approach was proposed. The relations in the global conceptual schema of a database are split horizontally and vertically, and the remaining fragments are used in computing a data allocation for the database. Vertically a relation is split based on its attributes, such that each resulting fragment contains only one attribute. Horizontally a relation is split based on the clauses in the queries and updates stated by users. The idea behind this splitting is that each tuple in a resulting fragment has the same probability of being referenced in a query. It was shown that, when minimizing total transmission cost, further splitting the fragments obtained will not decrease the total transmission cost if the processing schedules are static under splitting.

In section 4.4 a centralized approach for solving the data allocation problem was discussed. This approach is feasible if the database is managed in a centralized way, i.e., if there is a database administrator, which may decide about changes in the allocation of the data, or if the database management system itself may do so. In detail the construction of a processing-schedules graph of an allocation was discussed, which includes the processing schedules of all queries and updates and forking graphs to represent the updates of the copies of the fragments. It was shown that the problem of determining an allocation with minimum total transmission cost is NP-complete. To be able to compute the cost of an allocation more efficiently the notion of static processing schedules were introduced. The processing schedules are computed only once, under the assumption that all fragments are located at a different site in the computer network (initial allocation). To compute the cost of an allocation

slight modifications may be made to these schedules to reflect differences between the initial allocation and the allocation under consideration. An example of such a change is the deletion of a transmission.

In section 4.5 a decentralized approach to the data allocation problem was treated. This approach is especially applicable if there is not no central organization, which may decide about the data allocation. An example of such a case is the integration of already existing databases, each having their own database administrator. The following approach was taken. Each group of users may create private copies of fragments they need. Whether they will actually be created depends on how frequently the original fragments are updated. The allocation of these private copies can be determined in exactly the same way as was done in the centralized approach. Only the processing-schedules graphs are a lot smaller than in the centralized case, making it feasible to compute optimal allocations for groups of users. Because a private copy is accessed by a particular group of users they may decide to only periodically update these copies.

In section 4.6 a qualitative and quantitative comparison was made between the centralized and decentralized approach. We expect that the centralized approach is only feasible for databases that are managed in a centralized way. It has the disadvantage that changes in the access pattern may require a re-computation of the complete allocation. Furthermore, the additional cost caused by special constraints, such as a quick response time, have to be paid by all users. The advantage, on the other hand, is that, compared to the decentralized approach, more information about the data accesses is available, which will lead to allocations with a lower cost than the ones obtained in the decentralized approach. A quantitative comparison was made to show this. The advantages of the decentralized approach are that changes in access patterns, special constraints and periodically updated copies are easily incorporated.

In section 4.7 the problem of minimizing total transmission cost was treated. If the cost of a processing schedule is mainly determined by the transmissions involved and the communication channels in the computer network have a relatively low bandwidth it is best to minimize total transmission cost. Both the usage of static and dynamic schedules to compute the cost of an allocation were discussed. Also, a third alternative, called semi-dynamic schedules, was treated. For these three alternatives both an admissible heuristic estimator for the Heuristic Path Algorithm was supplied, as well as a heuristic algorithm that runs in polynomial time. For both the static and semi-dynamic schedules a heuristic algorithm, called *total data allocation*, was proposed. The solutions obtained by this algorithm had only a 3% higher total transmission cost than the optimal ones when using static schedules. An admissible heuristic estimator when dynamic schedules are used, is less efficient than when static schedules are used, because the computation of processing schedules for one-query-allocations satisfying a partially specified allocation is required. As heuristic algorithm *total data allocation* was used only it was not applied to a processing-schedules graph but to a LINK-graph. A comparison was made between the usage of static and dynamic schedules. Clearly, the dynamic approach produced allocations, which had lower total transmission cost only more computational effort was required; it will depend on the efficiency of the query processing algorithm used whether this approach is feasible.

To get a better insight in the constituents of the total transmission cost, allocations were computed for varying query/update ratio. Part of the *TTC* was formed by transmissions between virtual sites in the processing-schedules graph of the initial

allocation. This shows that only allowing for simple processing schedules, as is done in the file allocation problem, may lead to solutions which do not characterize the way the data is accessed in a distributed database. Furthermore, the average number of sites that participates in query processing, was counted. The results showed that in the completely specified allocations computed for the processing-schedules graphs generated most of the queries only accessed one or two sites.

Finally, the usage of primary copies was investigated. The conclusion was that, as far as total transmission cost is concerned, forcing all updates to use one particular copy of a fragment, called primary copy, leads to allocations with a higher *TTC*.

In section 4.8 minimizing the average response response time was investigated. In the first half simple processing schedules were considered to get a better insight in the problem and to develop tools, which can be used for arbitrary processing schedules. For the simple schedules both minimizing average response transmission time and average response processing time were treated. A general technique was developed to estimate the cost of a partially specified allocation, which was incorporated in the admissible estimators in this section. Heuristic algorithms were proposed and compared with the optimal ones. If the number of fragments referenced per query is small compared to the total number and does not exceed the number of sites in the computer network, the allocations produced by the heuristic algorithms are quite reasonable compared to the optimal ones. Although, we may say that the greedy algorithms do not perform as well as for minimizing total transmission cost. This is partly caused by constraints on the feasible solutions and the serialization of the schedules. Minimizing average response processing time was considered from both the system's point of view and the users'. Both problems were shown to be NP-complete. A heuristic algorithm was presented that, on the one hand, tried to maximize parallelism and, on the other hand, tried to keep the utilization factors of the sites in the computer network as small as possible.

The processing-schedules graph with arbitrary schedules contains virtual sites for fragments and for operations, because the latter do not necessarily have to be allocated to the same site as the fragments. To determine optimal allocations for minimizing response transmission time when arbitrary processing schedules are used, the processing-schedules graph belonging to a partially specified allocation is first transformed such that each virtual site is directly connected to a physical site. To this transformed processing-schedules graph the heuristic estimator *psa resp data* can be applied. It was shown that the heuristic estimator, which includes this transformation and the application of *psa resp data*, is admissible. A heuristic algorithm, called *resp operation allocation*, was presented, which determines an operation allocation given a data allocation, such that the response transmission time of a query is minimized. For tree-structured processing-schedules the operation allocation are optimal. This algorithm is used in *resp data operation allocation*, which computes the allocation for both the data and the operations.

To end this section on the data allocation problem we may conclude that the problem is essentially more complex than the file allocation problem, because the objects to be allocated have to be determined and because more complex processing schedules are used, an allocation for the objects can not be determined for each separately.

5.3. Future Research

In this monograph two problems, namely query processing and data allocation, were investigated. Comparison between the solutions produced by algorithms presented were compared by means of simulation. To value the merits of algorithms proposed by other researchers and the ones presented here, a comparison in a real distributed system would be preferable. Therefore, in the near future, more research effort will be required to actually construct distributed database management systems. Special attention should be given to the decentralized control of these systems both at the level of management and at the access level. As far as the efficiency is concerned, the decentralized control implies that the database management systems should be provided with more tools to increase the efficiency. Also in the area of concurrency control and crash recovery the decentralized control should be pursued. The database management systems should take into account that the underlying computer network, which may consists of hundreds of sites, is always changing.

Current research deals with databases, which can still be modeled with the relational, the hierarchical or the network data model. Because databases are barely getting accepted in industry, this line of research will continue for many years to come. However, the integration of existing databases will raise insurmountable problems, because the databases do not contain enough knowledge about the real world. Therefore, and also because there will be a need for systems that may help people with ordinary problems, such as facing bureaucracy, some of the research on databases will shift to the area of artificial intelligence to study "knowledge bases".

Another thing we have to keep in mind is, if people are getting more aware of the value of information and what can be done with it, the privacy and security problems will place a heavy burden on the owners of databases. Maybe, the ownership of the data about a person should be given to the person himself such that he can decide what may be done with it.

REFERENCES

[Adiba1978] Adiba, M., J.C. Chupin, R. Demolombe, G.Gardarin, and J. Le Bihan, "Issues in Distributed Data Base Management Systems: A Technical Overview," *Proc. 4th Int. Conf. Very Large Data Bases*, pp.89-110 (September 1978).

[Adiba1980a] Adiba, M., J.M. Andrade, P. Decitre, F. Fernandez, and Nguyen Gia Toan, "POLYPHEME: An experience in distributed database system design and implementation," *Proc. Int. Symposium on Distributed Data Bases*, pp.67-84 (March 1980).

[Adiba1980b] Adiba, M.E. and B.G. Lindsay, "Database Snapshots," *Proc. Conf. on Very Large Data Bases*, pp.86-91 (October 1980).

[Adiba1981] Adiba, M.E., "Derived Relations: A Unified Mechanism for Views, Snapshots and Distributed Data," *Proc. 7th Int. Conf. Very Large Data Bases*, pp.293-305 (September 1981).

[Aho1974] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

[Apers1978] Apers, P.M.G., "Distributed Query Processing with Inverted File Organization," IR 43, Vrije Universiteit, Amsterdam (December 1978).

[Apers1979a] Apers, P.M.G., "Critique on and Improvement of Hevner and Yao's Distributed Query Processing Algorithm," IR 48, Vrije Universiteit, Amsterdam (February 1979).

[Apers1979b] Apers, P.M.G., "Distributed Query Processing: Minimum Response Time Schedules for Relations," IR 50, Vrije Universiteit, Amsterdam (March 1979).

[Apers1980a] Apers, P.M.G., "Data Allocation and Distributed Query Processing," *Proc. ACM PACIFIC '80*, pp.48-54 (November 1980).

[Apers1980b] Apers, P.M.G., A.R. Hevner, and S.B. Yao, "Algorithm for Distributed Query Optimization," submitted for publication (1980).

[Apers1981a] Apers, P.M.G., "Redundant Allocation of Relations in a Communication Network," *Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp.245-258 (February 1981).

[Apers1981b] Apers, P.M.G., "Centralized or Decentralized Data Allocation," *Proc. 2nd Seminar on Distributed Data Sharing Systems*, pp.101-116 (June 1981).

[Apers1981c] Apers, P.M.G., R.M. Langefeld, and K.C. Swart, "Distributed Query Processing," preliminary version, Vrije Universteit, Amsterdam (1981).

[Astrahan1975] Astrahan, M.M. and D.D. Chamberlin, "Implementation of a Structured English Query Language," *Communications ACM* **18**(10), pp.580-588 (October 1975).

[Astrahan1976] Astrahan, M.M. et al., "System R: relational approach to database management," *ACM Trans. Database Systems* **1**(2), pp.97-137 (June 1976).

[Baldissera1979] Baldissera, C., G. Bracchi, and S. Ceri, "A Query Processing Strategy for Distributed Data Bases," *Proc. EURO-IFIP 1979*, pp.667-677, North-Holland Publ. Co. Amsterdam (1979).

[Bernstein1981a] Bernstein, P.A. and D.W. Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM* **28**(1) (January 1981).

[Bernstein1981b] Bernstein, P.A. and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* **13**(2), pp.185-221 (June 1981).

[Bernstein1981c] Bernstein, P.A., N. Goodman, E. Wong, C.L. Reeve, and J.B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. Database Systems* **6**(4), pp.602-625 (December 1981).

[Cardenas1975] Cardenas, A.F., "Analysis and Performance of Inverted Data Base Structures," *Communictions ACM* **18**(5), pp.253-263 (May 1975).

[Casey1972] Casey, R.G., "Allocation of Copies of Files in an Information Network," *Proc. AFIPS 1972 SJCC* **40**, pp.617-625, AFIPS Press, (1972).

[Cellary1980] Cellary, W. and D. Meyer, "A Simple Model of Query Scheduling in Distributed Data Base Systems," *Information Processing Letters* **10**(3), pp.137-147 (April 1980).

[Chamberlin1976] Chamberlin, D.D., "Relational Data-Base Management Systems," *ACM Computing Surveys* **8**(1) (March 1976).

[Charrel1981] Charrel, P.J., "L'Influence des Critères d'Application dans la Résolution du Problème de l'Allocation Optimale des Ressources d'une Base de Données Réparties," *Position Papers of the Second Seminar on Distributed Data Sharing Systems* (June 1981).

[Chu1969] Chu, W.W., "Optimal File Allocation in a Multiple-Computer Information System," *IEEE Trans. Computers* **C-18**, pp.885-889 (1969).

[Chu1973] Chu, W.W., "Optimal File Allocation in a Computer Networks," pp. 83-94 in *Computer-Communication Network*, ed. N. Abramson and F.F. Kuo, Prentice-Hall, Englewood Cliffs N.J. (1973).

[Chu1979] Chu, W.W. and P. Hurley, "A Model for Optimal Processing for Distributed Databases," *Proc. 18th IEEE COMPCON*, pp.116-122 (Spring 1979).

[Chu1980] Chu, W.W., L.J. Holloway, Min-Tsung Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer* (November 1980).

[Codd1970] Codd, E.F., "A relational model for large shared data banks," *Communications ACM* **13**(6), pp.909-917 (June 1970).

[Cook1971] Cook, S.A., "The Complexity of Theorem-Proving Procedures," *Proc. 3rd Annual ACM Symposium on Theory of Computing*, pp.151-158.

[Czarnik1975] Czarnik, B., S. Schuster, and D. Tsichritzis, "ZETA: A Relational Data Base Management System," *Proc. ACM Pacific Regional Conf.*, pp.21-25 (April 1975).

[Daniels1982] Daniels, D., "Query Compilation in a Distributed Database System," RJ3423, IBM Research Laboratory, San Jose, Calif. (March 1982). thesis

[Date1977] Date, C.J., *An Introduction to Database Systems,* Addison-Wesley, Reading, Mass. (1977).

[Davida1981] Davida, G.I, "The Case Against Restraints on Non-Governmental Research in Cryptography," *Communications ACM* **24**(7), pp.445-450 (July 1981).

[Denning1982] Denning, P.J., "A Scientific's View of Government Control Over Scientific Publication," *Communications ACM* **25**(2), pp.95-97 (February 1982).

[Elam1978] Elam, J., "A Model for Distributing a Database," 78-1, Dept. of Decision Sciences, The Wharton School, University of Pennsylvania, Philadelphia (March 1978).

[Enslow1978] Enslow, P.H., "What is a Distributed Data Processing System," *Computer* **11**(1), pp.13-21 (January 1978).

[Epstein1979] Epstein, R., M.R. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Data Base System," *Proc. ACM-SIGMOD*, pp.169-180 (May 1979).

[Epstein1980a] Epstein, R., "Query Processing Techniques for Distributed Data Base Systems," PhD. Thesis Memorandum No. UCB/ERL M80/9, Univ. Calif. Berkeley (March 1980).

[Epstein1980b] Epstein, R. and M.R. Stonebraker, "Analysis of distributed data base processing strategies," *Proc. Sixth Conf. on Very Large Data Bases*, pp.92-5 (October 1980).

[Eswaran1974] Eswaran, K.P., "Placement of records in a file and file allocation in a computer network," *Information Processing 1974*, pp.304-307, North-Holland Publ. Co. Amsterdam (1974).

[Eswaran1976] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System," *Communications ACM* **19**(11), pp.624-633 (November 1976).

[Garcia-Molina1979a] Garcia-Molina, H., "Performance of update algorithms for replicated data in a distributed database," Ph.D. dissertation, Computer Science Dept., Stanford University, Stanford, Calif. (June 1979).

[Garcia-Molina1979b] Garcia-Molina, H., "A concurrency control mechanism for distributed data bases which use centralized locking controllers," *Proc. 4th Berkeley Workshop Distributed Databases and Computer Networks* (August 1979).

[Garey1979] Garey, M.R. and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness,* Freeman (1979).

[Griffiths1976] Griffiths, P.P. and B.W. Wade, "An Authorization Mechanism for a Relational Database System," *ACM Transactions Database Sytems* **1**(3), pp.242-255 (September 1976).

[Hansler1972] Hansler, E., G.K. McAuliffe, and R. S. Wilkov, "Exact calculation of computer network reliability," *Proc. AFIPS 1972, FJCC* **41**, pp.49-54, Pt. I. AFIPS PRESS (1972).

[Held1975] Held, G.D., M.R. Stonebraker, and E. Wong, "INGRES - A Relational Data Base System," *Proc. NCC* **44** (1975).

[Hevner1979a] Hevner, A.R. and S.B. Yao, "Query Processing in Distributed Database Systems," *IEEE Transactions on Software Engineering* **SE-5**(3), pp.177-187 (May 1979).

[Hevner1979b] Hevner, A.R., "The Optimization of Query Processing on Distributed Database Systems," PhD thesis, Purdue University (December 1979).

[Hevner1981] Hevner, A.R., "A Survey of Data Allocation and Retrieval Methods for Distributed Systems," MS/S 81-036, University of Maryland (October 1981).

[Hill1978] Hill, E., "Analysis of An Inverted Data Base Structure," *SIGIR*, pp.37-64 (1978).

[Hoffer1975] Hoffer, J.A. and D.G. Severance, "The use of cluster analysis in physical data base design," *Proc. First Conf. on Very Large Data Base*, pp.69-86 (September 1975).

[Horowitz1978] Horowitz, E. and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press (1978).

[Hsiao1970] Hsiao, D. and F. Harary, "A Formal System for Information Retrieval from Files," *Communications ACM* **13**(2), pp.67-73, corrigenda, Communications ACM, vol. 13, no. 4, p. 266 (April 1970) (February 1970).

[Huffman1952] Huffman,, "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE* **40**, pp.1098-1101 (September 1952).

[Jackson1957] Jackson, J.R., "Networks of Waiting Lines," *Operations Research* **5**, pp.518-521 (August 1957).

[Karp1972] Karp, R.M., "Reducibility Among Combinatorial Problems," pp. 85-103 in *Complexity of Computer Computations*, ed. R.E. Miller and J.W. Tatcher, Plemun Press, New York (1972).

[Kersten1981] Kersten, M.L. and A.I. Wasserman, "The Architecture of the PLAIN Data Base Handler," *Software - Practice and Experience* **11** (1981).

[Kim1979] Kim, Won, "Relational Database Systems," *ACM Computer Surveys* **11**(3), pp.185-226 (September 1979).

[Kleinrock1975a] Kleinrock, L., *Queueing Systems, Volume 1: Theory*, Wiley, New York (1975).

[Kleinrock1975b] Kleinrock, L., *Queueing Systems, Volume 2: Computer Applications*, Wiley, New York (1975).

[Kohler1981] Kohler, W.H., "A survey of techniques for synchronization and recovery in decentralized computer systems," *ACM Computer Surveys* **13**(2), pp.149-183 (June 1981).

[Lamport1978] Lamport, L., "Time, clocks and ordering of events in a distributed data storage system," *Communications ACM* **21**(7), pp.558-565 (July 1978).

[Lawler1962] Lawler, E.L., "The Quadratic Assignment Problem," *Management Science* **9**, pp.586-599 (1962).

[Lawler1966] Lawler, E.L. and D.E. Wood, "Branch-and-Bound Methods:.A Survey," *Operations Research* **14**(4), pp.699-719 (July 1966).

[LeBihan1980] LeBihan, J., C. Esculier, G. LeLann, and L. Treille, "SIRIUS-DELTA: Un prototype de système de gestion de bases de données réparties," *Proc. Int. Symposium on Distributed Data Bases*, pp.137-159 (March 1980).

[LeLann1978] LeLann, G., "Algorithms for distributed data-sharing systems which use tickets," *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, pp.259-272 (August 1978).

[Levin1974] Levin, K.D., "Organizing Distributed Data Bases in Computer Networks," Ph.D. Thesis, The Wharton School, University of Pennsylvania,, Philadelphia (September 1974).

[Levin1975] Levin, K.D. and H.L. Morgan, "Optimizing Distributed Databases- A Framework for Research," *Proc. 1975 AFIPS NCC* **44**, pp.pp.473-478, AFIPS Press (1975).

[Litwin1980] Litwin, W., "A Model for a Distributed Database," *ACM 2nd Annual Louisiana Computer Exposition* (February 1980).

[Liu1976] Liu, J.W.S., "Algorithms for Parsing Search Queries in Systems with Inverted File Organization," *ACM Transactions Database Systems* **1**(4), pp.299-316 (December 1976).

[Mahmoud1976] Mahmoud, S. and J.S. Riordon, "Optimal Allocation of Resources in Distributed Information Networks," *ACM Transactions on Database Systems* **1**(1), pp.66-78 (March 1976).

[Martella1981] Martella, G., B. Ronchetti, and F.A. Schreiber, "On Evaluating Availability in Distributed Database Systems," *Proc. 5th Berkely Workshop Distributed Data Management and Computer Networks*, pp.154-171 (February 1981).

[Martin1975] Martin, J., *Computer Data-Base Organization,* Prentice-Hall, Englewood Cliffs, N.J. (1975).

[McQuillan1977] Mcquillan, and Walden, "The ARPA network Design Decisions," *Computer Networks* **1**, pp.243-289 (August 1977).

[Metcalfe1976] Metcalfe, R.M. and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications ACM* **19**, pp.395-404 (July 1976).

[Neuhold1977] Neuhold, E.J. and H. Biller, "POREL: A Distributed Data Base on an Inhomogeneous Computer Network," *Proc. Third Int. Conf. on Very Large Data Bases*, pp.380-389 (October 1977).

[Ng1982] Ng, P., "Distributed Compilation and Recompilation of Database Queries," RJ3375, IBM Research Laboratory, San Jose, Calif. (January 1982). thesis

[Nguyen Gia Toan1979] Nguyen Gia Toan, "A unified method for query decomposition and shared information updating in distributed systems," *First Int. Conf. on Distributed Computing Systems*, pp.679-685 (October 1979).

[Nguyen Gia Toan1980] Nguyen Gia Toan, "Decentralized Dynamic Query Decomposition for Distributed Database Systems," *Proc. ACM Pacific '80*, pp.55-60 (November 1980).

[Nilsson1971] Nilsson, N.J., *Problem-Solving Methods in Artificial Intelligence,* McGraw-Hill, New York (1971).

[Paik1979] In-Sup Paik, and C. Delobel, "A strategy for optimizing the distributed query processing," *Proc. First Int. Conf. on Distributed Computing Systems*, pp.686-698 (October 1979).

[Palermo1974] Palermo, F.P., "A Data Base Search Problem," *Information Systems: COINS IV*, Plenum Press (1974).

[Pelagatti1979] Pelagatti, G. and F.A. Schreiber, "A Model of an Access Strategy in a Distributed Database," *IFIP-TC2, Data Base Architecture* (June 1979).

[Pohl1972] Pohl, I., "Is Heuristic Search Really Branch-and-Bound?," *Proc. 6th Princeton IEEE Symposium on Information Sciences and Systems*, pp.370-373 (March 1972).

[Ramamoorthy1979] Ramamoorthy, C.V. and B.W. Wah, "The Placement of Relations on a Distributed Relational Database," *Proc. of the 1st International Conference on Distributed Computing Systems*, pp.642-650 (October 1979).

[Rosenkrantz1978] Rosenkrantz, D.J., R.E. Stearns, and P.M. Lewis, "System level concurrency control for distributed database systems," *ACM Trans. Database Systems* **3**(2), pp.178-198 (June 1978).

[Rosenthal1981] Rosenthal, A.S., "Note on the Expected Size of a Join," *SIGMOD RECORD* **11**(4), pp.19-25 (July 1981).

[Rothnie1977a] Rothnie, J.B. and N. Goodman, "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases," *Proc. 2nd Berkeley Workshop Distributed Data Management and Computer Networks*, pp.39-57 (May 1977).

[Rothnie1977b] Rothnie, J.B. and N. Goodman, "A survey of research and development in distributed database management," *Proc. 3rd Int. Conf. Very Large Data Bases*, pp.48-62, IEEE (October 1977).

[Sang Ajang1981] Sang Ajang, G. and E. Spoor, "Ideeën voor het ontwerpen van een gedistribueerd database management systeem," Master Thesis, Vrije Universiteit, Amsterdam (1981). in Dutch

[Schmid1975] Schmid, H.A. and P.A. Bernstein, "A Multi-Level Architecture for Relational Data Base Systems," *Proc. Int. Conf. Very Large Data Bases*, pp.202-226 (Sepetember 1975).

[Selinger1979] Selinger, P.G., M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Acces Path Selection in a Relational Database Management System," *Proc. ACM Conference*, pp.23-34 (May 1979).

[Selinger1980] Selinger, P.G. and M.E. Adiba, "Access Path Selections in Distributed Data Base Management Systems," *Proc. Int. Conf. on Databases*, pp.204-215 (July 1980).

[Stonebraker1976] Stonebraker, M., E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES," *ACM Trans. Database Systems* **1**(3), pp.189-222 (September 1976).

[Stonebraker1977] Stonebraker, M.R. and E. Neuhold, "A Distributed Version of INGRES," *Proc. 2nd Berkeley Workshop Distributed Data Management and Computer Networks*, pp.19-36 (May 1977).

[Stonebraker1979] Stonebraker, M., "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Trans. Software Eng.* **SE-5**(3), pp.188-194 (May 1979).

[Tanenbaum1981] Tanenbaum, A.S., *Computer Networks,* Prentice-Hall, Englewood Cliffs, N.J. (1981).

[Thomas1979] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions Database Systems* **4**(2), pp.180-209 (June 1979).

[Toth1978] Toth, K.C., S.A. Mahmoud, J.S. Riordon, and O. Sherif, "The ADD System: An Architecture for Distributed Databases," *Proc. 4th Int. Conf. Very Large Data Bases*, pp.462-471 (September 1978).

[Toth1980] Toth, K.C., "Distributed Database Architecture and Query Processing Strategies," PhD thesis, Dept. of Systems and Computer Engineering Carleton University, Ottawa (June 1980).

[Toth1981] Toth, K.C., S.A. Mahmoud, and J.S. Riordon, "Query Processing Strategies in a Distributed Database Architecture," *Proc. 2nd Seminar on Distributed Data Sharing Systems* (June 1981).

[Tsichritzis1977] Tsichritzis, D.C. and F.H. Lochovsky, *Data Base Management Systems,* Academic Press, New York (1977).

[Ullman1980] Ullman, J.D., *Principles of Database Systems,* Pitman Publ. (1980).

[Wasserman1981] Wasserman, A.I. et al., "Revised Report on the Programming Language PLAIN," *SIGPLAN* **16**(5), pp.59-80 (May 1981).

[Williams1981] Williams, R. et al., "R*: An Overview of the Architecture," RJ 3325, IBM Research Laboratory, San Jose, Calif. (December 1981).

[Wilms1980] Wilms, P., "Qualitative and quantitative comparison of update algorithms in distributed databases," *Proc. Int. Symposium Distributed Databases*, pp.275-294 (March 1980).

[Wong1976] Wong, E. and K. Youssefi, "Decomposition - A Strategy for Query Processing," *ACM Trans. Database Systems* **1**(3), pp.223-241 (September 1976).

[Wong1977] Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Data Bases," *Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks*, pp.217-235 (May 1977).

[Yao1977] Yao, S.B., "Approximating Block Accesses in Database Organizations," *Communications ACM* **20**(4), pp.260-261 (April 1977).

[Zimmerman1980] Zimmermann, H., "OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions Communication* **COM-28**, pp.425-432 (April 1980).

INDEX

A

A-list 49
ADD 12
adjacent 132
admissible 126
allocation, cost of an 103
  one-query- 137
  unit of 105
  of the global data dictionary 110
application layer 8
arbitrary schedule 163
Arbitrary Transmission Cost Model 21, 28, 58
architecture of a database 5
ARPT 158
arrival distribution 148
ART 147
ARTT 150
assigned 100
assigned set 100
attribute 8
  dependence 83
  independence 83
authentication 16
authorization 16
availability 1, 14, 5
average response processing time 156
  heuristic allocation 159
  optimal allocation 158
average response time 147, 163
average response transmission time, heuristic allocation 153
  optimal allocation 150

B

B-list 49
balanced split 108
basic operations 2, 19, 27
being assigned to 100
branch-and-bound 123
branch structure 32
bulk size distribution 148

C

candidate nucleus-site 166
Cartesian product 8
centralized data allocation 4, 110, 116
centralized locking 111
centroiding phase 34
check point 13
Class A query 36
clique 157
clocks 12
cluster 132
  simple 132
clustering phase 34
coefficient of variation 148
COLLECTIVE 90
commit 13
  two-phase 13
compile 37
complement 40
completely inverted system 40
completely specified allocation 101
computer network 8

conc 67
conceptual schema 5
  global 1, 5
concurrency 1, 12
conjunction 44
consistency 12
  mutual 144
  strong 144
  weak 144
constituents of total transmission cost 141
construction of processing-schedules graph 101, 110
copies, primary 144
  private 114
  secondary 144
correlation 35
cost, estimate- 124
cost function, response transmission time 21
  response time 2, 20
  total cost 2, 20
  total time 20
  total transmission cost 21
  total transmission time 21
cost of an allocation 103
  of a path of length greater than 1 125
  of a schedule 20
  of a subset 124

D

D-INGRES 30
data allocation 1, 150, 165
  centralized 110, 116
  decentralized 114, 116
data allocation problem 3, 99
  NP-completeness of 112
data and operation allocation 98, 163
data dictionary 17
  allocation of the global 110
data link layer 8
data model, hierarchical 8
  network 8
  relational 8
database, fully replicated 14
  partitioned 14
database management system, relational 11
databases, integration of 1
decentralized data allocation 4, 114, 116
decomposition 30, 92
difference 9
directory 39
disjunctive normal form 44
distributed databases 1, 5
distributed database management systems 5, 8, 11
distributed data processing system 5
distributed operating system 8
distributed query processing 17, 41, 73
  algorithms 28
Distribution Strategy 35
domain 8
domino effect 13
distributor 22
dynamic optimization 22, 29, 31, 37
dynamic schedules 112, 114, 139
  heuristic allocation using 138
  optimal allocation using 136

E

encryption 16
end node 33
Equal Transmission Cost Model 21, 28, 53
equi-join 9
estimate-cost 124
EXHDECOM 93
*EXHDECOM** 93
expansion 1, 5
export-operation 165
external schema 5

F

file allocation problem 3, 97, 99
file node 33

filtering phase 34
forking graph 104
    point 22
fragment 14, 105
    virtual site 163
fully replicated database 14

G

general query 79
global conceptual schema 1, 5
global data dictionary, allocation of 110
global internal schema 5
greedy 30, 127

H

heterogeneous 6
heuristic allocation average response processing time 159, 161
    average response transmission time 153, 155
    total transmission cost 127, 132, 134
    using dynamic schedules 138
    using static schedules 127, 134
heuristic operation and data allocation *ARTT* 169
Heuristic Path Algorithm 123
hierarchical data model 8
homogeneous 6
horizontal split 14, 106

I

incoming selectivity 75
Independence Theorem 25
INGRES 11
initial allocation 101
input nucleus-site 166
integration 1, 5, 64, 84
integrity 1, 12
internal schema 5
intersection 9
Intersection Assumption 44
inverted file organization 38, 39
inverted list 39
ISO proposal 8

J

join 9, 29, 107
    equi- 9
    natural 9
    semi- 10, 30, 32, 73, 92

L

legislation, privacy 1
length greater than 1, cost of a path of 125
length of a path 124
linear structure 32
LINK 127
LINK-graph 139
list 39
literal 44
local conceptual schema 5
    external schema 6
    internal schema 5
Local Processing Assumption 21, 28
locking 12
locking, centralized 111
    two-phase 12
Logical Strategy 35

M

macro-schedule 39, 63
materialization 15, 21, 29, 34, 92
max parallelism 159
merge join 37
micro-schedule 39, 64
MRTT query 48, 57, 61

MRTT term 56, 60
mutual consistency 21, 144

N

natural join 9
nearest physical site 164
nested loop join 36
network data model 8
    layer 8
    operation system 8
non-result site 65
not-at-same-site graph 157
notification node 104
nucleus-site 100

O

OMEGA 11
one-query allocation 137
operation 100
    basic 2
    export- 165
    allocation 156, 165
    node 33
    tree 33, 37, 41
    virtual site 163
optimal allocation average response processing time 158, 161
    average response transmission time 150, 155
    using dynamic schedules 134, 136
    using static schedules 123
optimal operation and data allocation *ARTT* 163
optimization, dynamic 22
    static 22
OSI model 8

P

PARALLEL 76
parallelism 32
Parallelism Assumption 2, 22
parallel schedules 22
parser 17
partially specified allocation 101
    satisfy a 124
partitioned database 14
path 124, 164
    length of a 124
    removing a 125
    of length greater than 1, cost of a 125
periodic update 115
physical layer 8
    site 100
physical site 3
PLAIN 11
POLYPHEME 12
POREL 12
precompile 35
prefix schedule 85
presentation layer 8
primary copies 144
privacy 16
    legislation 1
private copies 114
processing-schedules graph 4, 101
    construction of 101, 110
    graphical representation 102
    simple 132
processing schedule 19
    time 19
projection 9, 107
psa dynamic cost 138
psa resp data 151, 164
psa resp operation 158
psa static cost 125

Q

Query By Example 11
query optimizer 19
query processing 1, 15
    distributed 17
queueing model 148

R

receiving node 104
recovery 13
redundant transmission 84
relation 8
relational algebra 8
    calculus 10
    database management system 11
    data model 8
reliability 14
removing a path 125
RESPONSE GENERAL 79
response time 2, 20, 22, 62, 148
response transmission time 21, 43, 61, 79
RTT 21
resp data operation allocation 169
resp operation allocation 167, 169
restriction 107
result site 66

S

satisfy a partly specified allocation 124, 137
schedule 19, 29, 75
    arbitrary 163
    cost of a 20
    dynamic 112, 114
    semi-dynamic 135, 136
    serialized 2, 25
    simple 149
    static 11
    tree-structured 166
    parallel 22
schema 5
    conceptual 5
    external 5
    internal 5
SDD-1 29
secondary copies 144
security 1, 16
selection 9
selectivity 30, 74, 92
Selectivity Assumption 76
semi-dynamic schedules 135, 136
semi-join 10, 30, 32, 73, 92
SERIAL 76
serializability 12
serialization 61
    of a schedule 25
serialized schedule 2
serializing parallel schedules 25
service distribution 148
session layer 8
set operations 9
simple cluster 132
    processing-schedules graph 132
    query 30, 76
simple resp data allocation 154
simple resp operation allocation 160
simple schedule 149
SIRIUS-DELTA 12
SJ EXHDECOM 93
split, balanced 108
    horizontal 106
    vertical 106
splitting, static under 107
stable storage 13
static optimization 22, 29
static schedules 112, 139
    heuristic allocation using 127
    optimal allocation using 123
static under splitting 107
strong consistency 144
synchronization point 22
System R 11
System R* 36

T

target 10
term 44
time stamp 12
token 13, 37
total cost 2, 20
total data allocation 129, 132, 134, 138
    extensions of 141
TOTAL GENERAL 88
total ordering 12
total time 20, 66
total transmission cost 21, 122

constituents of 141
total transmission time 21, 84
Transmission Assumption 21, 28
Strategy 35
transmission time 19
transport layer 8
tree-structured schedule 166
TTC 21
TTT 21
TTT query 68
TTT term 71
tuple 8
substitution 31
two-phase commit 13
locking 12

U

undo 13
union 9, 100
unit of allocation 105
update, periodic 115
utilization factor 149

V

vertical split 14, 106
structure 32
virtual ring 13, 37
site 3, 100

W

wait-for graphs 12
weak consistency 144

Z

ZETA 11