Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

Distribution Design for Complex Value Databases

Hui Ma

2007

A dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Information Systems at Massey University

Abstract

Distribution design for databases usually addresses the problems of fragmentation, allocation and replication. However, the main purposes of distribution are to improve performance and to increase system reliability. The former aspect is particularly relevant in cases where the desire to distribute data originates from the distributed nature of an organization with many data needs only arising locally, i.e., some data are retrieved and processed at only one or at most very few locations. Therefore, query optimization should be treated as an intrinsic part of distribution design. Due to the interdependencies between fragmentation, allocation and distributed query optimization it is not efficient to study each of the problems in isolation to get overall optimal distribution design. However, the combined problem of fragmentation, allocation and distributed query optimization is NP-hard, and thus requires heuristics to generate efficient solutions.

In this thesis the foundations of fragmentation and allocation in databases on query processing are investigated using a query cost model. The considered databases are defined on complex value data models, which capture complex value, object-oriented and XML-based databases. The emphasis on complex value databases enables a large variety of schema fragmentation, while at the same time it imposes restrictions on the way schemata can be fragmented. It is shown that the allocation of locations to the nodes of an optimized query tree is only marginally affected by the allocation of fragments. This implies that optimization of query processing and optimization of fragment allocation are largely orthogonal to each other, leading to several scenarios for fragment allocation. Therefore, it is reasonable to assume that optimized queries are given with subqueries having selection and projection operations applied to leaves. With this assumption some heuristic procedures can be developed to find an "optimal" fragmentation and allocation. In particular, cost-based algorithms for primary horizontal and derived horizontal fragmentation, vertical fragmentation are presented.

Acknowledgements

I would like to thank Professor Klaus-Dieter Schewe, my supervisor, for his attentive guidance, endless patience, invaluable advice and constant support during this research. Especially I am thankful for the opportunity offered by him to step into an academic career. It is he who showed me how to do research step by step. It is also he who always encouraged me to make me more and more confident and enthusiastic about doing research.

I am also thankful to my co-supervisor, Professor Sven Hartmann, for his support, patience, encouragement and guidance during this research. Especially, my deep appreciation owns to Professor Sven Hartmann for his always being available for discussing and answering questions, not only during week days, working hours but also during weekends, holidays, late nights, through the whole period of my study.

In the mean time, I would like to express my appreciation of the understanding and help from Markus Kirchberg and Sebastian Link, who have helped me during my study in their own ways.

Finally, I am grateful to my husband, my parents, and my son, for their support, patience and *love*. Without them this work would never have come into existence (literally).

Table of Contents

1 Introduction **1**

1.1 Distributed Databases: Definition and Motivation 1

1.2 Database Distribution Design 3

1.2.1 Design Techniques: Fragmentation and Allocation 3

1.2.2 Alternative Design Strategies 4

Top-down Approach. 4

Bottom-up Approach. 5

1.2.3 The Objective of the Design of Data Distribution 5

1.3 Distribution Design Dilemma 6

1.3.1 The Complexity of the Problems 6

1.3.2 Interdependencies with Query Optimization 7

1.3.3 Ad Hoc Solutions 7

1.4 Contributions 8

1.5 The Outline of the Thesis 9

2 Literature Review on Distribution Design for Databases **11**

2.1 Horizontal Fragmentation 11

2.1.1 Primary Horizontal Fragmentation for Relational Databases 11

2.1.2 Primary Horizontal Fragmentation for Object Oriented Databases . . 14

2.1.3 Derived Horizontal Fragmentation for Relational Databases 15

2.1.4 Derived Horizontal Fragmentation for Object Oriented Databases . . . 16

2.2 Vertical Fragmentation 18

2.2.1 Vertical Fragmentation for Relational Databases 18

Affinity-Based Approach 18

Cost-Driven Approach 20

2.2.2 Vertical Fragmentation for Object Oriented Databases 21

2.3 Mixed Fragmentation 22

2.4 Allocation 23

2.4.1 Simple Query Environment 23

2.4.2 Query Site Strategy 24

2.4.3 Others 26

2.5 Summary 26

3 Complex Value Databases and Query Language **29**

3.1 The Data Model 29

3.1.1 Types and Trees 29

3.1.2 Subtypes . . . 33
3.1.3 Rational Trees . . . 34
3.2 Query Algebra and Optimization . . . 36
3.2.1 A Generic Query Algebra . . . 37
3.2.2 Path Expressions . . . 38
3.2.3 A Simple Query Algebra . . . 38
3.3 A Query Processing Cost Model . . . 41
3.3.1 Query-Trees . . . 41
3.3.2 Size Estimation for Leaves of a Query Tree . . . 41
3.3.3 Size Calculation for Intermediate Nodes of a Query Tree . . . 42
3.3.4 Query Processing Costs . . . 43

4 Fragmentation Operations 45
4.1 Horizontal Fragmentation . . . 45
4.2 Vertical Fragmentation . . . 47
4.3 Splitting . . . 49
4.4 Correctness Rules for Fragmentation . . . 50

5 Foundations of Fragmentation and Allocation 53
5.1 The Impact of Splitting on Query Costs . . . 53
5.1.1 Scenario I . . . 54
5.1.2 Scenario II . . . 54
5.1.3 Scenario III . . . 55
5.2 The Impact of Horizontal Fragmentation on Query Costs . . . 56
5.2.1 Scenario I . . . 56
5.2.2 Scenario II . . . 57
5.2.3 Scenario III . . . 58
5.3 The Impact of Vertical Fragmentation on Query Costs . . . 59
5.3.1 Scenario I . . . 59
5.3.2 Scenario II . . . 60
5.3.3 Scenario III . . . 61
5.4 Summary . . . 61

6 Heuristics for Horizontal Fragmentation and Allocation 63
6.1 Primary Horizontal Fragmentation and Allocation . . . 63
6.1.1 Fragmentation and Allocation Refinement for Horizontal Fragments . 69
6.1.2 An Example . . . 70
Fragmentation Step 1 . . . 72
Fragmentation Step 2 . . . 75
6.1.3 Simple Selection Predicates for Horizontal Fragmentation . . . 76
6.1.4 Experimental Evaluation . . . 78
6.2 Derived Horizontal Fragmentation . . . 80
6.2.1 A Motivating Example . . . 80
6.2.2 Some Terms . . . 83
6.2.3 Heuristics for Derived Horizontal Fragmentation . . . 85
6.2.4 An Example . . . 86
6.2.5 Discussion . . . 87

6.2.6 Experimental Evaluation 89

7 Heuristics for Vertical Fragmentation and Fragment Allocation 91

7.1 A Motivating Example 91
7.2 Some Terms 94
7.3 A Heuristic Approach for Vertical Fragmentation 96
7.4 Examples 98
7.5 Discussion 102

8 Conclusions 105

8.1 Summary 105
8.2 Open Problems and Challenges 106

Bibliography 109

List of Figures

3.1 HERM Diagram of the University Database 30
3.2 Example of a Query Tree 42

5.1 Scenario I for Query Tree Rewriting in Case of Splitting Fragmentation . . . 54
5.2 Scenario II for Query Tree Rewriting in case of Splitting Fragmentation . . . 55
5.3 Scenario III for Query Tree Rewriting in Case of Splitting Fragmentation . . 56
5.4 Scenario I for Query Tree Rewriting in Case of Horizontal Fragmentation . . 57
5.5 Scenario II for Query tree Rewriting in Case of Horizontal Fragmentation . . 58
5.6 Scenario III for Query Tree Rewriting in Case of Horizontal Fragmentation . 59
5.7 Scenario I for Query Tree Rewriting in Case of Vertical Fragmentation 60
5.8 Scenario II for Query Tree Rewriting in Case of Vertical Fragmentation . . . 61
5.9 Scenario III for Query Tree Rewriting in Case of Vertical Fragmentation . . . 62

6.1 Allocation without Fragmentation 71
6.2 Reallocation of Fragments: Step 1 72
6.3 Reallocation of Fragments: Step 2 74
6.4 Reallocation of Fragments: Step 3 76
6.5 Simple Predicate and Query Costs 79
6.6 Scenario I - Primary Fragmentation Only 81
6.7 Scenario II - Derived Fragmentation according to One Fragmentation Schema 81
6.8 Scenario III - Derived Fragmentation according to Two Fragmentation Schemata 82

7.1 Scenario I: Fragmentation with Two Queries at One Location 92
7.2 Scenario II: Fragmentation with Two Queries at Different Locations 92
7.3 Scenario III: Fragmentation with Two Queries at Different Locations 93
7.4 Scenario IV: Fragmentation with Two Queries at Different Locations 93
7.5 Allocation of Fragments for Example 7.3 99

List of Tables

3.1 An Instance of the University Schema 32
3.2 An Instance of the University Schema (continued from Table 3.1) 33

4.1 Horizontal Fragmentation of *db*(DEPARTMENT) 46
4.2 Horizontal Fragmentation of *db*(PAPER) 47
4.3 Derived Horizontal Fragmentation of *db*(LECTURE) 48
4.4 Vertical Fragmentation of *db*(LECTURE) 50

6.1 Atomic Fragment Request Matrix 67
6.2 Atomic Fragment Pay Matrix 67
6.3 Atomic Fragment Allocation 68
6.4 Horizontal Fragmentation of *db*(LECTURER) 68

7.1 Attribute Usage Frequency Matrix for Example 7.3 99
7.2 Attribute Request Matrix for Example 7.3 99
7.3 Attribute Pay Matrix for Example 7.3 99
7.4 Attribute Usage Frequency Matrix for Example 7.4 100
7.5 Attribute Request Matrix for Example 7.4 101
7.6 Transportation Cost Factors for Example 7.4 101
7.7 Attribute Pay Matrix for Example 7.4 101
7.8 Attribute Allocation for Example 7.4 102

Chapter 1

Introduction

Database distribution design is an important research area because it is critical to the success of applications that facilitate organizations to provide access to and sharing of data and information to users from different geographical locations. The investigation of distributed databases systems (DDBS) started in the 1970s [29, 98]. This chapter presents a brief context for the study of database distribution design. In the following of this chapter, the definition of a distributed database and motivations for developing distributed databases will be reviewed first. Further, two main distribution design techniques, fragmentation and allocation, will be briefly defined, followed by design strategies and objectives. Furthermore, a fundamental distribution design dilemma will be discussed to show the efficiencies of studying fragmentation and allocation for complex value databases. Finally, the contributions of this thesis will be listed and the structure of the thesis will be presented.

1.1 Distributed Databases: Definition and Motivation

What is a distributed database? Ceri and Pelagatti [26] defined a distributed database as a collection of data that logically belongs to the same system but is spread over the sites of a computer network. Özsu and Valduriez [93] gave a similar definition: that a distributed database system is a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (DDDMS) is then defined in [93] as the software system that provides the management of the distributed database system and makes the distribution transparent to the users .

Ceri and Pelagatti [26] emphasized that the data at different sites must have properties that tie them together, and that access to the files should be via a common interface. Özsu and Valduriez [93] explained that the logically related files, which are individually stored at each site of a computer network, are not enough to form a distributed database. There needs to be a structure among them. They explained that physical distribution means that data does not reside at the same site in the same processor. It is pointed out in [93] that physical distribution does not necessarily imply that the computer systems are geographically distributed. The sites among the network could even have the same address. They could be in the same room, but the communication between them is done over a network instead of shared memory, and the communication network is the only shared resource.

With the technological advance of communication, hardware, software protocols and standards, developing distributed database systems become more and more feasible and needed.

To meet the requirement, almost all major database system vendors offer products to support database distribution, e.g., IBM, Oracle, Microsoft, Sybase [67]. The motivation for the development of distributed databases can be briefly described as the properties below:

- *Reliability and Availability.* Reliability refers to the ability to tolerate faults. Availability refers to the probability that a system is available during desired time periods. Improved reliability and availability are potential advantages of distributed databases, which centralized databases lack [26, 93]. When replica of data have been placed at different sites, the crash of one site or the failure of the communication link would not necessarily make the data inaccessible. When the system crashes and the communication link fails, even though some of the data will not be accessible, the distributed database system still provides limited services [26, 93].
- *Organizational Reasons.* With the advances of telecommunication techniques, for many organizations, especially global organizations that are decentralized, implementing information systems in a decentralized way might be more suitable [7, 26, 53].
- *Interoperability of Existing Databases.* For organizations in which there already exist several databases and there is the necessity of executing global applications, integrated distributed databases are the natural solution, being designed bottom-up from existing local databases [67, 93].
- *Expandability.* It is easier to accommodate increasing database sizes in a distributed environment if an organization grows by adding new and relatively autonomous branches or warehouses, with the minimum degree of impact on the existing system [26, 93].
- *Local Autonomy.* This refers to the fact that data in a distributed database can be placed to the site that users work and local controls can be allocated to local users to enable them to take partial responsibility for information management in the distributed database [93].
- *New applications.* Many new applications rely heavily on distributed database technology, including computer-supported collaborative system, tele-conferencing and electronic commerce, and workflow management [67].
- *Performance.* This refers to the ability to reduce query response time and increase throughput by improving data localization which, in turn, can reduce the size of data that need to be transported and reduce contention for CPU and I/O services [92]. Meantime, inter-query parallelism and intra-query parallelism can be achieved.
- *Economic reasons.* It normally costs much less to put together a system of smaller computers with the equivalent power of a single big machine due to the advance of workstations and PCs. In the meantime the communication cost can be reduced when distributed databases are implemented. If databases are geographically dispersed and the applications accessing them are at the intersection of dispersed data, it will be much more economical to partition the relations and the applications so that data processing can be done locally at each site [26, 93].

The above list shows many reasons for developing distributed databases. The full benefit of improved performance and reduced communication overheads can only be achieved by proper database distribution design.

Database distribution design was first discussed in the context of the relational data model (RDM), then in the object-oriented data model (OODM). With computer-based systems penetrating all areas, there are increased demands for the non-conventional applications, such as computer-aided design (CAD), geographic information systems, image and graphic database

systems, etc., and distributed complex database systems are required to model complex valued data required at different locations. Also, with the current popularity of web information systems, there are increasing needs for distributed database systems to provide back-end support for web-based database applications. In particular, this applies to non-relational database systems such as object-oriented databases [103], object-relational databases [109] or databases that are based on the eXtensible Markup Language (XML) [1], which are used more and more for advanced database applications. This leads to the challenge of database distribution design for complex value databases, which covers the common aspects of object-oriented databases, object-relational databases and XML. The focus of this thesis is to investigate distribution design techniques with the aim of improving system performance for complex valued databases.

1.2 Database Distribution Design

1.2.1 Design Techniques: Fragmentation and Allocation

Distribution design is one of the major research problems whose solution is supposed to enhance performance of a distributed database. It involves data acquisition, fragmentation of databases, allocation and replication of the fragments, and local optimization. Fragmentation and allocation are the most important elements of a distributed database design phase. They play important roles in the development of a cost efficient system [93].

Fragmentation is a design technique to divide a single database into two or more partitions such that the combination of the partitions yields the original database without any loss or addition of information [96]. This reduces the amount of irrelevant data accessed by the application, thus reducing the number of disk accesses. The result of the fragmentation process is a set of fragments defined by a fragmentation schema. Fragmentation in the RDM can be either horizontal, vertical or mixed.

Horizontal fragmentation partitions a relation or a class into disjoint unions (fragments), which will have exactly the same structure but different contents. Thus a horizontal fragment of a relation or class contains a subset of the whole relation or class instance. *Vertical fragmentation* results in attributes and methods being partitioned into different fragments and therefore reduces irrelevant data accessed by local applications [102].

Allocation is the process of assigning a node on the network to each fragment after the database has been properly fragmented [93]. When data are allocated, it may either be replicated or maintained as a single copy. The replication of fragments will improve the reliability and efficiency of read-only queries. The intention of allocation is to minimize the data transfer cost and the number of messages needed to process a given set of applications, so that the system functions effectively and efficiently [62, 93, 107]. This thesis will not consider replication. Once a non-redundant allocation schema is obtained, some approaches can be applied to replicate fragments in a distributed database system [64, 93].

The purpose of fragmentation design is to determine non-overlapping fragments which could be the logical unit of allocation [26]. The individual tuple or attribute of a relation cannot be considered as the unit of allocation because the allocation problem would become unmanageable. The fragments are constituted by grouping tuples or attributes that have the same "properties" from the viewpoint of their application [26]. This is based on the idea that two elements in the same fragment that have the same "properties" will be accessed by the

applications together. Therefore, the fragments obtained in this way are the appropriate units of allocation [26]. The reasons for fragmenting databases are discussed in [11, 47, 93]:

- Applications are usually based on the views of subsets of relations. Thus the applications often access any subset of an entire relation locally. Therefore, fragmentation can reduce irrelevant data accesses and increase data local availability;
- If there is a relation on which many application views are defined at different sites, storing a given relation at one site will result in an unnecessarily high volume of remote data accesses. Storing a given relation at different sites will cause problems in executing updates and may not be desirable if storage is limited;
- The decomposition of a relation into fragments permits many transactions to be executed concurrently and results in the parallel execution of a single query by dividing it into a set of sub-queries that operate on fragments.

However, on the other hand, fragmentation may cause the following problems [93]:

- Applications whose views are defined on more than one fragment may suffer performance degradation when the relations are not partitioned into mutually exclusive fragments.
- When the attributes participating in a dependency of a relation are decomposed into different fragments and stored at different sites, the task of checking for dependencies would result in chasing after data in a number of sites.

1.2.2 Alternative Design Strategies

The design of a distributed database system involves making decisions on the architecture of DDBMS. Two major strategies proposed by Ceri and Pelagatti [26] for designing distributed databases are: top-down approach and bottom-up approach. In the case of tightly integrated distributed databases, design proceeds top-down from requirements analysis and logical design of the global database to physical design of each local database. In the case of distributed multidatabase systems, the design process is bottom-up and involves the integration of existing databases. But real applications are rarely simple enough to fit nicely in either of these approaches. The two approaches may need to be applied together to complement each other [93].

Top-down Approach. In the top-down approach, the process starts with a requirement analysis that defines the environment of the system and elicits both the data and processing needs of all potential database users [113]. The requirements analysis also specifies where the final system is expected to stand with respect to the objectives of the DDBMS. The objective is defined with respect to performance, reliability and availability, economics, and expandability (flexibility).

The requirements documents are the inputs to two parallel activities: view design and conceptual design. The outputs of view design are the interfaces for the user, and the outputs of conceptual design are entity types and relationship types which are used to construct an external schema.

In a distributed database environment, the objective of distribution design is to design the local conceptual schemata by distributing the fragments. The fundamental issues in top-down design are fragmentation and allocation [93].

The last step in the design process is the physical design, during which local conceptual schemas are mapped to the physical storage devices available at the corresponding local sites.

Bottom-up Approach. Ceri and Pelagatti [26] and Özsu and Valduriez [93] stated that top-down design is suitable for the systems which are developed from scratch. But when the distributed database is developed as the aggregation of existing databases, it is not easy to follow the top-down approach. The bottom-up approach, which starts with individual local conceptual schemata, is more suitable for this environment [26, 93]. Ceri and Pelagatti [26] explained that the bottom-up approach is based on the integration of existing schemata into a single, global schema. Integration is the process of the merging of common data definitions and the resolution of conflicts among different representations that are given to the same data. The global conceptual schema is the product of the process [93]. Ceri and Pelagatti [26] concluded that there are three requirements for bottom-up design:

1. the selection of a common database model for describing the global schema of the database,
2. the translation of each local schema into the common data model, and
3. the integration of the local schemata into a common global schema.

In this thesis the focus is on top-down approach which aims at developing fragmentation and allocation schema of complex value databases.

1.2.3 The Objective of the Design of Data Distribution

Several objectives that should be taken into account in the design of distribution are presented in [26]:

- In a distributed database system one of the major costs is associated with communication. To minimize communication costs, one goal of DDBMSs is to achieve processing applications locally. The degree of local processing can be maximized by distributing data, therefore minimizing transaction costs. To achieve this goal, the data should be kept as close as possible to the applications which use them. The advantage of processing applications locally is not only the reduction of remote access costs, but also increased simplicity in controlling the execution of the application.
- The availability and fault-tolerance of read-only applications can be improved by storing multiple copies of the same information at different sites. When one site of the database is down or the community link for that site is broken, the system can still execute the applications by accessing the other copies of the information.
- Distributing workload over the sites is done in order to take advantage of the different powers of utilization of the computers at each site, and to maximize the degree of parallelism of execution of applications. But the trade-off between processing locally and distributing workloads should be considered in the designing of data distribution.
- Database distribution should reflect the cost and availability of storage at each site. Even though the storage cost is not relevant when compared with the cost of input or output (I/O), central processing unit (CPU), and transmission costs of the applications, the limitation of available storage at each site should be considered.

During the design process of fragmentation and allocation, minimizing communication costs is the main objective. With the advance of current computer power, storage cost is not a big concern any more. The other two objectives, improving availability and fault-tolerance and distributing workload, can be achieved when databases are fragmented and distributed properly among the network.

1.3 Distribution Design Dilemma

Fragmentation and allocation for distributed databases are the key issues of database distribution design. The research in this area often involves design methods (e.g. mathematical programming) in order to minimize the combined cost of storing the database, processing transactions against it, and communication [93]. It is practically impossible to study database distribution design together with other problems because each of the problems is difficult enough to be studied by itself.

1.3.1 The Complexity of the Problems

The combined problem of fragmentation and allocation is proven NP-hard [18]. Both fragmentation and allocation are distribution design techniques used to improve system performance. Each of the problems has an immense search space for optimal solutions.

In the case of horizontal fragmentation, if n simple predicates are considered to perform primary horizontal fragmentation, 2^n is the number of horizontal fragments using minterm predicates. If there are k network nodes, the complexity of allocating horizontal fragments is $O(k^{2^n})$. For example, using 6 simple predicates to perform horizontal fragmentation results in $2^6 = 64$ fragments. To find the optimal allocation of the fragments one needs to compare all the $4^{64} \approx 10^{39}$ possible allocations. This is practically incomputable with the power of current computers.

In the case of vertical fragmentation, if a relation has m non-primary key attributes, the possible fragments are given by the *Bell* number which is approximately $B(m) \approx m^m$. With this number of possible fragments, the fragment allocation using a suitable cost model is of the complexity $O(k^{m^m})$, with k as the number of network nodes. Heuristic approaches are proposed in the literature for vertical fragmentation to reduce the complexity. For example, the affinity-based approach that uses an objective function proposed in [88] is of complexity $O(m^2 \cdot \log m)$, while graphical approach proposed in [89] is of complexity $O(m^2)$.

It is computationally infeasible to use a cost model to evaluate all possible fragmentation schemata resulting from using minterm predicates or all possible vertical fragmentation schemata. Indeed, using affinity to group attributes or predicates can reduce the number of fragments in the resulting fragmentation schema. However, due to the problems of affinity-based approaches for both horizontal and vertical fragmentation, as shown in Chapter 7, the possibility of obtaining optimized distribution design on the step of allocation is reduced.

To evaluate system performance, distributed query trees should be considered. Allocation of intermediate nodes should be coupled with the allocation of leaves, which may be fragments. To improve overall system performance, we need to allocate intermediate nodes of all the queries that are taken into consideration. This further increases the complexity of allocation. Assume the average number of intermediate nodes of the input queries is h, then the allocation space is increased by $O(k^{q \cdot h})$, with q as the number of queries.

Due to the complexity of both fragmentation and allocation problems, allocation is often treated independently from fragmentation. In the literature, most of the allocation approaches assume fragmentation has been done already. The output of fragmentation is input to allocation. The motivation to isolate fragmentation from allocation is to simplify the formulation of the problem by reducing the decision space. However, the isolation actually contributes to the complexity of allocation models. Both steps take user applications as input information and aim at improving system performance; they differ only in that fragmentation works on global

database schema while allocation works on fragments. Therefore, the application information and relationship between fragments need to be specified again while doing allocation. It would be reasonable to develop a methodology to reflect the interdependence of fragmentation and allocation [27, 92].

1.3.2 Interdependencies with Query Optimization

Designing distributed database systems is a fairly complex task as several other issues are also involved, including query processing and optimization, data replication, concurrency control, directory management, reliability, and recovery. Among these problems, query processing and optimization is a closely interrelated problem with fragmentation and allocation. Distributed query optimization depends on how data are fragmented and allocated, since query processing schedules define the sequence of operations of queries and the allocations of the operations according to the allocation of fragments. On the other hand, optimal fragmentation and allocation of data depends on query processing strategies used to execute queries.

An optimal database distribution design should incorporate query information, which can be represented using query trees. After fragmentation, query trees need to be adjusted to include only those fragments that are needed for processing the queries [93]. The consideration of distributed query processing makes the allocation of fragments even more complex. The allocation of intermediate nodes is interrelated to the allocation of fragments. If a query has h intermediate nodes then the complexity will be $O(k^{2^n} \cdot k^h)$ for the case of primary horizontal fragmentation, and $O(k^{m^m} \cdot k^h)$ for the case of vertical fragmentation. To obtain an optimal design of a distributed database we need to consider total query costs of the most frequently issued and important queries, usually taking 20% most frequently queries as heuristics. Then the complexity will increase to $O(k^{m^m} \cdot k^{h \cdot q})$ for the case of vertical fragmentation, with q as the number of queries considered.

In summary it can be concluded that:

- It is infeasible to get real optimized distribution design because of the number of possible fragments and the search space of optimal allocation of fragments.
- The optimization of fragmentation and allocation has to be homogenized with query optimization because query optimization requires knowledge about fragmentation and allocation while to get optimal fragmentation and allocation requires knowledge of queries.
- In addition, it is desirable to achieve stability of fragmentation and allocation. This requires that small changes in input information should not affect the solution of fragmentation and allocation.

1.3.3 Ad Hoc Solutions

In the literature, to reduce the complexity of the problem and to increase the problem tractability researchers often employ the following approaches:

- Fragmentation and allocation are often treated separately as two different steps. Fragmentation is performed first without considering how resulting fragments will be allocated; while allocation is performed with the assumption that fragmentation has been decided already. Allocation is studied with the assumption that a fixed query optimization method is used to generate processing schedule; while the study of query optimization is conducted with an assumption of fixed data allocation [18].

- Either simple query environment or query site strategy is often assumed while studying allocation. With the first assumption, network information is not considered. With the second assumption, queries are not considered to be processed in a distributed way. Therefore, query trees are not employed and allocation of intermediate nodes is not considered.
- Query optimization is disregarded while studying data allocation. A real fragment allocation can only be achieved when distributed query optimization is performed after fragmentation.

While some ad hoc solutions are proposed in the literature to lead to effective solutions to parts of the overall system design, ignorance of interdependencies between individual problems makes this approach inefficient in the sense of obtaining optimal database distribution design.

1.4 Contributions

The objective of this thesis is to generalize distribution design to complex value data models, i.e. capture complex value, object-oriented and XML-like databases [1, 103, 109], with the consideration of distributed query optimization such that a performance increase by distribution can be really achieved. The goal is to define fragments and allocate them in a way such that the overall query processing costs are minimized. This goal can be approached in two ways. The first one, which will only be sketched briefly, maps databases and queries to a relational model and then exploits known results for the relational model [74]. It is doubted that this approach is applicable in general, as the relational storage of complex value databases may not always be the best idea. The second approach uses directly the complex value data model and the query trees defined for it. Both approaches lead to similar optimization problems.

In this thesis we consider constructors for sets and lists as fundamental constituents of complex value data models. By studying complex value data models on the basis of a type system one does not need to deal with individual data models but can focus on the impact of particular type constructors on the database research problem under investigation. We will discuss query languages, a cost model for estimating the costs of query processing, and fragmentation techniques in the presence of the set and the list constructor. Firstly, queries can be formalized using a generic query algebra that incorporates theses constructors. Queries formulated in such an algebra give rise to query trees [72, 93] which form the basis of a query processing cost model in the relational data model. We extend such a cost model to cover size calculations for complex value data. Major activities of query optimization can be regarded as tree manipulations. These include algebraic, tableaux and join order optimizations. We like to mention that queries given in other query languages can be mapped to queries over the chosen algebra. However, the question of how such a mapping can be achieved is beyond the scope of this thesis and left for further investigation. Finally, once such an algebra is at hand the generalization of common fragmentation operations from the relational data model is rather straightforward. As mentioned in [103], sets require more care with respect to vertical fragmentation as one typically requires the presence of an embedded key dependency or the introduction of an artificial key attribute (surrogate). In addition to horizontal and vertical fragmentation known from the relational data model we also study a splitting operation which basically replaces an embedded component by a reference, cf. [102].

The first major contribution of this thesis is a novel approach to discuss the optimization problem of fragmentation and allocation incorporating query information with the following strategies:

- Considering the most relevant queries with the assumption that optimized query trees and optimized allocation of intermediate results are given;
- Investigating necessary changes to the query trees in the case of one fragmentation operation to analyze new allocation of fragments, new allocation of query nodes after another round of query optimization;
- Analysis of the effects of fragmentation for complex value databases with the aim of searching for tractable solutions for the combined problem of fragmentation and allocation.

With the approach above the following observations, which are important to support the consideration of separating the two NP-hard problems, distributed query optimization and database distribution design, but have never been mentioned by previous work in the literature, will be shown.

- The allocation of intermediate nodes of query results in the case of running a query on a distributed database is orthogonal to the allocation problem for fragments, i.e. the decision regarding which network nodes should be assigned to the nodes in the query tree does not depend on the allocation of fragments.
- We can concentrate on simple sub-queries provided. As a consequence, we may assume optimized query trees with optimal assignment of nodes. Using these trees the effects of fragmentation on these query trees can be investigated.
- The data models that the fragmentation and allocation are studied for mainly impact on sizes of leaves and intermediate nodes. That means there is no fundamental difference if fragmentation and allocation are studied in the context of complex value data models rather than in the context of the RDM.

The second major contribution of the thesis is a discussion of a heuristic approach to the optimization problem of fragmentation and allocation. This heuristic approach includes a set of algorithms, listed below, for fragmentation and allocation of complex value databases. To set a framework for studying fragmentation in the context of complex value data models, formal definitions and correctness criteria of fragmentation operations are presented to cover complex data types.

- An algorithm to reduce the number of selection predicates that are needed to perform primary horizontal fragmentation;
- A cost-based algorithm for primary horizontal fragmentation, which produces a reasonable number of fragments that meet the requirement of data local availability;
- A cost-based algorithm for derived horizontal fragmentation, which can be performed either on an owner database type or a member database type;
- A cost-based algorithm for vertical fragmentation, which incorporates query information;

At the end of the thesis, in the conclusion, several open problems for future research are identified.

1.5 The Outline of the Thesis

In this chapter fundamental problems of database distribution design for complex value databases have been addressed. The remainder of this thesis is organized as follows.

Chapter 2 will give an overview of related work on database distribution design. After a state-of-the-art summary on familiar distribution techniques such as horizontal and vertical fragmentation, and allocation, several deficiencies of existing approaches are highlighted. The motivation for the research undertaken by the author and reported on in this thesis is then to overcome some of these deficiencies in the context of complex value data models.

In Chapter 3, fundamentals of complex value data will be discussed. For this, a type system will be introduced that allows the recursive application of record, set and list constructors to a collection of base types. In addition, a query algebra, query trees and a query processing cost model for complex value data will be presented.

Chapter 4 will concentrate on fragmentation techniques for complex value databases. Splitting, horizontal and vertical fragmentation will be defined, along with application rules for fragmentation operations.

In Chapter 5, the effects of fragmentation on query trees will be investigated for given queries that have been optimized before fragmentation. For each of the three fragmentation techniques (splitting, horizontal and vertical fragmentation) different scenarios will be discussed.

Chapter 6 will present a cost-based approach for horizontal fragmentation. Based on the cost model from Chapter 3, algorithms for primary and derived horizontal fragmentation will be provided.

Chapter 7 will focus on a cost-based approach for vertical fragmentation. Based on the cost model from Chapter 3 algorithms for vertical fragmentation will be presented that incorporate relevant query information, including information on the frequency of queries and the locations where the queries are issued.

Finally, Chapter 8 will provide conclusions of this work, and discuss open problems for future research.

Chapter 2

Literature Review on Distribution Design for Databases

Fragmentation and allocation are two main database distribution design techniques. Since the 1970s database distribution problem has been studied, first as the problem of file distribution, then as the problem of distributing relations or relation fragments in the context of the relational data model. With the emergence of the object-oriented data model, the existing approaches of fragmentation and allocation have been adopted to the object-oriented data model by taking into consideration the features of the object-oriented data model. In recent years, with the popularity of web information systems, which often have backbone databases with XML as the database model, researchers have started to study fragmentation and allocation for XML but with little work done in the literature [6, 19]. To get a whole picture of the state-of-the-art of database distribution design, in this chapter I present an overview of previous work in database distribution design: horizontal fragmentation, vertical fragmentation, and allocation. For each of the design techniques, we will see how it is developed while the flavor of data models changes with time.

2.1 Horizontal Fragmentation

There are two types of horizontal fragmentation, primary and derived. Primary horizontal fragmentation of a relation or a class is performed using predicates of queries accessing this relation or class, while derived horizontal fragmentation of a relation or a class is performed based on horizontal fragmentation of another relation or class.

2.1.1 Primary Horizontal Fragmentation for Relational Databases

In the context of the relational data model, existing approaches for horizontal fragmentation mainly fall into following two categories:

- minterm-predicate-based approaches: which perform primary horizontal fragmentation using a set of minterm predicates, e.g., [25, 45, 93].
- affinity-based approaches: which first group predicates according to predicate affinities and then perform primary horizontal fragmentation using conjunctions of the grouped predicates, e.g., [16, 31, 94, 114]. The way of grouping predicates is either graph-based or using an objective function.

For the ease of reviewing related works in the literature, some terms that often occur in the literature are listed below [25, 26, 93] .

Definition 2.1. [93] For a given relation $R = \{A_1 : D_1, \ldots, A_n : D_n\}$, a *simple predicate* is in the form of

$$p_k : A_i \;\; \theta \;\; Value$$

with A_i as an attribute defined over domain D_i, $\theta \in \{=, <, \neq, \leq, >, \geq\}$ and $Value \in D_i$. A set of all simple predicates defined on relation R is denoted by $Pr = \{p_1, p_2, \ldots, p_m\}$. □

Definition 2.2. [93] *Minterm predicates* $M = \{m_1, m_2, \ldots, m_z\}$ over a set Pr of simple predicates are the conjunctions of simple predicates and their negations:

$$M = \{m_j | m_j = \bigwedge_{p_k \in Pr} p_k^*\}, \quad k = 1, \ldots, m, j = 1, \ldots, z.$$

where $p_k^* = p_k$ or $p_k^* = \neg p_k$. Note that all simple predicates in Pr appear (positively or negatively) in each minterm predicate. □

Definition 2.3. [93] A set of simple predicates Pr is said to be *complete* if and only if there is an equal probability of access by every application to any tuple belonging to any fragment that is defined according to Pr. □

Definition 2.4. [25, 93] Let m_i contain p_i, and let m_j be obtained from m_i by replacing p_i by $\neg p_i$. Let F_i and F_j be fragments defined according to m_i and m_j, respectively. Then p_i is *relevant* if and only if

$$\frac{acc(m_i)}{card(F_i)} \neq \frac{acc(m_j)}{card(F_j)}$$

where $acc(m_i)$ and $acc(m_j)$ denote the access frequencies of minterm predicate m_i and m_j, $card(F_i)$ and $card(F_j)$ denote the cardinalities of fragment F_i and F_j.
If all the predicates of a set Pr are relevant, then Pr is *minimal.* □

Definition 2.5. The *predicate affinity* between each pair of simple predicates p_i, p_j is the sum of frequencies of the queries accessing both predicates together. □

Using minterm predicates to perform horizontal fragmentation was first proposed in [25] to fragment file horizontally to optimize the number of accesses performed by the application programs to different portions of data. They state that the minterm fragments contain records that are homogeneously accessed by all transactions and therefore are the proper units of allocation. Ceri and Pelagatti [26] then proposed to use minterm predicates to fragment relations. To perform *primary horizontal fragmentation*, a set of disjoint and complete selection predicates should be determined. Firstly, using application information derives simple predicates $P = \{p_1, \ldots, p_n\}$, which should satisfy *complete* and *minimal* properties, in order to guarantee that elements in the same fragments share the "same properties" in terms of allocation. Then a set of *minterm predicates* are constructed from P. A set I of implications among the p_i^* can be used to determine (and remove) these unsatisfiable minterms. Note that to test the completeness of the set of simple predicates, the probabilities of access by applications need to be compared. However, often the size of the set of simple predicates is big, and the cost of computation might be too expensive. Also, if resulting minterm fragments of a predicate

are relevant and accessed differently by queries at the same site, they may still be allocated at the same site. That is, the fragmentation is not necessary and the predicate is not needed for fragmentation.

Follow the line of [25], Özsu and Valduriez [93] first presented an iterative algorithm, COM_MIN, to generate a complete and minimal set of predicates Pr' from a given set of simple predicates Pr. The algorithm checks each predicate p_i in the given set of simple predicates Pr to see if it can be used to partition the relation R into at least two parts which are accessed differently by at least one application. If p_i satisfies the fundamental rule of completeness and minimality then it should be included in Pr'. If p_i is nonrelevant then it should be removed from Pr'. But this algorithm is not practical because checking p_i can not be defined with machine readable language. An algorithm named PHORIZONTAL is introduced to describe primary horizontal fragmentation. It uses the algorithm COM_MIN and a set of implications I as inputs to produce a set of satisfiable miniterm predicates M. If a minterm predicate m_i is contradictory to an implication rule in I, then it is removed from M. Minterm fragments are defined according to the set of satisfiable minterm predicates M. But the set I of implications is hard to define.

In fact, the algorithm is not very practical, as it will always result in a subset Pr' of Pr, the set of minterm predicates M' determined by Pr' and the corresponding set of fragments. Simple predicates are omitted from Pr if they do not contribute to the fragmentation, i.e. they only violate the minimality principle. This results in considering just the simple predicates $A_i \theta v_i$ in the most important queries and to take all satisfiable minterm predicates. This obviously leads to fragments that are accessed differently by at least two queries. The algorithm further does not give executable rules for eliminating the unsatisfiable minterm predicates. The major problem, however, is that the number of fragments resulting from the algorithm is exponential in the size of Pr. In practice, it would be important to reduce this number significantly, which would mean to re-combine some of the fragments. In fact, this implies giving up the completeness principle and replacing it by optimization criteria based on a cost model.

Several researchers have adopted affinity-based vertical fragmentation algorithms to horizontal fragmentation. Due to the complexity of checking completeness of the set of simple predicates used for horizontal fragmentation, Zhang [114] adopted an affinity-based vertical fragmentation approach to horizontal fragmentation. This approach takes predicate usage and predicate affinity matrix as input and employs the bond energy algorithm to cluster predicates. However, the fragments in the resulting fragmentation schema may overlap each other and therefore cannot satisfy the correctness criteria of fragmentation. Ra [94] presented a graph-based algorithm for horizontal fragmentation, with which predicates are clustered based on the predicate affinities. To remove overlapping, an adjust function is presented to merge two overlapped fragments if merging can reduce transaction costs using cost functions. However, the cost function does not show how costs are computed. Using predicate matrix as input, Cheng *et al.* [31] proposed a genetic algorithm-based clustering approach, which treats horizontal fragmentation as a traveling salesman problem (TSP). Horizontal fragmentation is achieved by performing selection operation using the set of the grouped predicates, which are grouped according to the distances. The distance of each pair of attributes actually measure the access frequencies of transactions that do not access the pair attributes together. Additional analysis is needed to simplify the clusters of predicates. Obviously, none of the affinity-based horizontal fragmentation approaches takes into consideration of data locality while clustering predicates.

There are other approaches in the literature besides the affinity-based and minterm-based approaches. Chang and Cheng [30] proposed a methodology of decomposition based on mapping user views onto a global schema. However, there is neither clear procedures for processing decomposition no evaluation of the result decomposition. Shin and Irani [104] proposed a knowledge-based approach in which user reference clusters are derived from the user queries to the database and the knowledge about the data. Their paper mainly emphasizes the extension of first order logic calculus without any procedure or algorithm on how to perform horizontal fragmentation procedurally. Also, the completeness of the knowledge base is a critical issue for the correctness of the final fragmentation. Shin and Irani [105] extended the work in [104] by presenting an example to illustrate how fragmentation can provide enhanced control over data replication and reduce costs on selection operations. However, the discussion is informal without based on any cost model. Considering the purpose of horizontal fragmentation is to minimize the total number of disk accesses, Khalil *et al.* [65] introduced a horizontal transaction-based partitioning algorithm, which takes as input a predicate usage matrix. However, the aim of fragmentation for database distribution design is to reduce not only the total number of disk accesses but also data transportation costs, which dominate the total query costs.

2.1.2 Primary Horizontal Fragmentation for Object Oriented Databases

Generalizations of horizontal fragmentation techniques to object-oriented databases (OODBs) have been proposed since the 1990s. Algorithms proposed in the literature are mainly cost-driven and affinity-based.

Karlapalem *et al.* [63] proposed to use predicate affinities to perform horizontal fragmentation. However, there is no detailed method on how to perform fragmentation. Following the line of [63], representation schemata for horizontal fragmentation were presented in [59, 61], followed by a solution for supporting method transparency in OODBs. Bellatreche *et al.* [15] studied horizontal class partitioning as a process of reducing the number of disk access to execute queries by reducing the number of irrelevant object access. Predicates are extended from that in the relational model by considering predicates defined on methods. Predicates are clustered according to the extended tree from the predicate affinity matrix. Some clusters of predicates are extended to include some predicates defined on the attributes that are not used by the cluster. The resulting set of clusters of predicates are of the same length in terms of number of simple predicates and are defined on the same set of attributes or methods.

Bellatreche *et al.* [12] stated that the effect of horizontal fragmentation should be measured by evaluating the performance of the applications in a distributed database system. Cost-Driven Algorithm is presented to find a scheme that lead to the lowest total query cost based on a cost model. However, in the cost model CPU costs and network communication costs are disregarded because only centralized databases are considered. Therefore it cannot be applied to distributed databases, where network communication cost predominant total costs. Moreover, complexity of the cost-driven horizontal partitioning algorithm is too high and makes the algorithm computationally intractable.

Bellatreche *et al.* [16] have studied horizontal class partitioning with input as queries which contain either simple and component predicates, the primary algorithm (PA) is based on a graph theoretic algorithm which clusters a set of predicates into a set of HCFs. The complexity of this algorithm is $O[l * n^2 + \alpha(r + u)]$, where n, l, α, r, u represent the number

of predicates, the number of queries, the number of HCFs, the number of attributes used by the queries, and the number of methods used by queries, respectively.

A taxonomy of the fragmentation problem in a distributed object based system is presented in [44] to include four realizable class models, simple attributes and methods, complex attributes and simple methods, simple attributes and complex methods and complex attributes and methods. For one of these class models, simple attributes and methods, a set of detailed horizontal fragmentation algorithms are proposed. Continuing the work in [44], Ezeife and Barker [45] presented a comprehensive set of algorithms for horizontally fragmenting over all the four realizable class models following [93]. Ezeife and Zheng [43] have proposed an Object Horizontal Partition Evaluator (OHPE), which contains two components, the local irrelevant access cost and the remote relevant access cost. However, both components only measure the number of instances of a fragment without taking into consideration of size of the object and network information. A class is fragmented using all predicates from the queries accessing the class directly, predicates of all queries of all the descendants of the class that access the class, and predicates of all its containing classes accessing the class, and predicates of all its complex method classes. An example is presented to show how to compute the performance of the object horizontal fragmentation schemata with proposed OHPE. However, it is not shown how the horizontal fragmentation schemata are achieved and how fragments are allocated. An algorithm is proposed to re-fragment the class once input information is changed, including the user query access pattern and frequencies, changes in class hierarchy, change in class composition hierarchy, and change in the instance objects of classes.

Nwosu [91] discussed distribution design technique on complex objects and focuses on method induced partitioning, which considers the impact of method behavior on partitioning schemes for object-oriented databases. Based on types of instance variables accessed, methods are first classified as value-based or object-based. Then a set of partitioning schemes for each type of the methods are enumerated. Some implementation and design issues regarding how to support method induced partitioning of object-oriented databases are addressed, e.g., how to support fragmentation transparency. However, there is no discussion on how horizontal fragmentation is performed if transaction information and other input information are given. From our point of view, the definition of different kinds of fragmentation can be more precise. Baiõo *et al.* [9, 10] adopted the algorithm proposed in [87] and take predicate affinity matrix as input to build a predicate affinity graph that is used to define horizontal class fragments. Again, the resulting horizontal fragmentation schema only reflects the togetherness of data accessed by transactions or queries but cannot reflect the affinities between data and network sites, that is, data locality. Also taking predicate affinity as input, Cheng *et al.* [31] performed horizontal fragmentation by clustering predicates according to the distance between predicates formulated as a traveling salesman problem (TSP).

2.1.3 Derived Horizontal Fragmentation for Relational Databases

Derived fragmentation in the RDM refers to horizontal fragmentation defined on a member relation r of a link according to fragmentation of one of its owner relations s [24, 93]. Derived horizontal fragmentation can be performed by applying semijoin operations.

$$r_i = r \ltimes s_i, 1 \leq i \leq w$$

where w is the number of horizontal fragments of relation s, and $s_i = \sigma_{\varphi_i}(s)$ with φ_i as the selection predicate.

In [24], a *link* among relations is introduced to depict the binary relationship between relations. A direct link is drawn between relations that are related to each other by an equijoin operation. The direction of a link shows a one-to-many relationship. It is assumed that the join attributes for a link include the primary key of the owner of the link. Note that, in our complex value data model an owner type is actually a component of a member type.

In [93] it is emphasized that care should be taken with the relations that have more than one link to the owner relations. The two criteria in [93] suggest choosing the fragmentation with better join characteristics or choosing the fragmentation used in more applications. However, there are often situations for which it is not straightforward to choose an owner relation using the two criteria. Further, how to deal with owner relations if their member relations have been horizontally fragmented using predicates but there is no predicate defined on the owner relations is not discussed. Furthermore, there is no explanation why derived horizontal fragmentation can only be performed on a member relation according to fragmentation of one of its owner relations.

2.1.4 Derived Horizontal Fragmentation for Object Oriented Databases

Unlike the relational model situation the definition of derived horizontal fragmentation is not straightforward in the object-oriented data model. In the literature different definitions have been presented:

- In [44, 45], derived horizontal fragmentation refers to fragmentation on member classes of links according to the fragmentation of its owner class, where subclasses are owner classes of a superclasses.
- In [12, 15, 62], derived horizontal fragmentation refers to fragmentation of a non-leaf class fragmented on fragmentation of a leaf classes.
- In [59], derived horizontal fragmentation is defined as a horizontal fragmentation of the domain class of an object based instance variable based on horizontal partitioning of a value based instance variable. In addition, associated horizontal partitioning is introduced as horizontally partitioning a class based on the class hierarchy of the domain class of an object based instance variable of the class.
- In [9] owner and member relationships are defined based on paths that an operation navigates through, where a member class is always defined at the "1" side of the relationship link. Owner and member relationship is not defined for many to many relationship.

In [15], derived horizontal fragmentation of a class is performed using component predicates that are defined with path expressions. This may result a set of overlapped fragments. The last step is then to remove overlap between fragments according to the sum of the frequency of accesses of the fragments. The overlapped objects are removed from the fragments that are accessed less frequently. However, it is not necessary to distinguish between simple attributes or complicated attributes. Similarly, it is not necessary to distinguish simple predicates and component predicates. The derived fragmentation is defined as using component predicates, the sink of which is an attribute of another class. The proposed algorithm uses logical connectives $(\vee, \wedge)$ but does not mention when each connective should be used. Also, a predicate defined on a path does not always mean that the predicate has a sink as an attribute of another class.

In [16], derived horizontal partitioning of a class C_i is performed based on the qualification clauses of class C_j that has been primarily partitioned into α horizontal class fragments

(HCFs), each of which is defined by a qualification clause cl_j. When the fan-out between C_i and C_j is bigger than 1, an extra algorithm is applied to remove the overlapping. The fragments that are accessed more frequently keep the overlapping instances. Two potential complications, multiple primary horizontal partitioning candidates and multiple derived horizontal partitioning candidates, are discussed. Multiple primary horizontal partitioning refers to a class being partitioned based on another one of the classes, which is already horizontally partitioned, at lower levels in a class composition hierarchy. Four ways of selecting possible candidates are proposed, as below:

1. share candidate: The share candidate approach selects a candidate C_j which has the minimum share with C_i.
2. frequency candidate: The frequency candidate approach selects the member class that has the maximum frequency of the queries that access it.
3. number of HCFs candidate: The number of fragments solution approach selects a candidate that has the smallest number of HCFs.
4. fusion candidate: The fusion solution approach perform derived horizontal partitioning based on all candidates and then merging all HCFs that are accessed together by some queries.

When a class C_j is partitioned and there is at least one path containing more than one class, from a root class C_i to C_j, multiple derived horizontal fragment candidates try to select a candidate class to be derived fragmented to achieve the least total query costs. To locate fragments needed by a query, the routing algorithm is proposed. A routing module can be added in a query processing methodology for the horizontally partitioned classes. Once object instances are updated, some objects may migrated from one HCF to another HCF. For this an object migration algorithm is proposed.

However, regarding multiple derived horizontal fragment candidates, it should be possible to have more than one class along the path to be fragmented to get the minimum query cost. Also, regarding multiple primary horizontal partitioning candidates, it is proposed to use the cost model to choose one that has the minimum cost from the four candidates resulted from the four proposed approaches. However, the complexity of doing this can be very high due to the complexity of the fusion approach. The cost model proposed in [12, 13] measures the number of disk page accesses. Transportation cost, which measures the cost of network communication among different nodes, is not considered at all because databases considered are centralized.

In [45], derived horizontal fragments of a class are generated according to primary fragments of its subclasses, its complex attributes (contained classes), and/or its complex methods. Heuristics are proposed to choose the most appropriate primary fragment to merge with each derived fragment of the member class. At last, derived fragments are merged with a primary fragment that has the highest affinity with it. However, this approach leads to overlaps between resulting derived fragments. Inheritance links are considered in the process of horizontal fragmentation. It is assumed that a pointer is contained in an instance of a storage structure for a class in the class hierarchy. However, storage structures are not implemented in most object DBMS products. How to choose a owner class for a member class if the member class have several owner classes is not mentioned. Further, it is not clear how to calculate affinities between each pair of primary fragment and derived fragment of the same class. Furthermore, there is no evaluation of the proposed algorithms regarding how it will improve the system performance.

In [10], derived horizontal fragmentation of each member class is performed according to its owner class in *frag(owner, member)* list, which is based on the owner-member classification. Derived horizontal fragmentation is implemented with a semijoin on the attribute used by the most frequent navigation operations from the member class to the owner class. However, it is not clear how to decide the owner classes to be used for fragmentation. The resulting distributed database schema is analyzed to show improvements in system performance. However, the analysis neither considers queries as distributed queries nor uses any cost models. Moreover, in practice, objects or object variables are often accessed by a big number of queries. There is no algorithm presented to incorporate query information to achieve proper fragmentation schemata.

2.2 Vertical Fragmentation

Vertical fragmentation can be applied to different areas: file partitioning in centralized environment, data distribution among different levels of memory hierarchies of a database, and data distribution in distributed databases. For applications accessing fragments on different memory levels, the costs are dominated by the cost of retrieving data from secondary storage to main memory while for distributed databases, query costs are dominated by remote data transportation costs. We need to be aware that vertical fragmentation techniques for physical database design with different memory levels cannot be applied to distribute databases in a straightforward manner. The following reviews the work done regarding vertical fragmentation for relational databases.

2.2.1 Vertical Fragmentation for Relational Databases

Vertical fragmentation has been studied since 1970s. There are two main approaches:

- The pure affinity-based approach takes attribute affinities as the measure of togetherness of attributes to fragment attributes of a relation schema. Research work includes [31, 57, 69, 70, 86, 88, 89, 93].
- The cost-driven approach uses a cost model while partitioning attributes of a relation schema. Research work includes [28, 33, 38, 41, 106].

Definition 2.6. For a given pair of attributes a_i, a_j of a relation schema $R = \{a_1, \dots, a_n\}$ that is accessed by a set of queries $Q = \{q_1, \dots, q_q\}$ affinity is defined as

$$aff_h(a_i, a_j) = \sum_{k=1 | use(q_k, a_i)=1 \wedge use(q_k, a_j)=1}^{q} acc_k$$

with acc_k as the total accesses of a query q_k and $use(q_k, a_i) = 1$ when query q_k accesses attribute a_i. Computing affinities for each pair of attributes results an $n \times n$ matrix, called *attribute affinity matrix* [88]. □

Affinity-Based Approach Affinity-based vertical fragmentation was first proposed by Hoffer and Severance [57], who used Bond Energy Algorithm (BEA) to cluster attributes according to the affinities between attributes. Since then the affinity measure has been widely used

for solving the problem of fragmentation. Navathe *et al.*[88] extended the BEA approach in [57] by proposing algorithms that produce non-overlapping fragments and overlapping fragments. This approach minimizes the number of fragments accessed by transactions while considering storage cost factors involved in storing the fragments. This approach consists of two steps:

1. In the first step the given input parameters are used in the form of an attribute usage matrix (AUM) to construct an attribute affinity matrix (AAM) on which clustering is performed.
2. In the second step estimated cost factors, which reflect the physical environment of fragment storage, are considered to further refine the partitioning schema.

This paper [88] discusses vertical partitioning problem in three contexts: a database stored on devices of a single type, a database stored in different memory levels, and distributed database. Allocation of fragments in distributed databases targets at maximizing the amount of local transaction processing. At the first stage, the same objective function are used for both contexts, single site one memory level, and single site with multiple memory levels. The objective function for distributed databases is designed with the consideration of the ratio of the transaction volume satisfied by the upper block to the total transaction volume and the ratio of the size of the fragment defined by upper block to the size of the object. At the second step, an objective function is presented to include cost factors, each of which is of different weight in different contexts. However, there is no justification of the values of the factors. Also, the transportation cost factor is fixed for all transactions between any pair of network nodes.

Navathe and Ra [89] improved the previous work on vertical partitioning by proposing a vertical partitioning algorithm using a graphical technique. The major feature of this graphical approach is that all fragments are generated by one iteration in a time of $O(n^2)$, which is better than $O(n^2 \log n)$, the complexity of vertical partitioning algorithm in [88]. In the meantime, there is no need of an arbitrary empirical objective function for the algorithm of partitioning. This graphical approach starts with an attribute affinity matrix, based on which, an affinity graph is constructed, then a linearly connected spanning tree is formed. Affinity cycles, which are the candidate partitions, are constructed simultaneously with the growing of the spanning tree. Partitions are made according to the weight of the edges comparing with the weight of each edge of candidate cycles. The output of the algorithm is a set of vertical partitions of a given relation. However, the resulted number of fragments is not related to the number of nodes over a distributed network. If the resulted number of fragments is bigger than the number of network nodes, fragment recombination needs to be performed. In addition, there is no evaluation of goodness of the resulting vertical fragmentation schema as to how it will improve the distributed database system performance.

Lin and Zhang [70] pointed out that the restriction of an affinity cycle results in formalization is an NP-hard problem and therefore the claimed properties in [89] cannot be guaranteed. A new graphic algorithm is proposed by using 2-connectivity instead of affinity cycle to construct nonoverlapping fragments, which is later allocated to distributed network nodes. Lin *et al.* [69] continued the work in [70] by presenting an algorithm for the construction of overlapping fragments. Cheng *et al.* [31] formulated the database partitioning problem in distributed databases as a traveling salesman problem (TSP), for which a genetic search-based clustering approach is proposed to achieve high system performance. Again, this approach is affinity-based, which clusters attribute according to the distance among attributes. Site information

is not taken into account while performing fragmentation. The objective of this approach is to group attributes such that the difference between the average distance within groups and the average distance between groups is minimized. However, there is no proof that this approach can indeed minimize the total query costs.

Cost-Driven Approach Cost-driven vertical fragmentation approaches use cost functions. Cornell and Yu [38] and Chu [33] discussed vertical fragmentation for relational databases and considered that the response time of transactions is impacted by the number of disk accesses by the transaction. Considering the utility of vertical fragmentation is to minimize the number of disk accesses, Cornell and Yu [38, 41] proposed a two step methodology that consists of a query analysis step to estimate the parameters and a binary partitioning step that can be applied recursively. With the cost model proposed, the result from [88] is evaluated to show that the solution from the affinity-based algorithm does not always lead to a reduction in the number of disk accesses. Chu [33] presented two procedures to solve the attribute partitioning problem to improve system performance by transferring small segments instead of big unpartitioned relations between the primary and the secondary storage. He first defined two concepts, *sufficient* and *support*, on which two procedures, MAX and FORWARD SELECTION, are proposed which are targeted at maximizing the value of $\nu = \alpha + \beta - \phi$, the total reduction of costs which are expressed in terms of the number of disk accesses. The important characteristic of these two procedures is that they treat the transactions instead of the attributes as the decision variables. Therefore, the run time of these procedures does not depend on the number of attributes and can be efficiently executed when the number of attributes are very big. However, this approach may not be suitable to the situation when there are a large number of transactions but a small number of attributes over a relation. Also, this approach only discusses the problem of attribute partitioning for two memory levels on one disk. The objective function only counts the the number of disk accesses. Approaches in both [38] and [33] are not suited for distributed databases.

Chakravarthy *et al.* [28] argued that there should be a way to measure the goodness of a vertical fragmentation schema. For this purpose they set up an objective function, Partition Evaluator (PE), for evaluating different vertical fragmentation algorithms using the same criteria. The PE consists of two components, irrelevant local attribute access cost and relevant remote attribute access cost. However, relevant remote attribute access cost reflects the number of relevant attributes in a fragment accessed remotely with respect to all other fragments by all transactions. Therefore, the PE cannot be used in distributed databases because neither size nor network transaction cost factors have been considered.

Cheng *et al.* [31] proposed a genetic search-based clustering algorithm for data partitioning to achieve high database retrieval performance (with the partitioning problem being formulated as the traveling salesman problem). Son and Kim [106] argued that vertical partitioning problem should consider the number of fragments finally generated. They discussed n-ary vertical partitioning problem which are more flexible than the optimal partitioning. Their novel contribution is an objective function which aims at minimizing not only the frequency of query accesses to different fragments but also the frequency of interfered accesses between queries. In the objective function, data localization is not considered because queries are not distinguished between sites.

2.2.2 Vertical Fragmentation for Object Oriented Databases

In the context of object-oriented data model, fragmentation algorithms are mainly affinity-based, with different affinities used as parameters, e.g., method affinities, attribute affinities, or instance variable affinities [17, 46, 63]. An increasing demand on the performance of object-oriented database systems has resulted in the adoption of vertical fragmentation techniques from the relational databases. The features of the object-oriented data model (such as inheritance, encapsulation, ISA relationship and the presence of method) add to the complexity of the partitioning problem [32]. Based on the existing work for vertical fragmentation for object-oriented databases, a taxonomy is proposed in [32] which has two categories, method based and attributed based. In [50] different approaches are classified into a hybrid affinity cost-based and pure affinity-based approach.

Further to the work in [63], Karlapalem and Li [59] discussed the foundations of vertical fragmentation by giving a formal representation of vertical fragmentation. Issues regarding internal representation and reconstruction of fragments are discussed. In addition, approaches for supporting ISA relationships and methods are briefly mentioned. There are neither algorithms for horizontal, path and vertical fragmentation nor discussion on when vertical fragmentation should be applied to schema or to methods. Karlapalem *et al.* [61] presented guidelines for method induced partitioning in object-oriented databases. Karlapalem and Li [60] extended the work done in [59] and [61] through detailed discussions of the method-induced partitioning on different types of methods in terms of instance variables accessed, and the complexity of the methods, with the focus on single method partitioning. There is no algorithms proposed in the paper.

To study the effectiveness of vertical partitioning in OODB, Fung *et al.* [52] presented a cost model to measure the number of disk accesses for query processing. Two algorithms for deriving optimal and near-optimal vertical class partitioning schemes are proposed. One is cost-driven algorithm, which provides an optimal vertical class partitioning by exhaustively enumerating all possible vertical fragmentation schemata to get a schema which leads to the least costs, using the cost model that calculates the number of disk accesses. Due to the complexity of the cost driven-approach, the other algorithm, hill-climbing heuristic algorithm (HCHA), is proposed. It takes as input the result from the affinity-based algorithm to improve it by further reducing the total number of disk accesses, following a set of procedures, left_merge, right_merge, and split_new, to get a set of possible vertical fragmentation schemata. Then the cost model is utilized to find the optimal one. Note that the heuristic is an iterative procedure, the complexity of which is the complexity of algorithm in [89] plus the complexity of HCHA. However, the cost model focuses only on the number of page accesses and therefore does not suit distributed databases. Fung *et al.* [51] improved the cost model in [52] by considering extra processing for dealing with the composite objects generated from vertical partitioning and propose a comprehensive analytical cost model that supports the main OODB features, including subclass hierarchy and class composition hierarchy. However, navigation operations may not benefit from fragmentation because relationships between classes are not considered during the process of fragmentation. Overall, the participation algorithms do not suit distributed databases.

Treating relational database as a special case of object-oriented databases, Malinowski and Chakravarthy [83] generalized the work for relational databases in [28] to object-oriented databases. Vertical fragmentation is performed using Transaction-Method Usage Matrix, Method-Method Usage Matrix and Method-Attribute Usage Matrix. A partition evaluator

function for object-oriented databases, PEOO, adopted from the relational databases, is used to evaluate all possible combinations of attributes with the number of fragments varying from one to the total number of attributes in the class. However, without considering the size of data transferred among network nodes, the PEOO actually measures the number of irrelevant local accesses and relevant remote accesses rather than real total query costs.

Based on the classification initially presented in [45], Ezeife and Barker [46] discussed vertical partition for the most complex object model, consisting of complex attributes with complex methods. Ezeife and Barker [47] emphasized that the network communication costs dominate query processing costs. Vertical fragmentation is discussed with reference to four different class models, consisting of simple or complex attributes combined with simple or complex methods. Fragmentation of a class is processed to group all attributes and methods of the class that are frequently accessed together by applications accessing either the class itself, its subclasses, its containing classes, or its complex method classes. For different models affinity matrixes are computed by incorporating all the object-oriented features, e.g., inheritance links and subclasses. For each of the class model a formal vertical fragmentation is presented. Method affinities of a class are calculated by summing up the frequencies of queries that access both the methods simultaneously, either directly or through this subclasses or containing classes. The partitioning algorithm in [88] is used. However, all the algorithms presented aim at splitting a class such that attributes and methods accessed most frequently are grouped together. However, there is no proof that the partitioning algorithm can indeed minimize remote transportation costs. Actually, site information are lost while building affinity matrixes, which means that data localization is not considered. The evaluation of proposed algorithm is based on the Partition Evaluator proposed in [28], which does not really measure the total query costs.

Treating attributes and methods in a uniform and undistinguished way, Baião [10] adopted the graphical approach in [89] and [87] to object-oriented databases. The process of vertical fragmentation contains two steps, building an element affinity matrix and building an element affinity graph. However, during the process of building the element affinity matrix, data local requirement information is lost. Therefore, there is no link showing that vertical fragmentation using element affinity can improve data locality which in turn can reduce irrelevant remote data transportation costs.

2.3 Mixed Fragmentation

Navathe *et al.* [87] proposed a mixed fragmentation methodology for initial distributed database design. The process proposed simultaneously applies horizontal and vertical fragmentation on a relation. The input of the procedure comprises a predicate affinity table and an attribute affinity table. A set of grid cells are created first which may overlap each other. Then some grid cells are merged such that total disk accesses for all transactions can be reduced. Finally, overlap between each pair of fragments is removed using two algorithms for the cases of contained and overlapping fragments. However, the merging algorithms are based on a model which measures times of disk access (I/O). Network factors are not considered. For distributed databases, it is important to not only reduce disk access but also reduce the data transportation between sites.

Adopting some developed heuristics and algorithms in [87] to fragmentation in object-oriented databases, Baião and Mattoso [8] proposed a design procedure which includes a

sequence of steps: analysis phase, vertical and horizontal fragmentation. In the first step, a set of classes that are needed for horizontal fragmentation, vertical fragmentation, or non-fragmentation, are identified. In the second and third steps, vertical and horizontal fragmentation are performed on the classes identified in the first step, using algorithms extended from the one in [87]. All fragmentation algorithms are affinity based. The evaluation of the resulting fragmentation are not based on any cost model. Baião *et al.*[10] considered mixed fragmentation as a process of performing vertical fragmentation on classes first and then performing horizontal fragmentation on the set of vertical fragments.

2.4 Allocation

In the literature, allocation problems are first raised for file allocation. Chu [34] presented a simple model for a nonredundant allocation of files. Casey [23] proposed a model which allows the allocation of multiple copies. Queries and updates are distinguished in the model. In addition to file allocation, allocation of application programs are considered in [68]. Mahmoud and Riodon [82] proposed a model for studying file allocation and the capacity of communication capacities to obtain optimized solution which minimize storage and communication cost.

Since the early 1980s data allocation has been studied in the context of relational databases. Due to the complexity of the problem of data allocation, different researchers make different assumptions to reduce the size of the problem. Some works do not consider replication while making decision of allocation [5, 24, 37, 39, 64, 85], while some others do not consider storage capabilities of network nodes [7, 18, 58, 100]. Some assume the query site strategy that shifts all data needed by a query to site required them [11, 14, 18, 37, 58, 82, 107], while others only consider communication costs as data volume in the cost model with an assumption of the simple query environment [7, 18, 24, 39, 40, 84, 85, 100].

2.4.1 Simple Query Environment

Many researchers presume the *simple query environment* (as defined in [56]) which assumes a fully connected, geographically distributed network with identical transmission cost factors among any pair of sites, no storage constraints at sites and negligible local processing costs.

Ceri *et al.* [24] proposed an optimization model for non-replicated data allocation with the objective function to minimize total transaction processing cost. A horizontal fragmentation schema and transaction execution strategies are needed as input for the fragment allocation model. It is assumed that the unit transmission cost is the same among any two nodes. Due to the complexity of the problem, decomposition heuristics are presented. A greedy algorithm is proposed to comprise replicated allocation of fragments of the whole relation. However, only predefined schedules consisting of basic operations are considered. This approach cannot be extended to vertical fragmentation because of the number of possible vertical fragments of a relation.

Aper [7] pointed out that the main differences between the file allocation problem and the allocation problem in a distributed database is that query processing should be considered while making decisions of fragment allocation. The study of fragment allocation is based on query processing strategies which treat non-redundant and redundant allocation of fragments in a uniform way. It is shown that the problem of determining a non-redundant optimized allocation for fragments is NP-hard. Using static and dynamic processing schedules, heuristic

and optimal algorithms for minimizing total transmission cost are investigated. A heuristic algorithm is proposed based on the assumption of there being a copy for all fragments for each query. However, the heuristic is still too complicated for practical use because the variables that need to be decided include not only the number of fragments but also the number of copies. Aper's dynamic processing schedules are more reasonable but infeasible to be applied by practical applications because of their complexities.

Cornell and Yu [39] proposed an allocation strategy which integrates the problem of relation allocation with query strategy to optimize performance of a distributed database system. However, the linear programming solution is complicated for practical application. To simplify the problem it is assumed that networks are fully connected with equal bandwidth for each link. To extend the work in [40], March and Rho [84] proposed a comprehensive cost model which takes into consideration replication, operation allocation, and concurrency control. However, there is no algorithm presented.

Sarathy *et al.* [100] considered the problem of allocating relations or fragments and their copies as a contained nonlinear integer model which is proved NP-hard. Based on linearization and subgradient optimization, an algorithm is developed for solving this model. Menon [85] extended the integer model in [100] by including storage and processing capacity constraints in the model. At the same time he simplified the integer programming formulations to study the non-redundant version of the fragment allocation problem. Both [100] and [85] simplify the problem by assuming a constant transportation cost between any pair of sites. Therefore, the problem of minimizing the total cost of data transmission is the problem of minimizing the total data transmitted.

Teorey [108] discussed database distribution with examples to show how to apply some fragmentation and allocation strategies. The strategies for fragmentation are not based on quantity information but on analysis of the relationship between transactions and relations. Two variations of the all beneficial sites method, one-step and cumulative, can not guarantee global optimal solutions. Ra and Park [95] developed a scheme for PC-based distributed database design which is based on mixed partitioning technique using a grid approach. Grid cells are merged to get mixed fragments, which are allocated for a PC based distributed database using transaction mapping information. However, there are no algorithms but some rules on how to perform allocation. Fragments requested from only one site are allocated to the site requested while fragments not requested would be allocated to a server for future use. Commonly accessed fragments can be allocated to a database server or to all sites requesting them. The determination depends on the trade-off between the transmission cost if it is allocated to a database server and update cost if it is allocated to all requesting sites.

2.4.2 Query Site Strategy

Some other researchers simplify the allocation problem by assuming that queries are processed using a *query site strategy* that shifts all data needed by a query to the site issuing the query.

Corcoran and Hale [37] proposed the use of a genetic algorithm (GA) to solve some combinatorial problems. They compare GA and greedy heuristic in obtaining optimal and near optimal fragment placements for the allocation problem with various data sets. A cost model is proposed first to measure the transmission cost between sites. A objective function is used to evaluate the fitness of each solution. Whenever an infeasible solution, which exceeds the capacity of any site, occurs a penalty will be incurred. The penalty depends on the number of sites. How to decide the penalty is not mentioned in the paper. However, this approach

neither considers distributed query execution plan nor considers transportation cost factors while calculating the costs. Moreover, replication is not considered in this paper.

Tamhankar and Ram [107] proposed an integrated methodology incorporating replication and concurrency control costs for fragmentation and allocation. This methodology consists of four steps: primary distribution, secondary distribution for response time, secondary distribution for availability, and secondary distribution for storage space. In step one, analysis is carried to explore all distribution options. Based on some guidelines and a simplified cost model, a distribution matrix is obtained to identify the sites of allocation for each relation or fragment and its synchronized and unsynchronized copies. However, for fragmentation, there are few guidelines that one can follow without the presence of any detailed design procedure. It is suggested that vertical fragmentation be used to move the accessed data to the site of transaction. However, there is no further discussion on how to perform vertical fragmentation.

Bellatreche *et al.* [14] adopted the allocation approaches in [13] and discussed methods and class allocation problems by taking into consideration complex interdependencies among queries, methods and classes. A cost model is proposed to measure the total data transfer incurred in processing a given set of queries. By using the cost model, an iterative approach generates a near-optimal solution for the combined methods and class allocation problems. Distributed query processing is not considered. The costs are computed as the costs of transferring data to sites of methods. The cost model consists of three parts: data transfer cost due to method-class dependency, method-method dependency, and the possible remote execution of method(s), respectively.

Barker and Bhar [11] stated that fragmentation is a process of grouping programs and data based on the locality of accesses shown by application access behavior. Based on the assumption that the data accessed by the method are 'shipped' to the site of the method, an algorithm is proposed to get a "near optimal" non-redundant distribution of fragments by exchanging or moving fragments between every pair of sites. An initial allocation is obtained based on the available space at each site and the applications that reference objects in the fragments. Fragmentation and allocation are considered as two orthogonal problems which are actually closely interrelated [93]. This approach works best for two sites. However, with an increase in the number of network sites, it becomes increasingly difficult to obtain a pairwise optimality, and this is less likely to be a globally optimal solution.

Huang and Chen [58] discussed fragment allocation in distributed database design with the consideration of replication by distinguishing retrieval and update queries. Using a cost model, two heuristics are proposed to minimize the communication cost as much as possible. Both heuristics have an initial allocation involving sending a replica to each site that needs it. The first heuristic removes replica one by one by exploring all the possible orders. The second removes replica according to the update costs the replica at each site caused in a descending order. Two experiments are conducted, with one comparing performance of the proposed heuristics with the optimal solutions under a hypothetical environment and the other validating the cost model through a simulation. However, how the results for the optimal solution under each case are computed is not presented. Distributed query execution is not considered by this approach while calculating query costs.

Gavish and Pirkul [53] studied the problem of location of computing resources and databases in a wide-area network. Blankinship *et al.* [18] proposed an iterative solution method as a heuristic to simultaneously solve the two NP-hard problems, data allocation and query optimization. An algorithm is proposed in [18] to allocate fragments to the site requesting them most frequently.

2.4.3 Others

Karlaplem and Pun [64] considered that a major cost of executing queries in distributed databases is the cost of transferring data between different sites that store the multiple data objects or fragments which are accessed by the queries. A site dependency graph is developed to model the dependencies among different fragments accessed by a query. Fragmentation design is assumed complete before allocation. With a fragmentation schema as input, an initial allocation schema is obtained by using the max-flow min-cut formulations of the problem with query-site query execution strategy. Then the second step, for the move-small query execution strategy, the initial allocation schema is refined iteratively using the hill climbing algorithm, considering dependency among fragments. However, it is not reasonable to set the capacity constraint of a site as the biggest number of fragments that can be stored at the site. Rather, it would be reasonable to set the capacity constraint of a site as the maximum data volume. Moreover, the costs of evaluating distributed queries should also contain the costs of transporting data between intermediate nodes of a query tree rather than just between fragments. This approach only suits the situation when fragmentation has been completed before allocation and the number of fragments is small.

Sacca and Wiederhold [99] assumed that network and processing capacities are limited and considered only transmissions between fragments as total query costs. Using the cost model proposed in [64], Ahmad *et al.* [5] proposed an evaluate evolutionary algorithms for data allocation for distributed database systems. Their experimental results show that no single algorithm outperforms all others in terms of both query response time and algorithm running time.

2.5 Summary

As shown in the above sections, most of the literature about database distribution design considers fragmentation and allocation as two different steps even though they are strongly interrelated problems which take the same input information to achieve the same objectives, i.e., improving system performance, reliability and availability.

Existing approaches for primary horizontal fragmentation can be characterized into three streams, one using minterm predicates, one using predicate affinity, and a cost-driven approach using a cost model. Even though each of the approaches claims to be able to improve system performance, there is no evaluation to prove that resulting fragmentation schemata can indeed improve the system performance. Horizontal fragmentation with minterm predicates often results in a large number of fragments which will later be allocated to a limited number of network nodes. It can be expected that the number of network nodes gives the upper bound of fragments because fragments allocated at the same network node can be recombined for the benefits of most queries. Affinity-based horizontal fragmentation approaches cannot guarantee to achieve optimal system performance because the information of data local requirement is lost while computing predicate affinities. Cost-driven approaches use cost models to measure the number of disk accesses without considering transportation cost. None of the three approaches takes data local availability as the objective of fragmentation.

For derived horizontal fragmentation, the restriction that derived fragmentation can only be performed on a member relation or non-leaf class is not reasonable. There are situations where there are predicates defined on member relations (or non-leaf classes) but no predicates defined on owner relations (or leaf classes) and there are queries access the owner relations

(or leaf classes) and the member relations (or non-leaf classes) simultaneously. Also, that the derived fragmentation of a member relation (or non-leaf class) is performed according to fragmentation of only one owner relation (or leaf class) is not reasonable. More finely derived fragmentation can be achieved by considering other owner relations or leaf classes.

For vertical fragmentation there are two main approaches existing in the literature: affinity-based and cost-based. The affinity-based vertical fragmentation approach originated for centralized databases with hierarchical memory levels, for which the number of disk accesses is the main factor that affects the system performance. Later, this approach was adapted to distributed databases for which transportation cost is the main cost that affects the system performance. Attribute affinities only reflect the togetherness of attributes accessed by applications. Vertical fragmentation based on affinities may reduce the number of disk accesses. However, there is no clear proof that affinity-based vertical fragmentation can indeed improve data local availability and thus improve system performance. The cost-driven approach performs vertical fragmentation based on a cost model that measures the number of disk accesses. The optimal solution chosen by this approach is the vertical fragmentation schemata that have the fewest number of disk accesses. Again, there is no proof that the resulting fragmentation schema is an optimal solution, or taking it as input will lead to optimal allocation, for distributed databases. Actually, for distributed databases, transportation costs dominate the total query costs. This means that improving data local availability, which in turn reduces remote data transportation, plays an important role in improving system performance.

As to allocation, due to the complexity of the allocation problem that is closely related to query optimization problem, it is infeasible to find optimal solutions. One has to seek heuristic solutions. To do this, many assumptions have been made to reduce the complexity of the problem. Beside the assumption that fragmentation has been done properly before allocation, other assumptions made in the literature include assuming the simple query strategy or the query site strategy, etc. The assumption that fragmentation is completed properly is not reasonable. Nor it is possible to solve the fragmentation problem independently from the allocation problem because the optimal fragmentation can only be achieved with respect to the optimal allocation of fragments [25]. The simple query strategy ignores the dynamic nature of networks. The query site strategy ignores the interdependency between allocation and distributed query optimization. As mentioned in [11], fragmentation based on "locality" of access exhibited by applications should be performed. However, there is no fragmentation approach, for both horizontal and vertical fragmentation, taking data locality into consideration. Further, in the step of allocation, all the data local requirement information is not attached with fragments as input information.

In summary, due to the deficiencies of fragmentation and allocation existing in the literature, and with the consideration of need of distribution design techniques for complex value databases, this research will study fragmentation and allocation in the context of complex value data models. The approach taken in this thesis will study fragmentation and allocation simultaneously. Based on a cost model, fragmentation and fragment allocation are performed with the objective of minimizing data transportation costs and maximizing data local availability (or data locality).

Chapter 3

Complex Value Databases and Query Language

3.1 The Data Model

In this chapter a generic approach to define a complex value data model, adopted from Higher-Order Entity Relationship Model (HERM) from [109], will be described. The aim of this chapter is to define a context of studying database distribution design. As mentioned in Chapter 1, the reason for choosing complex data models is that they cover the common aspects of object-oriented databases, object-relational databases and eXtensible Markup Language (XML). In order to define a complex data model, type systems with an ordinary set semantics are utilized. Further, a query language and a cost model will be presented for evaluating system performance of complex value databases.

3.1.1 Types and Trees

Following a basic observation in [101], each data model is mainly determined by the collection of types it supports. Here the simple approach to define a type system by base types and type constructors is taken. Types are then constructed by applying these constructors in an orthogonal way, though not all data models support full orthogonality. Using abstract syntax we may describe a *type system* as:

$$t = b \mid \mathbb{1} \mid (a_1 : t_1, \dots, a_n : t_n) \mid \{t\} \mid [t] \tag{1}$$

with pairwise different labels $a_1, \dots, a_n$. Here b represents a not further specified collection of base types, which may include *BOOL*, *CARD*, *INTEGER*, *DECIMAL*, *DATE*, etc. Furthermore, $(\cdot)$ is a record type constructor, $\{\cdot\}$ a set type constructor, $[\cdot]$ a list type constructor, $\mathbb{1}$ is a trivial type . For complex value data model the outmost constructor is a tuple type constructor.

Each type defines a set of values, its *domain* $dom(t)$, as follows:

- The domain $dom(b_i)$ of a base type b_i is some not further specified set V_i, e.g. $dom(BOOL) = \{\mathbf{T}, \mathbf{F}\}$, $dom(CARD) = \{0, 1, 2, \dots\}$, etc.
- $dom(\mathbb{1}) = \{\top\}$.
- $dom((a_1 : t_1, \dots, a_n : t_n)) = \{(a_1 : v_1, \dots, a_n : v_n) \mid v_i \in dom(t_i)\}$ for record types.

– For set types $dom(\{t\}) = \{\{v_1, \dots, v_k\} \mid v_i \in dom(t)\}$.
– For list types $dom([t]) = \{[v_1, \dots, v_k] \mid v_i \in dom(t)\}$, i.e. elements may appear more than once and are ordered.

Obviously, each value $v \in dom(t)$ for any type t can be represented by an ordered finite tree using the type constructors as labels for non-leaf nodes, while each leaf is labeled by a value of some base domain V_i.

Let us illustrate the mapping of a concrete data model to our generic one for a simplified version of the HERM [109].

Definition 3.1. A *database type of level* $k \geq 0$ has a name E and consists of a set $comp(E) = \{r_1 : E_1, \dots, r_n : E_n\}$ of components with pairwise different role names r_i and database types E_i on levels lower than k with at least one E_i of level exactly $k-1$, except for $k = 0$, a set $attr(E) = \{a_1, \dots, a_m\}$ of attributes, each associated with a data type $dom(a_i)$ as its domain, and a key $id(E) \subseteq comp(E) \cup attr(E)$. We write $E = (comp(E), attr(E), id(E))$. □

Definition 3.2. A *complex value database schema* is a finite set $\mathcal{S}$ of database types such that for all $E \in \mathcal{S}$ and all its components E_i we also have $E_i \in \mathcal{S}$. □

EXAMPLE 3.1. The following is a complex database schema for a simple university application:

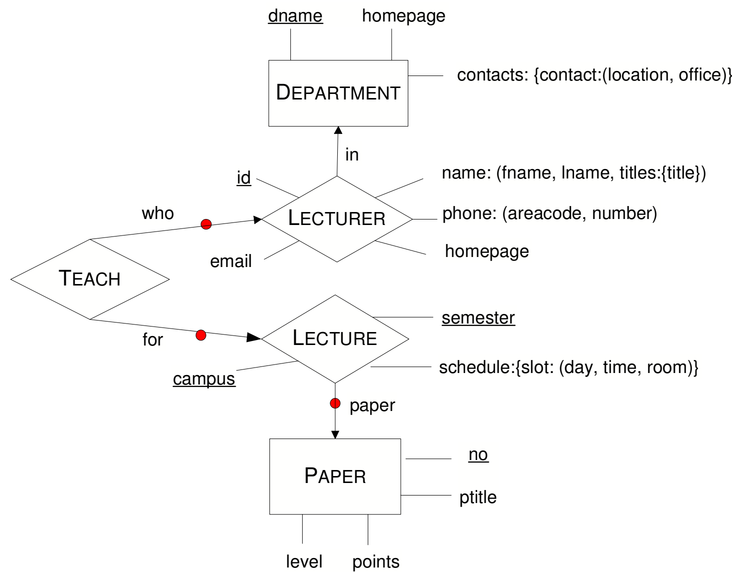

Fig. 3.1. HERM Diagram of the University Database

DEPARTMENT = (∅, {dname, homepage, contacts}, {dname})
LECTURER = ({in: DEPARTMENT}, {id, name, email, phone, homepage,}, {id})
PAPER = (∅, {no, ptitle, level, points}, {no})
LECTURE = ({paper: PAPER}, {semester, campus, schedule}, {paper: PAPER, semester, campus})
TEACH = ({who: LECTURER, for: LECTURE}, ∅, {who: LECTURER, for: LECTURE})

The domain types of complex attributes are as follows:

dom(contacts) = {contact: (location: *STRING*, office: *STRING*)}
dom(name)= (fname: *STRING*, lname: *STRING*, titles: {title: *STRING*})
dom(phone)= (areacode: *CARD*, number: *CARD*)
dom(schedule) = {slot: (day: *STRING*, time: *TIME*, room: *STRING*)} in the database type LECTURE. All other domains have been omitted.

□

Definition 3.3. Given a complex value database schema $\mathcal{S}$, we associate two types t_E and k_E – called *representation type* and *key type*, respectively – with each $E = (\{r_1 : E_1, \dots, r_n : E_n\}, \{a_1, \dots, a_k\}, \{r_{i_1} : E_{i_1}, \dots, r_{i_m} : E_{i_m}, a_{j_1}, \dots, a_{j_\ell}\}) \in \mathcal{S}$:

– The representation type of a database type E is the tuple type

$$t_E = (r_1 : k_{E_1}, \dots, r_n : k_{E_n}, a_1 : dom(a_1), \dots, a_k : dom(a_k)).$$

– The key type of a database type E is the tuple type

$$k_E = (r_{i_1} : k_{E_{i_1}}, \dots, r_{i_m} : k_{E_{i_m}}, a_{j_1} : dom(a_{j_1}), \dots, a_{j_\ell} : dom(a_{j_\ell})).$$

□

EXAMPLE 3.2. The list contains some of the representation types for the complex value database schema from Example 3.1:

$t_{\text{DEPARTMENT}}$ = (name: *STRING*, homepage: *URI*, contacts: {contact: (location: *STRING*, office: *STRING*)})
t_{LECTURER} = (in: (name: *STRING*), id: *STRING*, name: (fname: *STRING*, lname: *STRING*, titles: {title: *STRING*}), email: *EMAIL*, phone: (areacode: *CARD*, number: *CARD*), homepage: *URI*)
t_{PAPER} = (no: *CARD*, ptitle: *STRING*, level: *CARD*, points: *CARD*)
t_{LECTURE} = (paper: (no: *CARD*), semester: *STRING*, campus: *STRING*, schedule: {slot: (day: *STRING*, time: *TIME*, room: *STRING*)})
t_{TEACH} = (who: (id: *STRING*), for: (paper: (no: *CARD*), semester: *STRING*, campus: *STRING*,))

□

Definition 3.4. *Atomic attributes* are attributes with their domains as base types $dom(b_i)$. *Complex attributes* are attributes that are defined by iteratively using the tuple type, the set type and the list type constructors. □

Definition 3.5. A *complex value database db* over a schema $\mathcal{S}$ is an $\mathcal{S}$-indexed family $\{db(E)\}_{E \in \mathcal{S}}$ such that each $db(E)$ is a finite set of values of type t_E satisfying the following two conditions:

– whenever $t_1, t_2 \in db(E)$ coincide on their projection to $id(E)$, they are already equal;
– for each $t \in db(E)$ and each $r_i : E_i \in comp(E)$ there is some $t_i \in db(E_i)$ such that the projection of t onto r_i is t_i.

□

Note that the relational data model employs only base types and tuple type constructor and can be treated as a special case of the complex value data model. A relation contains a finite set of tuple-valued attributes.

EXAMPLE 3.3. An instance of the database schema defined in Example 3.1 is shown as in Table 3.1 and Table 3.2.

Table3.1. An Instance of the University Schema

db(DEPARTMENT)			
dname	homepage	contacts	
		contact	
		location	office
Information	is.massey.ac.nz	Turitea	SST2.05
		Wellington	DLB305
Accounting	ac.massey.ac.nz	Turitea	SST1.14
		Albany	PK3.05
Marketing	mk.massey.ac.nz	Napier	BSC1.11
		Albany	TIU331

db(PAPER)			
no	ptitle	level	points
110001	Accounting Principles	100	15
110105	Taxation	100	15
156100	Marketing Management	100	15
157221	Information Systems Analysis	200	15
157331	Database Concepts	300	15

db(LECTURER)								
in: DEPARTMENT	id	name			email	phone		homepage
		fname	lname	titles		area-code	number	
				title				
Accounting	1001	John	Dever	Professor Dr	j.dever	09	8556677	dever.com
Marketing	1002	Allan	Barry	Senior Lecturer Dr	a.barry	06	3556688	barry.com
Information	2010	Shirley	Churchill	Lecturer	s.churchill	04	4983677	churchill.com
Information	3203	Jerry	Hubbard	HoD Professor Dr	j.hubbard	06	3569988	hubbard.com
Accounting	2618	James	Hooks	Lecturer	j.hooks	04	4663365	hooks.com

□

Table3.2. An Instance of the University Schema (continued from Table 3.1)

db(LECTURE)					
paper	semester	campus	schedule		
			slot		
no			day	time	room
110001	0401	ALB	Monday	10am	SSLB1
			Thursday	2pm	SSLB5
156100	0402	ALB	Wednesday	9am	BSC201
			Friday	1pm	BSC201
110105	0501	WN	Tuesday	1pm	SST1
			Thursday	3pm	SST1
157221	0501	WN	Monday	9am	SST1
			Wednesday	1pm	SST1
157221	0601	PN	Monday	11am	SSLB2
			Wednesday	2pm	SSLB2
157331	0601	PN	Monday	11am	SSLB3
			Thursday	2pm	SSLB2
157331	0701	PN	Monday	10am	SSLB4
			Thursday	3pm	SSLB4

db(TEACH)			
who	for		
ID	no	semester	campus
1001	110001	0401	ALB
1002	156100	0402	ALB
2618	110105	0501	WN
2010	157221	0501	WN
3203	157221	0601	PN
3203	157331	0601	PN
3203	157331	0701	PN

3.1.2 Subtypes

Subtyping on the type system is defined in [101] with preorder $\leq$ on the types.

Definition 3.6. Suppose base types contains $\mathbb{1}$. Then subtyping can be defined as the smallest preorder such that the following holds:

1. $t \leq \mathbb{1}$ for any type t,
2. $\{t_1\} \leq \{t_2\}$ if and only if $t_1 \leq t_2$,
3. $[t_1] \leq [t_2]$ if and only if $t_1 \leq t_2$,
4. $(a_{i_1} : t_{i_1}, \ldots, a_{i_m} : t_{i_m}) \leq (a_1 : t_1, \ldots, a_n : t_n)$ with $1 \leq i_1 \leq i_2 \leq \ldots, i_m \leq n$ and $(a_1 : t_1, \ldots, a_n : t_n) \leq (a_1 : t'_1, \ldots, a_n : t'_n)$ if and only if $t_i \leq t'_i$ holds.

□

EXAMPLE 3.4. From Example 3.2 we take a representation type t_{PAPER} = (no: *CARD*, ptitle: *STRING*, level: *CARD*, points: *CARD*)
Then $t_{\text{PAPER}}$is a *subtype* of $t_{\text{PAPERTITLE}}$ = (no: *CARD*, ptitle: *STRING*). In other words, $t_{\text{PAPERTITLE}}$ is a *supertype* of t_{PAPER}. □

3.1.3 Rational Trees

Some data models including [1, 103] support also references, which then define rational trees, i.e. trees with only a finite number of distinct subtrees. In terms of the type system this can be expressed in two ways. The first one adds labels, so we get:

$$t = b \mid \mathbb{1} \mid \ell \mid \ell : t \mid (a_1 : t_1, \dots, a_n : t_n) \mid \{t\} \mid [t] \mid \langle t \rangle \mid (a_1 : t_1) \oplus \cdots \oplus (a_n : t_n) \tag{2}$$

Then a type $\ell : t'$ occurring inside the definition of a type t is said to *define the label* ℓ, whereas each plain occurrence of ℓ is said to *use the label* ℓ. Following [1] the restriction to rational trees can be simply expressed by the following two conditions:

– Within a type t a label ℓ can at most be defined once.
– Within a type t each label ℓ that is used must also be defined.

Example 3.5. Let us look at the following schema definition in XML Schema:

```
⟨xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"⟩
  ⟨xs:element name="cellar"⟩
    ⟨xs:complexType⟩
      ⟨xs:sequence⟩
        ⟨xs:element ref="wines"/⟩
        ⟨xs:element ref="wineries"/⟩
        ⟨xs:element ref="regions"/⟩
      ⟨/xs:sequence⟩
    ⟨/xs:complexType⟩
  ⟨/xs:element⟩
  ⟨xs:element name="wines"⟩
    ⟨xs:complexType⟩
      ⟨xs:element ref="wine" minOccurs="0" maxOccurs="unbounded"⟩
    ⟨/xs:complexType⟩
  ⟨/xs:element⟩
  ⟨xs:element name="wineries"⟩
    ⟨xs:complexType⟩
      ⟨xs:element ref="winery" minOccurs="0" maxOccurs="unbounded"⟩
    ⟨/xs:complexType⟩
  ⟨/xs:element⟩
  ⟨xs:element name="regions"⟩
    ⟨xs:complexType⟩
      ⟨xs:element ref="region" minOccurs="0" maxOccurs="unbounded"⟩
    ⟨/xs:complexType⟩
  ⟨/xs:element⟩
  ⟨xs:element name="wine"⟩
    ⟨xs:complexType⟩
      ⟨xs:sequence⟩
        ⟨xs:element name="name" type="xs:string"/⟩
        ⟨xs:element name="year" type="xs:integer" minOccurs="0"/⟩
        ⟨xs:element ref="blend" maxOccurs="unbounded"/⟩
```

```
        ⟨xs:element name=“price” type=“xs:decimal”/⟩
      ⟨/xs:sequence⟩
    ⟨/xs:complexType⟩
    ⟨xs:attribute name=“w-id” type=“xs:ID” use=“required”/⟩
    ⟨xs:attribute name=“producer” type=“xs:IDREF” use=“required”/⟩
  ⟨/xs:element⟩
  ⟨xs:element name=“blend”⟩
    ⟨xs:complexType⟩
      ⟨xs:sequence⟩
        ⟨xs:element name=“grape” type=“xs:string”/⟩
        ⟨xs:element name=“percentage” type=“xs:integer”/⟩
      ⟨/xs:sequence⟩
    ⟨/xs:complexType⟩
  ⟨/xs:element⟩
  ⟨xs:element name=“winery”⟩
    ⟨xs:complexType⟩
      ⟨xs:sequence⟩
        ⟨xs:element name=“name” type=“xs:string”/⟩
        ⟨xs:element name=“owner” type=“xs:integer” maxOccurs=“unbounded”/⟩
        ⟨xs:element name=“area” type=“xs:string” minOccurs=“0”/⟩
        ⟨xs:element name=“established” type=“xs:date” minOccurs=“0”/⟩
      ⟨/xs:sequence⟩
    ⟨/xs:complexType⟩
    ⟨xs:attribute name=“v-id” type=“xs:ID” use=“required”/⟩
    ⟨xs:attribute name=“in-region” type=“xs:IDREF” use=“required”/⟩
  ⟨/xs:element⟩
  ⟨xs:element name=“region”⟩
    ⟨xs:complexType⟩
      ⟨xs:sequence⟩ ⟨xs:element name=“name” type=“xs:string”/⟩ ⟨/xs:sequence⟩
    ⟨/xs:complexType⟩
    ⟨xs:attribute name=“r-id” type=“xs:ID” use=“required”/⟩
    ⟨xs:attribute name=“famous-wines” type=“xs:IDREFS” use=“required”/⟩
  ⟨/xs:element⟩
⟨/xs:schema⟩
```

That is, a cellar contains a list of wines, wineries and regions. A wine is described by a name, a year (optional) and a blend (which is a sequence of grapes together with their percentages). A winery is described by a name, a list of owners, an area (optional) and an establishment date (optional). A region just has a name. Furthermore, there are references from a wine to the winery that produces it, from a winery to the region it is located in, and from a region to all its famous wines.
Using the label-extended type system, we can represent this schema by three classes, wines, wineries and regions, with the following representation types (with labels w-id, v-id and r-id):

t_{wines} = w-id : (w-name : $STRING$, year : (y : $CARD$) $\oplus$ (n : $\mathbb{1}$), blend : [(grape : $STRING$, percentage : $CARD$)], price : $DECIMAL$, producer : v-id)

t_{wineries} = v-id : (v-name : *STRING*, owner : [*STRING*], area : (k : *STRING*) $\oplus$ (u : $\mathbb{1}$), established : (y : *DATE*) $\oplus$ (n : *EMPTY*), in-region : r-id)
t_{regions} = r-id : (r-name : *STRING*, famous-wines : [w-id])

□

It is commonly known that reference structures can be modeled equivalently by using identifiers. That is, we assume a special base type *ID* with $dom(ID) \cap V_i = \emptyset$ and set $dom(\ell) = dom(ID)$ and $dom(\ell : t) = \{(i, v) \mid i \in dom(ID), v \in dom(t)\}$. Using such a representation leads back to finite ordered trees at the price of having more complicated occurrence constraints [103].

Example 3.6. Using a representation with identifiers we may further modify the representation types in Example 3.5 to:

t_{wines} = (w-id : *ID*, w-name : *STRING*, year : (y : *CARD*) $\oplus$ (n : $\mathbb{1}$), blend : [(grape : *STRING*, percentage : *CARD*)], price : *DECIMAL*, producer : *ID*)
t_{wineries} = (v-id : *ID*, v-name : *STRING*, owner : [*STRING*], area : (k : *STRING*) $\oplus$ (u : $\mathbb{1}$), established : (y : *DATE*) $\oplus$ (n : $\mathbb{1}$), in-region : *ID*)
t_{regions} = (r-id : *ID*, r-name : *STRING*, famous-wines : [*ID*])

□

Therefore, in the following I will only consider the slightly simpler type system in (1), assuming that one of the base types is *ID*.

The query algebra used in this thesis is based on the work in [101]. The starting point for this approach to query algebra is that complex value data models mainly differ by the supported complex value type constructors (e.g. tuples, lists, sets, multisets, unions, etc.), the presence or absence of references, and the restrictions on how these constructors can be combined. This captures: complex value and object-relational data models [109]; object oriented data models [103]; and the models of semi-structured data and XML [1, 54], in particular the various approaches to defining schemata for XML [20, 35, 49] including the de facto standard XML Schema [112].

Generic algebra combines operations defined for the type constructors including structural recursion, generalized join operations and abstract object creation [3]. As a consequence, it captures XML algebra [48], and can be used to represent the gist of XML query languages [4, 21, 42] including the emerging standard XQuery [111], as shown in [66].

3.2 Query Algebra and Optimization

Let us now take a look at queries. It is beyond the scope of this thesis to fully discuss query languages for complex value data models. Instead of this the viewpoint is adopted that the implementation of any kind of query will somehow exploit a query algebra, and that queries in such an algebra give rise to query trees, e.g., [93]. Such query trees are the subject of query optimization. It is well known that query trees that represent relational algebra queries are subject to heuristic algebraic query optimization, which rearranges the tree. However, other query optimization techniques such as sideways information passing [2], which affects the order of joins, and tableau optimization [2], which for conjunctive queries discovers and removes redundant joins, also give rise to a reorganization of query trees.

Therefore, it makes sense to restrict ourselves to query algebra. In particular, we will see later that optimized query trees will only be marginally affected by subsequent fragmentation. In other words, we can and should start from optimized query trees.

3.2.1 A Generic Query Algebra

According to the definition of complex value databases in Section 3.1 the definition of a query algebra is mainly determined by operations defined on the type system. There is no operation on $\mathbb{1}$, but for $BOOL$ we may consider the operations:

- *conjunction* $\wedge : BOOL \times BOOL \to BOOL$,
- *negation* $\neg : BOOL \to BOOL$,
- *implication* $\Rightarrow: BOOL \times BOOL \to BOOL$,
- two constants $\mathtt{true} : \mathbb{1} \to BOOL$ and $\mathtt{false} : \mathbb{1} \to BOOL$.

For tuple types we consider:

- *projection* $\pi_i : t_1 \times \cdots \times t_n \to t_i$ and
- *product* $o_1 \times \cdots \times o_n : t \to t_1 \times \cdots \times t_n$ for given operations $o_i : t \to t_i$.

For set types we may consider

- union $\cup$,
- difference $-$,
- the constant $\mathtt{empty} : \mathbb{1} \to \{t\}$ and
- the *singleton* operation $\mathtt{single} : t \to \{t\}$ with well known semantics.

In addition, we may consider structural recursion, which exploits the constructors for finite sets, i.e. the constant `empty`, the singleton operation and the union operation. In order to define an operation on a set type, say $\mathtt{op} : \{t\} \to t'$, it is therefore sufficient to define it on the empty set, on singleton sets and on unions. Formally, we can define structure recursion as below:

Definition 3.7. Given $\mathtt{op} = \mathtt{src}[e, g, \sqcup]$ with a value e of type t', a function $g : t \to t'$ and a function $\sqcup : t' \times t' \to t'$. Then $\mathtt{src}[e, g, \sqcup]$ is defined as:

$$\begin{aligned} \mathtt{src}[e, g, \sqcup](\emptyset) &= e, \\ \mathtt{src}[e, g, \sqcup](\{x\}) &= g(x) \text{ for each } x \text{ of type } t, \text{ and} \\ \mathtt{src}[e, g, \sqcup](X \cup Y) &= \mathtt{src}[e, g, \sqcup](X) \sqcup \mathtt{src}[e, g, \sqcup](Y) \text{ for disjoint } X, Y \text{ of type } [t]. \end{aligned}$$

□

Similar definitions can be given for lists as in [66].

It is easy to see that structural recursion is able to express projections, selections and aggregate functions as they appear in standard query languages such as SQL.

Definition 3.8. Considering a function $f : t \to t'$ for arbitrary types t and t'. We can "raise" f to a function $\mathtt{map}(f) : \{t\} \to \{t'\}$ by applying f to each element of a set to get:

$$\mathtt{map}(f) = \mathtt{src}[\emptyset, \mathtt{single} \circ f, \cup]$$

Projection is just a special case of `map`. □

Definition 3.9. Considering a function $\varphi : t \rightarrow BOOL$, we define *selection* as an operation $\texttt{filter}(\varphi) : \{t\} \rightarrow \{t\}$, which associates with a given set the subset of all elements "satisfying the predicate" φ, i.e. elements that are mapped to $\mathbf{T}$. Then we may write

$$\texttt{filter}(\varphi) = \texttt{src}[\emptyset, \texttt{if_then_else} \circ (\varphi \times \texttt{single} \times (\texttt{empty} \circ \texttt{triv})), \cup].$$

with the function $\texttt{if_then_else} : BOOL \times t \times t \rightarrow t$ with $(\mathbf{T}, x, y) \mapsto x$ and $(\mathbf{F}, x, y) \mapsto y$ and $\texttt{triv} : t \rightarrow \mathbb{1}$ as a unique "forget"-operation for each type t. □

For completeness we also use operations on functions, in particular, *composition* $\circ : (t_2 \rightarrow t_3) \times (t_1 \rightarrow t_2) \rightarrow (t_1 \rightarrow t_3)$, *evaluation* $\texttt{ev} : (t_1 \rightarrow t_2) \times t_1 \rightarrow t_2$, and *abstraction* $\texttt{abstr} : (t_1 \times t_2 \rightarrow t_3) \rightarrow (t_1 \rightarrow (t_2 \rightarrow t_3))$. All these operations are standard.

Furthermore, assume an equality predicate $=_t : t \times t \rightarrow BOOL$ for all types t except function types and a membership predicate $\in : t \times \{t\} \rightarrow BOOL$. Combining all the operations for all types of the type system gives all operations induced from the type system.

In [101] it has been shown that these operations suffice to express more complex operations for nesting and unnesting.

Finally, we will need a generalized join-operation. If t is a common supertype of t_1 and t_2 with associated subtype functions $\pi_t^i : t_i \rightarrow t$, then there exists a common subtype $t_1 \bowtie_t t_2$ together with subtype functions $\pi_{t_i} : t_1 \bowtie_t t_2 \rightarrow t_i$ such that $\pi_t^1 \circ \pi_{t_1} = \pi_t^2 \circ \pi_{t_2}$ holds. Furthermore, for any other common subtype t' with subtype functions $\pi'_{t_i} : t' \rightarrow t_i$ with $\pi_t^1 \circ \pi'_{t_1} = \pi_t^2 \circ \pi'_{t_2}$ there is a unique subtype function $\pi : t' \rightarrow t_1 \bowtie_t t_2$ with $\pi_{t_i} \circ \pi = \pi'_{t_i}$.

3.2.2 Path Expressions

In the relational data model, only simple attributes are used. Therefore, it is straightforward to specify attributes of a relation schema. In complex value data models, however, complex attributes are involved. For this we need to define path expressions. Path expressions are defined as below.

Definition 3.10. Path expressions may have the following formats:

- the empty path ε is a path for every type t,
- if $path_i$ is a path for t_i then $a_i.path_i$ is a path for $t = (a_1 : t_1, \dots, a_n : t_n)$,
- if $path$ is a path for t' then $path$ is also a path for $t = \{t'\}$,
- if $path$ is a path for t' then $path$ is also a path for $t = [t']$.

□

3.2.3 A Simple Query Algebra

Using the query algebra introduced in the above section, each query gives rise to a query tree in the same way as for the relational data model. Furthermore, there are equalities among the algebraic query expressions which will allow us to rearrange the execution order of the operations. Without going into detail – the heuristics is the same as for the RDM – we can rearrange a query tree in a way that (if possible) we first apply structural recursion operations $src[e, g, \sqcup]$ on the sets of input database, i.e. on some $db(E_i)$.

However, no complete theory of query optimization based on structural recursion is known. We therefore turn back to a simpler query algebra with operations that can be expressed in the generic algebra above, but will be sufficient for studying database distribution design in this thesis. The following defines a simple query algebra:

– Projection $\pi_{t_{E'}}(db(E))$ with a super type $t_{E'}$ of t_E, and resulting in

$$db(E') = \{v' : t_{E'} \mid \exists v \in db(E).v' = \pi^{t_E}_{t_{E'}}(v)\}.$$

– Selection $\sigma_\varphi(db(E))$ with a selection formulae φ as discussed above, resulting in

$$db(E') = \{v : t_E \mid \exists v \in db(E) \wedge \varphi(v) = true\}.$$

In general, a selection condition or *simple predicate* for an instance $db(E)$ has the form $path\ \theta\ v$ with a path expression $path$ on $db(E)$, a value v of the corresponding type, and a comparison operator θ, which can be one of $=, \neq, \leq, <, >, \geq, \subseteq, \supseteq, \not\subseteq, \not\supseteq, \in, \notin, \ni$, and $\not\ni$. More complex selection formulae can be built as logical combinations of simple ones using $\wedge$, $\vee$ and $\neg$.

– Join $db(E_1) \bowtie_t db(E_2)$ with common super type $t \leq t_i$ for $i = 1, 2$ as discussed in the previous subsection, resulting in

$$db(E_1) \bowtie_t db(E_2) = \{v : t_1 \bowtie_t t_2 \mid \exists v_1 \in db(E_1).\exists v_2 \in db(E_2).\pi_{t_1}(v) = v_1 \wedge \pi_{t_2}(v) = v_2\}.$$

– Semijoin as applying join operation first and then applying projection operation:

$$db(E_1) \ltimes_t db(E_2) = \{v : \pi_{t_1}(t_1 \bowtie_t t_2) \mid \exists v' \in db(E_1) \bowtie_t db(E_2). \wedge v = \pi_{t_1}(v')\}.$$

– Set operations $\cup$, $\cap$ $-$ for union, intersection and difference of $db(E_1)$ and $db(E_2)$, both of type t.

$$db(E_1) \cup db(E_2) = \{v : t \mid v \in db(E_1) \vee v \in db(E_2)\}.$$

$$db(E_1) \cap db(E_2) = \{v : t \mid v \in db(E_1) \wedge v \in db(E_2)\}.$$

$$db(E_1) - db(E_2) = \{v : t \mid v \in db(E_1) \wedge v \notin db(E_2)\}.$$

– Nesting ν_t with an embedded type t inside a record constructor. All values w of type t that are part of values that are equal "except for w" are grouped into a set.

$$\nu_t(db(E)) = \{v : (t_E - t) \cup \{t\} \mid \exists v_1 \in db(E).\pi^{t_E}_{t_E - t}(v) = \pi^{t_E}_{t_E - t}(v_1) \wedge \pi^{t_E}_t(v) = \{\pi^{t_E}_t \mid \exists v_2 \in db(E) \wedge \pi^{t_E}_{t_E - t}(v_1) = \pi^{t_E}_{t_E - t}(v_2)\}\}.$$

– Unnesting υ_t with a set type $\{t\}$. It reverses a nest-operation.

$$\upsilon_t(db(E)) = \{v : (t_E - \{t\}) \cup t \mid \exists v' \in db(E).\pi^{t_E}_{t_E - \{t\}}(v) = \pi^{t_E}_{t_E - \{t\}}(v') \wedge \pi^{t_E}_t(v) \in \pi^{t_E}_t(v')\}.$$

– Dereferencing δ, which replaces identifiers in the first argument by the corresponding value in the second argument.

$$\delta(db(E), db(D)) = \{v : (t_E - k_D) \cup t_D \mid \exists v_1 \in db(E).v_2 \in db(D).\pi^{t_E}_{t_E - k_D}(v) = \pi^{t_E}_{t_E - k_D}(v_1) \wedge \pi^{t_E}_{t_D}(v) = v_2)\}.$$

– Identifier creation γ, which creates new identifiers.

$$\gamma(db(E)) = \{(i, v) : \{I\} \cup t_E \mid i :: I \wedge i \notin id(db).\exists v \in db(E)\}.$$

– Renaming ϱ_f with a renaming function f that maps names of labels in types to new names. So we can assume to write $f = \{n_1 \mapsto m_1, \ldots, n_k \mapsto m_k\}$ with pairwise different "old" names n_i $(i = 1, \ldots, k)$ that are replaced by pairwise different "new" names m_i $(i = 1, \ldots, k)$.

$$\varrho_{n_1 \mapsto m_1, \ldots, n_k \mapsto m_k}(db(E)) = \{v : t_E - (n_1, \ldots, n_k) \cup (m_1, \ldots, m_k) \mid \exists v' \in db(E). \\ \pi_{m_1, \ldots, m_k}(v) = \pi_{n_1, \ldots n_k}(v') \wedge \pi_{t_E - (m_1, \ldots, m_k)}(v) = \pi_{t_E - (n_1, \ldots, n_k)}(v')\}.$$

Example 3.7. Let us look again at Example 3.1. The following are algebraic query expressions:

– $\pi_{\text{id: } STRING, \text{ name.fname:} STRING}(db(\text{Lecturer}))$, which projects to lecturers' id and first name.
– $\gamma(\nu_{\text{in: (dname: } STRING)}(\pi_{\text{in: (dname: } STRING), \text{ name.lname: } STRING, \text{ id: } ID}(db(\text{Lecturer}))))$ project each lecturer onto the department he or she works in, his or her lname and id, then nest with respect to the department, then create new identifiers for each element in the set.

□

In addition to these operations we use *amalgams*, i.e. sequences of operations that are executed in one step. We write $amg[o_n, \ldots, o_1]$ for an amalgam involving the operations $o_1, \ldots, o_n$. Then

$$\begin{aligned} &amg[o_n, \ldots, o_1](db(E_1), \ldots, db(E_k)) \\ =&amg[o_n, \ldots, o_2](o_1(db(E_1), \ldots, db(E_{ar_1})), db(E_{ar_1+1}), \ldots, db(E_k)), \end{aligned}$$

where ar_i is the arity of operation o_i. In particular, we need $k = 1 + \sum_{i=1}^{n}(ar_i - 1)$ arguments for $amg[o_n, \ldots, o_1]$.

Example 3.8. $amg[\pi_{t'}, \bowtie_t]$ denotes an amalgam of arity $k = 2$ with

$$amg[\pi_{t'}, \bowtie_t](db(E_1), db(E_2)) = \pi_{t'}(db(E_1) \bowtie_t db(E_2)).$$

The rationale behind this amalgam is that if the join is implemented by a merge-join, or a similar implementation approach, elements in $db(E_1)$ and $db(E_2)$ would first be ordered, then joined. A follow-on ordering for the projection would be unnecessary. To the contrary, the projection of each value can be made part of the join.

□

Using this, algebra queries give again rise to query trees. We assume that queries have been optimized before fragmentation using various distributed query optimization techniques, cf. [2, 67, 93, 110]. It is common practice in query optimization for relational databases to perform selection operations first, and projection operations as early as possible [2, 67, 93, 110]. It has further been shown that most algebraic optimization rules for relational databases can be extended to complex value databases, with the exception of the commutativity laws for joins

and selections when these operations involve subrelations or set-valued attributes [36, 71, 97]. Selection operators used in this thesis do not involve subrelations. Therefore the optimization assumption mentioned in the beginning of this paragraph allows us to focus on subqueries of the form

$$\pi_t(\sigma_\varphi(db(E))) \qquad (*)$$

We will see that splitting, horizontal and vertical fragmentation with follow-on algebraic query optimization will only affect these subqueries.

3.3 A Query Processing Cost Model

In this work we are interested in fragmentation design to improve system performance by decreasing the total cost of query execution. Therefore, a cost model is needed to evaluate a fragmentation schema. We use the query tree to estimate query costs. We also need some suitable formulae to estimate the sizes of sets of complex values, fragments, and intermediate nodes of a query tree.

3.3.1 Query-Trees

Suppose we are given a query in query algebra. As discussed before we are not concerned with query optimization here as it is a challenging research topic of its own and beyond the scope of this thesis. The interested reader is referred to [2, 67, 93, 110]. Rather we assume that the query is already given in optimized form. In order to calculate query costs, we adapt query trees from [102]. A query tree is just a graph representation of a query.

The leaves of a query tree will be elementary queries such as instances $db(E)$ or fragments $db(F_i)$. All other nodes are operators of query algebra: σ_φ, π_t, ρ, $\bowtie$, $\cup, \cap$, or $-$ introduced in Section 3.2.3 . The query tree is formed inductively (inside-out) from Q as follows:

- If Q is elementary, then the tree consists of just one root, which is Q.
- For $Q = op(Q')$ with op standing for a selection, projection or renaming take the whole tree for Q' plus a new root op having the root of the Q'-tree as its successor.
- For $Q = Q_1 op\ Q_2$ with op standing for a join, a union, dererence or a difference take the trees for Q_1 and Q_2 plus a new root op having the roots of the Q_1-tree and the Q_2-tree as successors.

Example 3.9. Suppose there is a query requesting numbers and titles of papers offered in semester 0701 at PN campus. Then we have the following query:

$$\pi_{\text{no}}(\sigma_{\text{campus='PN'}\wedge\text{semester=0701}}(db(\text{Lecture}))) \bowtie \pi_{\text{no, ptitle}}(db(\text{Paper}))$$

The corresponding query tree is shown in Figure 3.2.

3.3.2 Size Estimation for Leaves of a Query Tree

Crucial to the query costs are the sizes of sets of complex values that have to be built during query execution, as these sets have to be stored at secondary storage, retrieved from there again, and sent between the locations of a network. Therefore, we first approach an estimation of these sizes.

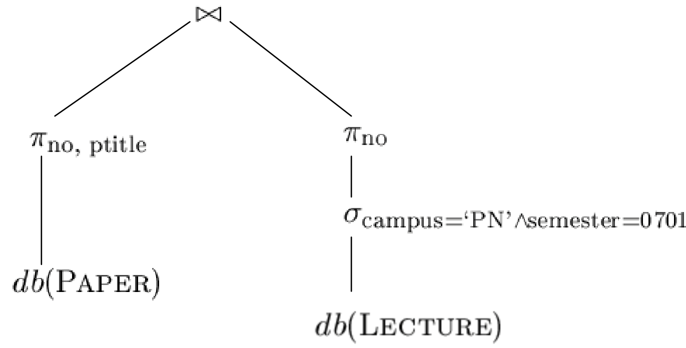

Fig. 3.2. Example of a Query Tree

In order to do so, we first look at types t, and estimate the size $s(t)$ of a value of type t, which of course depends on the context. Then the size of an instance $db(E)$ is $n_E \cdot s(t_E)$, where n_E is the average number of elements in the instance $db(E)$.

Let s_i be the average size of elements for a base type b_i. This can be used to determine the size $s(t)$ of an arbitrary type t, i.e. the average space needed for it in storage. We obtain:

$$s(t) = \begin{cases} s_i & \text{if } t = b_i \\ \sum_{i=1}^{n} s(t_i) & \text{if } t = (a : t_1, \dots, a_n : t_n) \\ r \cdot s(t') & \text{if } t = \{t'\} \text{ or } t = [t'] \end{cases}$$

In the last of these cases r is the average number of elements in sets or lists of type $\{t'\}$ or $t = [t']$, respectively, within a value of type t.

3.3.3 Size Calculation for Intermediate Nodes of a Query Tree

The calculation of sizes of instances of a type E applies also to the intermediate results of all queries as done in [75]. However, as we will see from the discussion in Chapter 5 and also in [81], which addresses the impact of fragmentation operations on query costs, we may restrict our attention to the nodes in the subqueries of the form $(*)$, as the other nodes in the query tree will not be affected by fragmentation and subsequent heuristic query optimization. Thus, we only have to look at selection and projection nodes, which are the only operations in the subqueries of the form $(*)$, and ignore all other nodes in query trees.

- The size of a selection node $\sigma_\varphi(db(E))$ is $p \cdot s$, where s is the size of the successor node $db(E)$ and p is the probability that an element in the successor will satisfy φ.
- The size of a projection node $\pi_{t'}(db(E))$ is $(1-c) \cdot s \cdot \frac{s(t')}{s(t)}$, where t is the type of elements in the set associated with the successor node $db(E)$ of size s, t' is the type of the elements in the set associated with the projection node, and c is the probability that two elements in the successor have the same projection.
- The size of a dereference node $\delta(db(E_1), db(E_2))$ is $\frac{s_1(s(t_1) + q(s(t_2) - s(k_{E_2})))}{s(t_1)}$, where s_i is the size of successor $db(E_i)$, t_i is the corresponding type – hence $\frac{s_i}{s(t_i)}$ is the average

number of elements in $db(E_i)$ – and q is the average number of identifiers in elements of $db(E_1)$ that occur in $db(E_2)$.

The work in [72] contains a discussion of sizes of results for other algebraic operations as well, but they will not be needed here because only subqueries in the form of $(*)$ are considered.

3.3.4 Query Processing Costs

Fragmentation results in a set of fragments $\{F_1, \dots, F_n\}$ of average sizes $s_1, \dots, s_n$. If the network has nodes $N_1, \dots, N_k$ we have to allocate these fragments to the nodes, which gives rise to a mapping $\lambda : \{1, \dots, n\} \rightarrow \{1, \dots, k\}$, which we call a *location assignment*.

However, the fragments only appear at the leaves of query trees. More generally, we must associate a node $\lambda(h)$ with each node h in each relevant query tree. $\lambda(h)$ indicates the node in the network, at which the intermediate query result corresponding to h will be stored.

Given a location assignment λ we can compute the total costs of query processing. Let the set of queries be $Q^m = \{Q_1, \dots, Q_m\}$. Query costs are composed of two parts: *storage costs* and *transportation costs*:

$$costs_\lambda(Q_j) = stor_\lambda(Q_j) + trans_\lambda(Q_j).$$

The storage costs give a measure for retrieving the data back from secondary storage, which is mainly determined by the size of the data. The transportation costs provide a measure for transporting between two nodes of the network.

The storage costs of a query Q_j depend on the size of the intermediate results and on the assigned locations, which decide the storage cost factors. It can be expressed as

$$stor_\lambda(Q_j) = \sum_h s(h) \cdot d_{\lambda(h)},$$

where h ranges over the nodes of the query tree for Q_j, $s(h)$ are the sizes of the involved sets, and d_i indicates the storage cost factor for node N_i $(i = 1, \dots, k)$.

The transportation costs of query Q_j depend on the sizes of the involved sets and on the assigned locations, which decide the transport cost factor between every pair of sites. It can be expressed by

$$trans_\lambda(Q_j) = \sum_h \sum_{h'} c_{\lambda(h')\lambda(h)} \cdot s(h').$$

Again the sum ranges over the nodes h of the query tree for Q_j, h' runs over the predecessors of h in the query tree, and c_{ij} is a transportation cost factor for data transport from node N_i to node N_j $(i, j \in \{1, \dots, k\})$.

Furthermore, for each query Q_j we assume a value for its frequency f_j. The total costs of all the queries in Q^m are the sum of the costs of each query multiplied by its frequency, i.e.

$$cost_\lambda = \sum_{j=1}^{m} cost_\lambda(Q_j) \cdot f_j.$$

Remark. Compared to the cost model in [64] the cost model introduced in this section is more general. It does not only consider transportation costs but also considers storage costs. Further, it does not only consider costs between fragments but also costs between intermediate nodes of a query tree, which may not be fragments. The cost model above does not need a fragment dependency graph as input but considers all the possible allocation schemata. Further, it is not restricted only to a simple query environment, which is assumed in [7, 39, 100].

Chapter 4

Fragmentation Operations

Let us now look at fragmentation. Similar to the relational approach, fragmentation exploits the fact that each database type E defines a set in a database db, thus can be partitioned into disjoint subsets. As the complex value data model defined in Chapter 3 provides deeply nested structures, splitting is used as another fragmentation operation. Roughly speaking, splitting introduces new data types referencing each other. In the sections below, horizontal fragmentation, vertical fragmentation and splitting will be defined, followed by correctness rules for fragmentation. Note that fragmentation operations considered here are very close to that defined for the RDM. There is no general decomposition and composition defined for complex value databases because inverse operations have not been defined for complex value databases. In particular, nesting and unnesting are not used when performing fragmentation because after unnesting a database instance of a complex database type we can not guarantee to reconstruct the original instance by using nesting [36].

4.1 Horizontal Fragmentation

Let us now define operations for horizontal fragmentation. Similar to the RDM horizontal fragmentation exploits the fact that each database type E defines a set $db(E)$ in a database db, thus can be partitioned into disjoint subsets.

There are two types of Horizonal fragmentation: primary horizontal fragmentation and derived fragmentation. Primary horizontal fragmentation refers to the fragmentation on database types using predicates.

Definition 4.1. *Primary horizontal fragmentation* on a database type E replaces E by a set $E_{H1}, \dots, E_{Hn}$ of new database types such that:

- the attribute and components in each E_{Hj} are the same as in E: $E_{Hj} = E$.
- each database instance over E can be split into pairwise disjoint instances $db(E_{H1}), \dots, db(E_{Hn})$ such that

$$db(E) = \bigcup_{i=1}^{n}(db(E_{Hi})), \qquad 1 \leq i \leq n.$$

- by using query algebra, the operation of horizontal fragmentation is expressed as:

$$db(E_{Hi}) = \sigma_{\varphi_i}(db(E)).$$

□

EXAMPLE 4.1. Let us consider an instance *db*(DEPARTMENT) (as outlined in Table 3.1) of DEPARTMENT from Example 3.1. Using two predicates $\varphi_1 \equiv$ contacts.contact.location $\ni$ 'Turitea' and $\varphi_2 \equiv$ contacts.contact.location $\not\ni$ 'Turitea', horizontal fragmentation of *db*(DEPARTMENT) into two fragments *db*(TURITEA_DEPARTMENT) and *db*(NO_TURITEA_DEPARTMENT) (as outlined in Table 4.1) are defined as follows:

$$db(\text{TURITEA_DEPARTMENT}) = \sigma_{\varphi_1}(db(\text{DEPARTMENT}))$$

$$db(\text{NO_TURITEA_DEPARTMENT}) = \sigma_{\varphi_2}(db(\text{DEPARTMENT}))$$

Table4.1. Horizontal Fragmentation of *db*(DEPARTMENT)

db(TURITEA_DEPARTMENT)			
dname	homepage	contacts	
		contact	
		location	office
Information	is.massey.ac.nz	Turitea	SST2.05
		Wellington	DLB305
Accounting	ac.massey.ac.nz	Turitea	SST1.14
		Albany	PK3.05

db(NO_TURITEA_DEPARTMENT)			
dname	homepage	contacts	
		contact	
		location	office
Marketing	mk.massey.ac.nz	Napier	BSC1.11
		Albany	TIU331

Derived horizontal fragmentation refers to performing fragmentation using semijoins with fragmentation of its component or a database type having it as a component.

Definition 4.2. *Derived horizontal fragmentation* on a database type E results that E is replaced by a set $\{E_{H1}, \ldots, E_{Hn}, E_{Hn+1}\}$ of new database types such that:

- the attributes and components in E_{Hi} are the same as in E: $E_{Hi} = E$.
- each instance over data type E can be split into pairwise disjoint instances $db(E_{H1}), \ldots, db(E_{Hn+1})$ such that

$$db(E) = \bigcup_{i=1}^{n+1}(db(E_{Hi})), \qquad 1 \leq i \leq (n+1).$$

- using query algebra derived horizontal fragmentation can be written as:

$$db(E_{Hi}) = db(E) \ltimes db(E'_i), \qquad 1 \leq i \leq n.$$

with $E' \in comp(E)$ or $E \in comp(E')$ and fragmentation schema of E' as $F_{E'} = \{E'_1, \ldots, E'_n\}$.

– there is always a remainder:

$$db(E_{Hn+1}) = db(E) - db(E) \ltimes db(E').$$

□

Note that we do not restrict ourself to performing derived horizontal fragmentation on a database type using horizontal fragmentation of only its components. This extension enables derived horizontal fragmentation to be applied in more general cases. Again, the biggest number of derived horizontal fragments are the number of network nodes.

In the complex data model introduced in Section 3.1, database types are on different levels. If a type is derived fragmented with the fragmentation schema of a type of its components, then the resulting fragments will be disjoint. If a type is fragmented using the fragmentation schema of a type which has it as a component, then the properties of disjointness cannot be guaranteed. However, if an extra procedure of removing overlaps is employed, disjointness can be guaranteed.

EXAMPLE 4.2. Take the schema from Example 3.1 and fragment the instance db(PAPER) into two new fragments db(ADVANCED_PAPER) and db(BASIC_PAPER) using $\varphi_1 \equiv$ level $\geq$ 300 and $\varphi_2 \equiv$ level $<$ 300 (see Table 4.2). Two fragments db(ADVANCED_LECTURE) and db(BASIC_LECTURE) (see Table 4.3) are defined as follows:

$$db(\text{ADVANCED_LECTURE}) = db(\text{LECTURE}) \ltimes db(\text{ADVANCED_PAPER})$$

$$db(\text{BASIC_LECTURE}) = db(\text{LECTURE}) \ltimes db(\text{BASIC_PAPER})$$

Table4.2. Horizontal Fragmentation of db(PAPER)

db(ADVANCED_PAPER)			
no	ptitle	level	points
157331	Database Concepts	300	15

db(BASIC_PAPER)			
no	ptitle	level	points
110001	Accounting Principles	100	15
110105	Taxation	100	15
156100	Marketing Management	100	15
157221	Information Systems Analysis	200	15

Note that database type LECTURE is derived fragmented using the fragmentation schema of its component PAPER. Therefore the disjointness criteria is satisfied because of the inclusion constraints between a database type and its component, i.e., t[PAPER](LECTURER) = t(PAPER).

4.2 Vertical Fragmentation

Definition 4.3. Taking a database type $E = (\{r_1 : E_1, \dots, r_n : E_n\}, \{a_1, \dots, a_k\}, \{r_{i_1} : E_{i_1}, \dots, r_{i_m} : E_{i_m}, a_{j_1}, \dots, a_{j_\ell}\})$, *vertical fragmentation* on E replaces E with a set of new types $E_{V1}, \dots, E_{Vm}$ with $E_{Vj} = (\{r_1^j : E_1^j, \dots, r_n^j : E_{n_j}^j\}, \{a_1^j, \dots, a_{n_j}^j\}, \{r_{i_1}^j : E_{i_1}^j, \dots, r_{i_m}^j : E_{i_m}^j, a_{j_1}^j, \dots, a_{j_\ell}^j\})$ such that:

Table4.3. Derived Horizontal Fragmentation of db(LECTURE)

db(ADVANCED_LECTURE)					
paper	semester	campus	schedule		
			slot		
no			day	time	room
157331	0601	PN	Monday	11am	SSLB3
			Thursday	2pm	SSLB2
157331	0701	PN	Monday	10am	SSLB4
			Thursday	3pm	SSLB4

db(BASIC_LECTURE)					
paper	semester	campus	schedule		
			slot		
no			day	time	room
110001	0401	ALB	Monday	10am	SSLB1
			Thursday	2pm	SSLB5
156100	0402	ALB	Wednesday	9am	BSC201
			Friday	1pm	BSC201
110105	0501	WN	Tuesday	1pm	SST1
			Thursday	3pm	SST1
157221	0501	WN	Monday	9am	SST1
			Wednesday	1pm	SST1

– the components and attributes will be distributed:

$$\{E_1, \ldots, E_n\} = \bigcup_{j=1}^{m} \{E_1^j, \ldots, E_{n_j}^j\},$$

$$\{a_1, \ldots, a_k\} = \bigcup_{j=1}^{m} \{a_1^j, \ldots, a_{n_j}^j\}.$$

– $db(E)$ is split into $db(E_{V1}), \ldots, db(E_{Vm})$ such that $db(E)$ could be reconstructed by using the join operation on all the instances:

$$db(E) = (\ldots((db(E_{V1}) \bowtie_{t_{1,2}} db(E_{V2})) \bowtie_{t_{1,2,3}} \ldots) \bowtie_{t_{1,\ldots,k}} db(E_{Vm})).$$

where $t_{1,\ldots,k} = (a_{x_1^i} : t_{x_1^i}, ..., a_{x_{l_i}^i} : t_{x_{l_i}^i})$ with $(a_{x_1^i}, \ldots, a_{x_{l_i}^i}) = (t_{E_1} \cup ... \cup t_{E_{i-1}}) \cap t_{E_i}$.

– Let $t_E = (a_1 : t_1, \ldots, a_n : t_n)$ be the representation type of E, $t_{E_{V1}}, \ldots, t_{E_{Vm}}$ be the corresponding representation types of $E_{V1}, \ldots, E_{Vm}$, with $t_{E_{Vj}} = (a_1^j : t_1^j, \ldots, a_{n_j}^j : t_{n_j}^j)$, which are supertypes of t_E. *Vertical fragmentation* of E is performed by projecting $db(E)$ to the set of supertypes $t_{E_{Vj}}$:

$$db(E_{Vj}) = \pi_{t_{E_{Vj}}}(db(E)), \text{for all } j \in \{1, \ldots, m\}.$$

□

To meet the criteria of reconstructivity, it is required that the key type k_E is part of all types E_{Vj}. Further, when projection is applied on attributes of a tuple type within a set type, e.g. $\{(a_{i1}, \dots, a_{in})\}$, an index I should be inserted as a surrogate attribute in the set constructor before fragmentation. This index should later be attached to each of the vertical fragments to ensure reconstructivity. However, no index is needed when performing vertical fragmentation on a list-valued attribute because two list-attributes, which are resulted from vertical fragmentation, are of the same length and can be joined on positions. Furthermore, for attribute a_j of a key type k_{E_j}, it is required that all attributes in k_{E_j} will be kept in together after vertical fragmentation.

Example 4.3. Take the schema from Example 3.1 and fragment the database type

Lecture = ({paper:Paper}, {semester, campus, schedule:{slot:(day, time, room)}})
into two new types, with one containing information about day and time of lectures and the other containing information of room of lectures. That means vertical fragmentation need to be performed on complex attribute 'schedule'. As there is an attribute 'slot' of a tuple type occurring inside the attribute 'schedule' of a set type, we need first to attach an index attribute to attribute 'slot'. Therefore, we alter the representation type t_{Lecture} to $t_{\text{Lecture}'}$ by attaching an index attribute as a sub-attribute of attribute 'slot'.

$t_{\text{Lecture}'}$ = (paper: (no: *CARD*), semester: *STRING*, schedule: {slot: (I: *CARD*, day: *STRING*, time *TIME*, room: *STRING*)});

Then vertical fragmentation is performed using two subtypes $t_{\text{Lecture_Time}}$ and $t_{\text{Lecture_Venue}}$, each containing a subset of the attributes of $t_{\text{Lecture}'}$, including the index attribute and the key attributes:

$t_{\text{Lecture_Venue}}$ = (paper: (no: *CARD*), semester: *STRING*, schedule: {slot: (I: *CARD*, room: *STRING*)});

$t_{\text{Lecture_Time}}$ = (paper: (no : *CARD*), semester: *STRING*, schedule: {slot: (I: *CARD* day: *STRING*, time: *TIME*,)}).

Accordingly, we get the two vertical fragments that result from project operations:

$$db(\text{Lecture_Time}) = \pi_{t_{\text{Lecture_Time}}}(db(\text{Lecture})),$$

$$db(\text{Lecture_Venue}) = \pi_{t_{\text{Lecture_Venue}}}(db(\text{Lecture})).$$

Analogously, the instances db(Lecture) and the resulting fragments are shown in Table 4.4.

4.3 Splitting

For *fragmentation by splitting* suppose the schema contains a database type E, and let t be a type that occurs inside t_E. Then create a database type D with $t_D = t$. Furthermore, replace E by E' with $t_{E'}$ resulting from t_E by replacing t by the key type k_D of D such that for each database db we obtain

$$db(E) = \delta(db(E'), db(D)).$$

The splitting operation is quite simple. Suppose the schema contains a database type E and take subsets $comp(D) \subseteq comp(E)$ and $attr(D) \subseteq attr(E)$. Then simply add a new database type $D = (comp(D), attr(D), id(D))$ to the schema $\mathcal{S}$ and change the definition of E to

$$E' = (comp(E) - comp(D) \cup \{r : D\}, attr(E) - attr(D), id(E'))$$

Table4.4. Vertical Fragmentation of db(Lecture)

db(Lecture_Time)					
paper	semester	campus	schedule		
			slot		
no			I	day	time
110001	0401	ALB	1	Monday	10am
			2	Thursday	2pm
156100	0402	ALB	1	Wednesday	9am
			2	Friday	1pm
110105	0501	WN	1	Tuesday	1pm
			2	Thursday	3pm
157221	0501	WN	1	Monday	9am
			2	Wednesday	1pm
157221	0601	PN	1	Monday	11am
			2	Wednesday	2pm
157331	0601	PN	1	Monday	11am
			2	Thursday	2pm
157331	0701	PN	1	Monday	10am
			2	Thursday	3pm

db(Lecture_Venue)				
paper	semester	campus	schedule	
			slot	
no			I	room
110001	0401	ALB	1	SSLB1
			2	SSLB5
156100	0402	ALB	1	BSC201
			2	BSC201
110105	0501	WN	1	SST1
			2	SST1
157221	0501	WN	1	SST1
			2	SST1
157221	0601	PN	1	SSLB2
			2	SSLB2
157331	0601	PN	1	SSLB3
			2	SSLB2
157331	0701	PN	1	SSLB4
			2	SSLB4

with $id(E') = ((id(E) - comp(D)) - attr(D)) \cup \{r : D\}$ if $id(E) \cap (comp(D) \cup attr(D)) \neq \emptyset$, $id(E') = id(E)$ otherwise, and $id(D) = comp(D) \cup attr(D)$.

Then $db(E)$ is reconstructed from $db(E')$ and $db(D)$ by a dereference operation δ.

Example 4.4. We could split the database type Lecture from 3.1 into the following two types:

Paper_Offer = ({paper:Paper}, {semester, campus}, {paper:Paper, semester, campus})
Lecture′ = ({for:Paper_Offer}, {schedule}, {for:Paper_Offer})

□

If we first apply splitting, we increase the number of database types in the schema. Thus, splitting widens the possibilities of applying horizontal or vertical fragmentation.

4.4 Correctness Rules for Fragmentation

As with performing fragmentation for the relational data model and the object-oriented data model, fragmentation should not lead to any loss of or addition to data. Adopted from [93] to complex value databases, the following three correctness rules during fragmentation should be enforced to make sure that no semantic change occurs during fragmentation.

1. **Completeness**
   The completeness of fragmentation requires that there should not be any loss of information after fragmentation.
2. **Reconstruction**
   If an instance $db(E)$ of a database type E is fragmented into fragments $db(E_1), \ldots, db(E_n)$, it should be possible to use query operation Ω to reconstruct $db(E)$ with $db(E_i)$ such that

$$db(E) = \Omega_{i=1}^{n} db(E_i)$$

For different forms of fragmentation, the operator Ω will be different.

- For horizontal fragments, reconstruction of a global instance $db(E)$ is performed by using the union operation in the horizontal fragmentation:

$$db(E) = \bigcup_{i=1}^{n} db(E_i).$$

- For vertical fragments, reconstruction of the original global instance $db(E)$ is made by using the join operation:

$$db(E) = (\ldots((db(E_1) \bowtie_{t_{1,2}} db(E_2)) \bowtie_{t_{1,2,3}} \ldots) \bowtie_{t_{1,\ldots,k}} db(E_n)).$$

- For splitting fragments, $db(E')$ and $db(D)$, reconstruction of the original global instance $db(E)$ is made by using dereference operation:

$$db(E) = \delta(db(E'),\ db(D)).$$

3. **Disjointness**
Informally, disjointness refers to the fact that fragmentation should not lead to a replication of information. But if replication comes into play, this requirement has to be relaxed. If an instance $db(E)$ is horizontally decomposed into fragments $db(E_1), \ldots db(E_n)$ and the data item d_i is in $db(E_i)$ with $1 \leq i \leq n$, then it does not appear in any other fragment $db(E_j)(i \neq j)$ with $1 \leq j \leq n$. If a database instance $db(E)$ is vertically partitioned, disjointness is defined only on the attributes that are not consisted in the key type k_E of a database type E. To simplify the problem, replication is not considered at the first stage of the design of distributed databases.

Chapter 5

Foundations of Fragmentation and Allocation

The interdependencies between the problem of database distribution design and the problem of distributed query optimization plays an important role in an optimal database distribution design. Distributed query optimization, which results in a distributed query processing schedule, is interrelated to fragment allocation. Fragment allocation depends on the processing schedules of all the queries that access the fragments. This gives rise to a circular problem which has been noted in [7, 100]. Further, fragment allocation depends on fragmentation and distributed query optimization. However, it is practically impossible to study database distribution design together with distributed query optimization because of the complexity of each of the problems. The discussion of distributed query optimization is outside of the scope of this thesis. To obtain a semi-optimal fragmentation and allocation schema we need to simplify the combined problems with some assumptions. We assume that distributed queries have been optimized before fragmentation with query decomposition rules and centralized query optimization rules. After fragmentation distributed queries are transferred into fragment queries by substituting each data type in the query trees by the dereferences, unions, or joins of the fragments, the data of which are accessed, reduction techniques are then used to generate simpler and optimized queries [93]. This is important because it reduces the irrelevant data accesses by replacing dereferences, unions, or join of all the fragments with the dereferences, unions, or join of only the fragments needed by the query. In this chapter I will discuss the impact of fragmentation and allocation on query costs with the aim of isolating two interrelated problems, database distribution design and query processing and optimization.

5.1 The Impact of Splitting on Query Costs

The aim of database distribution is to decrease overall query costs. So assume we have a query Q_j that involves subqueries of the form $\pi_t(\sigma_\varphi(db(E)))$ for database type E. For such subqueries we can always assume that an optimal location assignment will choose the same network nodes for π_t, σ_φ and $db(E)$, as the two unary operations will only reduce the size. In this section I discuss the impact of splitting $db(E)$ on the query costs of Q_j. We will investigate three scenarios for this.

5.1.1 Scenario I

After an instance $db(E)$ of database type E is split into two instances $db(E_1)$ and $db(E_2)$ with E_1 referencing E_2, a subquery $\pi_t(\sigma_\varphi(db(E)))$ has to be replaced by $\pi_t(\sigma_\varphi(\delta(db(E_1), db(E_2))))$ in general. However, it may well be the case that only the fragment $db(E_1)$ is needed for the result, in which case we could replace the query by $\pi_t(\sigma_{\varphi'}(db(E_1)))$ using a modified selection formula φ'.

Therefore, the leaf $db(E)$ would be replaced by the fragment $db(E_1)$ arising from the splitting. As the result of the whole subquery remains unchanged, any optimal location assignment would remain optimal, if only the node N_a associated with π_t, σ_φ and $db(E)$ (i.e. the node on which $db(E)$ resides) would be changed to N_b, the node associated with π_t, $\sigma_{\varphi'}$ and $db(E_1)$ in an optimal allocation of $db(E_1)$. The modified query tree of Q_j is shown in Figure 5.1.

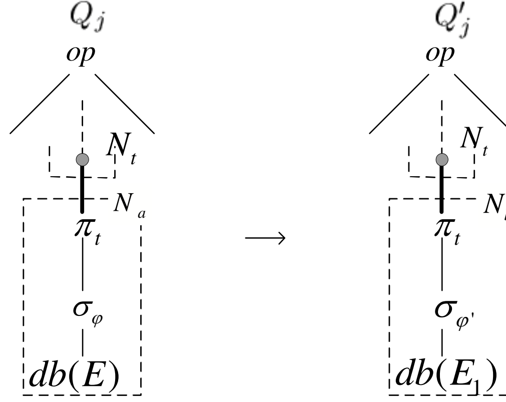

Fig. 5.1. Scenario I for Query Tree Rewriting in Case of Splitting Fragmentation

Assume that the storage cost factors d_i for all sites among the network are equal. Then the effect of splitting is that the storage cost is reduced by $s_E - s_{E_1} + (1-p) \cdot s_{E_1}$ with the factor p corresponding to the selection σ_φ. Hence, we get

$$stor(Q'_j) = stor(Q_j) - s_E + p \cdot s_{E_1}.$$

Assume the predecessor of π_t in the query tree of Q_j is allocated to node N_t as shown in Figure 5.1. Let s denote the size of $\pi_t(\sigma_\varphi(db(E)))$. If the fragment $db(E_1)$ is allocated to node $N_b \neq N_t$, then the transportation cost for Q_j is reduced by $(c_{at} - c_{bt}) \cdot s$, i.e. we get

$$trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{bt} \cdot s.$$

In the extreme case that $db(E_1)$ is allocated to $N_b = N_t$, the transportation cost for Q_j is reduced the most, i.e. by $c_{at} \cdot s$. In this case we get

$$trans(Q'_j) = trans(Q_j) - c_{at} \cdot s.$$

5.1.2 Scenario II

While scenario I is a rather special case, most queries would still need to access both fragments $db(E_1)$ and $db(E_2)$, in which case the subquery of query Q_j would be replaced by $\pi_t(\sigma_\varphi(\delta(db(E_1), db(E_2))))$. In this case the following query optimization rules can be applied:

- Replace $\sigma_\varphi(\delta(db(E_1), db(E_2)))$ by $\delta(\sigma_\varphi(db(E_1)), db(E_2))$, if the selection only depends on $db(E_1)$.
- Replace $\sigma_\varphi(\delta(db(E_1), db(E_2)))$ by $\delta(db(E_1), \sigma_{\varphi'}(db(E_2)))$, if the selection only depends on $db(E_2)$. In this case, however, the selection formula φ has to be modified to φ' due to the fact that the type t_{E_2} is embedded in t_E.
- In the general case replace $\sigma_\varphi(\delta(db(E_1), db(E_2)))$ by $\delta(\sigma_{\varphi_1}(db(E_1)), \sigma_{\varphi_2}(db(E_2)))$ with $\varphi \Leftrightarrow \varphi_1 \wedge \varphi_2$ and φ_1 depends only on $db(E_1)$ and φ_2 depends only on $db(E_2)$.
- Replace $\pi_t(\delta(db(E_1), db(E_2)))$ by $\delta(\pi_{t_1}(db(E_1)), \pi_{t_2}(db(E_2)))$ with $t_i = (t \cap t_{E_i}) \cup k_{E_2}$ for $i = 1, 2$.

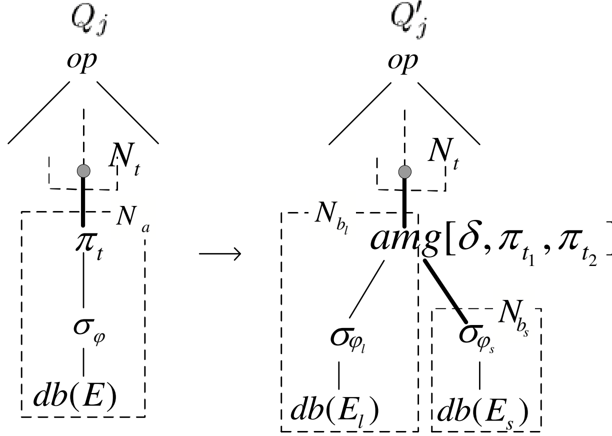

Fig. 5.2. Scenario II for Query Tree Rewriting in case of Splitting Fragmentation

Query optimization might shift the selection σ_φ and the projection π_t inside the newly introduced dereference operation. In general, we would get a subquery of the form $\delta(\pi_{t_1}(\sigma_{\varphi_1}(db(E_1))),\ \pi_{t_2}(\sigma_{\varphi_2}(db(E_2))))$. In this subquery we could further replace the dereferencing and the projections by an amalgam $amg[\delta, \pi_{t_1}, \pi_{t_2}]$. However, as in scenario I, the upper part of the query tree of Q_j would not be affected. Figure 5.2 illustrates the query trees before and after $db(E)$ is split.

Then the storage cost of the modified query Q'_j is

$$stor(Q'_j) = stor(Q_j) - s_C - p \cdot s_C + p_1 \cdot s_{E_1} + s_{E_1} + p_2 \cdot s_{E_2} + s_{E_2}.$$

Let b_i be the node to which $db(E_i)$ is allocated $(i = 1, 2)$. Let $\ell, s \in \{1, 2\}$ be chosen such that the size of $\sigma_{\varphi_\ell}(db(E_\ell))$ is larger than (or equal to) $\sigma_{\varphi_s}(db(E_s))$. Then the effect of the splitting on the transportation costs is as follows:

- If $N_{b_s} \neq N_t$, then $trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_\ell t} \cdot s + c_{b_s b_\ell} \cdot p_s \cdot s_{E_s}$.
- If $N_{b_s} = N_t$, then $trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_\ell b_s} \cdot p_\ell \cdot s_{E_\ell}$.

5.1.3 Scenario III

Basically, we make the same assumptions as for scenario II, but in addition assume that the sizes of $\sigma_{\varphi_i}(db(E_i))$ are almost equal for $i = 1, 2$. In this case it is advantageous to transport both intermediate results to the same node N_t and to perform the amalgam-operation $amg[\delta, \pi_{t_1}, \pi_{t_2}]$ at that node. The optimized query tree is illustrated in Figure 5.3.

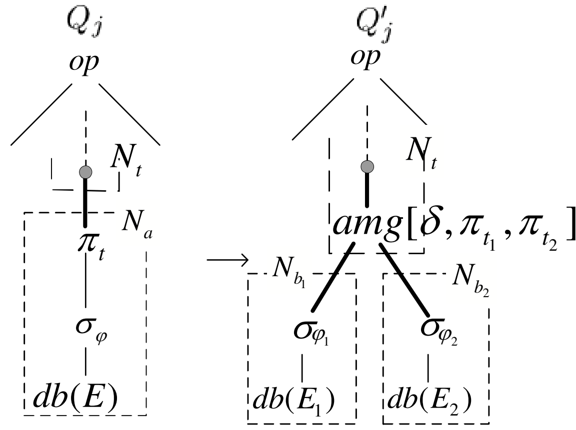

Fig. 5.3. Scenario III for Query Tree Rewriting in Case of Splitting Fragmentation

If $N_{b_1} \neq N_t$ and $N_{b_2} \neq N_t$, then storage costs are the same as in Scenario II. Changes only occur in the transportation costs of query Q'_j:

$$trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_1 t} \cdot p_1 \cdot s_{E_1} + c_{b_2 t} \cdot p_2 \cdot s_{E_2}.$$

Fact 5.1 summarizes the discussion about fragmentation by splitting.

Fact 5.1 *Let Q be an optimized query and λ an optimal allocation of network nodes to the nodes in the query tree of Q. If the leaf $db(E)$ is fragmented into $db(E_1)$ and $db(E_2)$ by splitting, then an optimal allocation for the resulting query tree will at most change the allocation of the two predecessors of $db(E)$ labeled by a selection σ_φ and a projection π_t. This is because all selection and projection operations in the query tree have been pushed down to the leaf $db(E)$ by query optimization before the fragmentation. After splitting, a dereference operation is introduced above the leaves. Then another round of query optimization might only shift the selection and projection inside the newly introduced dereference operation. The upper part of the query tree does not change.*

5.2 The Impact of Horizontal Fragmentation on Query Costs

Let us now look at the effects of horizontal fragmentation with a single simple selection formula (and its negation) to query costs. Consider again a query Q_j and a subquery of the form $\pi_t(\sigma_\varphi(db(E)))$. As for the case of splitting we distinguish three basic scenarios.

5.2.1 Scenario I

Assume that the selection formula φ in $(*)$ has the form $\varphi = \psi \wedge \omega$ and that we use ψ to fragment $db(E)$ into $db(E_1)$ and $db(E_2)$, i.e. $db(E_1) = \sigma_\psi(db(E))$ and $db(E_2) = \sigma_{\neg\psi}(db(E))$.

Then the subquery $(*)$ of Q_j only needs to access the fragment $db(E_1)$ and thus can be rewritten as $\pi_t(\sigma_\omega(db(E_1)))$. So, the leaf $db(E)$ would be replaced by the fragment $db(E_1)$ of $db(E)$ that is determined by ψ, and its predecessor in the query tree would become σ_ω – we neglect the special case that ω is $true$, in which case the selection would completely disappear.

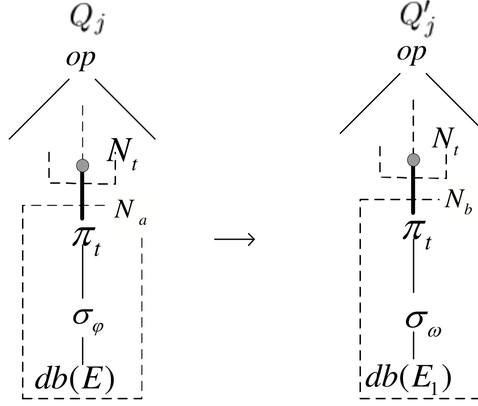

Fig. 5.4. Scenario I for Query Tree Rewriting in Case of Horizontal Fragmentation

As a consequence, the fragmentation can only require the change of the assigned network node from N_a, i.e. the node for π_t, σ_φ and E, to some N_b. This corresponds to the allocation of the fragment E_1. This change is shown in Figure 5.4.

Since the result of the whole subquery remains unchanged, there is no further change to the query tree. Let p be the possibility that the objects in $db(E)$ satisfy the condition ψ then we have $s_{E_1} = p \cdot s_E$. If we know the size of selection node $\sigma_\omega(db(E_1))$ or the size of selection node $\sigma_{\omega\wedge\psi}(db(E))$, then we obtain that one leaf node is reduced in size by factor p, and one internal node may be reduced to 0, i.e. not exist anymore.

Assume the storage cost factors d_i are equal for all network nodes. Then the effect on storage costs is:

$$stor(Q'_j) = stor(Q_j) - (1-p) \cdot s_E$$

If s is the size of $\pi_t(\sigma_\varphi(db(E)))$ and the processor π_t is allocated to node N_t, then the transport cost is only affected if E_1 is allocated to a site b that is different from site a to which E was allocated. Then the transportation cost is changed from $s \cdot c_{at}$ to $s \cdot c_{bt}$. Hence we get:

$$trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{bt} \cdot s$$

5.2.2 Scenario II

Assume now that fragmentation with a selection formula ψ' leads to two fragments $db(E_1)$ and $db(E_2)$, i.e. the query Q_j requires data from both fragments. Then the subquery $(*)$ would be replaced by

$$\pi_t(\sigma_\varphi(db(E_1) \cup db(E_2))). \qquad (\dagger)$$

Further algebraic query optimization will move the selection and projection inside the union, i.e. we obtain

$$\pi_t(\sigma_\varphi(db(E_1))) \cup \pi_t(\sigma_\varphi(db(E_2))). \qquad (\ddagger)$$

As a consequence, the fragmentation can only require that the network nodes assigned to the two new fragments are different from each other and maybe also different from the target network nodes N_t. Then the fragment with the larger size after selection will determine the location assignment for the union and the projection node, unless the fragment with the

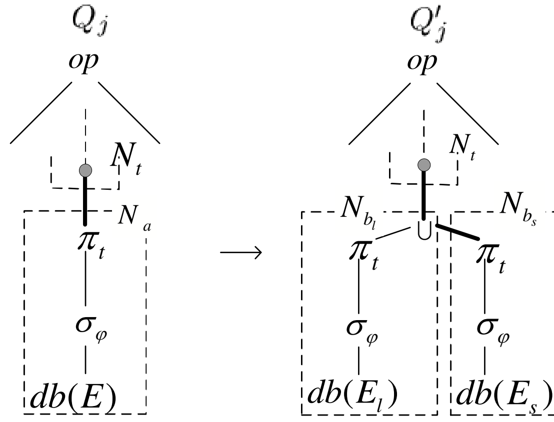

Fig. 5.5. Scenario II for Query tree Rewriting in Case of Horizontal Fragmentation

smaller size after selection is allocated to the target node N_t, in which case the larger fragment should be moved to N_t. Figure 5.5 shows the changes of query tree with site allocation.

After the horizontal fragmentation the effect on the storage costs is that only a union node is added, i.e.

$$stor(Q'_j) = stor(Q_j) + s$$

with the size s of the result of the whole subquery $\pi_t(\sigma_\varphi(db(E)))$.

Let N_{b_i} be the node to which E_i is allocated ($i = 1, 2$). Let $\ell, s \in \{1, 2\}$ such that the size s_s of $\pi_t(\sigma_\varphi(db(E_s)))$ is not larger than the size s_ℓ of $\pi_t(\sigma_\varphi(db(E_\ell)))$. Then the effect of the fragmentation on the transportation costs is as follows:

- If $N_s \neq N_t$, then $trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_\ell t} \cdot s + c_{b_s b_\ell} \cdot s_s$.
- If $N_s = N_t$, then $trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_\ell b_s} \cdot s_\ell$.

5.2.3 Scenario III

The assumption for the third scenario is the same as for scenario II, but in addition we now assume that the sizes of $\sigma_\varphi(db(E_i))$ are almost equal for $i = 1, 2$, and that the projection has only a small impact on the size of the result of (†). Then we get $s_1 \approx s_2 \approx \frac{s_E}{2}$.

In this case it is advantageous to transport both fragments (after selection with φ) to the same node N_t, at which the projection operation will be performed together with the union of the two fragments. We may even assume that the union and the follow-on projection can be combined in an amalgam $amg[\pi_t, \cup]$, as both operations would use sorting. The result on the query tree is illustrated in Figure 5.6.

As in the two other cases, we may neglect the changes to the storage costs. For the transport costs we get

$$trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_1 t} \cdot p \cdot s_{E_1} + c_{b_2 t} \cdot p \cdot s_{E_2} \approx trans(Q_j) + (\frac{c_{b_1 t} + c_{b_2 t}}{2} - c_{at}) \cdot s.$$

Fact 5.2 summarizes the discussion about horizontal fragmentation.

Fact 5.2 *Let Q be an optimized query and λ an optimal allocation of network nodes to the nodes in the query tree of Q. If the leaf $db(E)$ is horizontally fragmented into $db(E_1)$*

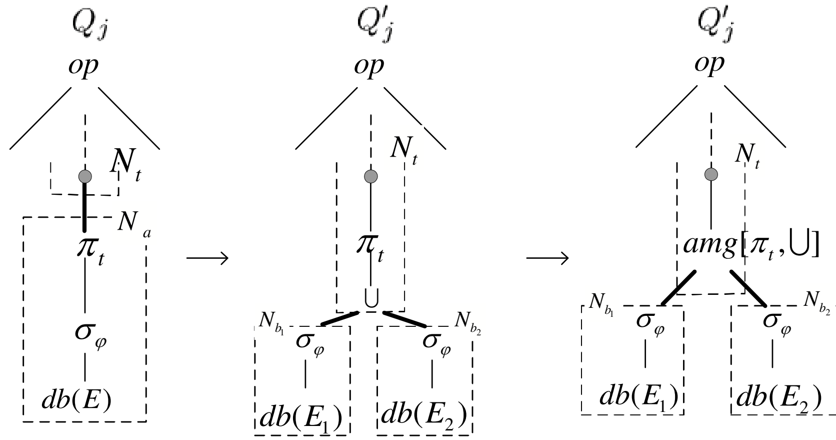

Fig. 5.6. Scenario III for Query Tree Rewriting in Case of Horizontal Fragmentation

and $db(E_2)$, then an optimal allocation for the resulting query tree will at most change the allocation of the two predecessors of $db(E)$ labeled by a selection σ_φ and a projection π_t. This is because all selection and projection operations in the query tree have been pushed down to the leaf $db(E)$ by query optimization before the fragmentation. After horizontal fragmentation, a union operation is introduced above the leaves. Then another round of query optimization might only shift the selection and projection inside the newly introduced union operation. The upper part of the query tree does not change.

5.3 The Impact of Vertical Fragmentation on Query Costs

Assume now that $db(E)$ is vertically fragmented into two fragments $db(E_1)$ and $db(E_2)$ with $db(E_i) = \pi_{t_i}(db(E))$. In general, the impact of vertical fragmentation on query Q_j would be that $db(E)$ in the query tree would be replaced by the join of the two fragments, i.e. $db(E_1) \bowtie_t db(E_2)$.

The algebraic query optimization may replace the subquery $\pi_{t'}(\sigma_\varphi(db(E)))$ by

$$\pi_{t'}(\sigma_\psi(\pi_{t_1}(\sigma_{\varphi_1}(db(E_1))) \bowtie_t \pi_{t_2}(\sigma_{\varphi_2}(db(E_2)))))$$

and then by

$$amg[\pi_{t'}, \sigma_\psi, \bowtie_t](\pi_{t_1}(\sigma_{\varphi_1}(db(E_1))), \pi_{t_2}(\sigma_{\varphi_2}(db(E_2))))$$

or

$$amg[\pi_{t'}, \sigma_\psi, \bowtie_t](\sigma_{\varphi_1}(db(E_1)), \sigma_{\varphi_2}(db(E_2)))$$

in case the projection cannot be moved inside the join. The impact of vertical fragmentation on query costs is discussed in the following three scenarios. Note that after vertical fragmentation, another round query optimization does not consider all the possible changes, i.e., join-order, semi-join programs, etc. are not considered here.

5.3.1 Scenario I

Let us first consider the special case that only one of the fragments is needed for the subquery, i.e. the subquery becomes $\pi_{t'}(\sigma_{\varphi_i}(db(E_i)))$. Assume that E_i is allocated to node N_b and that

this is different from node N_t, i.e. the node associated with the predecessor of $\pi_{t'}$ in the query tree. The query tree is changed as shown in Figure 5.7.

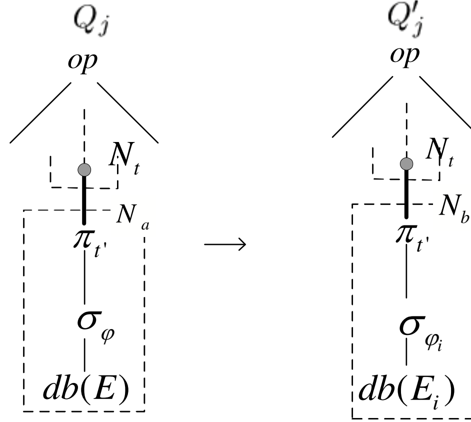

Fig. 5.7. Scenario I for Query Tree Rewriting in Case of Vertical Fragmentation

The storage costs of query Q_j are reduced by $(1+p) \cdot s(E) + (1+p_i) \cdot s(E_i)$, i.e. we have

$$stor(Q'_j) = stor(Q_j) - (1+p) \cdot s(E) + (1+p_i) \cdot s(E_i).$$

Transportation costs are reduced by $(c_{at} - c_{bt}) \cdot s$, so we have

$$trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{bt} \cdot s.$$

We observe that in case $db(E_i)$ being allocated to site N_t transportation costs of query Q_j can even be reduced by $c_{at} \cdot s$, which gives $transQ'_j = trans(Q_j) - c_{at} \cdot s$.

5.3.2 Scenario II

In the general case the query Q_j needs access to both fragments $db(E_1)$ and $db(E_2)$. In this case let $\ell, s \in \{1,2\}$ such that the size of $\pi_{t_\ell}(\sigma_{\varphi_\ell}(db(E_\ell)))$ is larger than (or equal to) the size of $\pi_{t_s}(\sigma_{\varphi_s}(db(E_s)))$. Hence it will be advantageous to transport the smaller one of these intermediate results to the node N_{b_ℓ} of the larger one, then execute the amalgam operation at node N_{b_ℓ}, and finally transport the result to node N_t to complete the processing of Q_j. Figure 5.8 shows how the query tree is changed.

Let s_{π_i} indicate the size of $\pi_{t_i}(\sigma_{\varphi_i}(db(E_i)))$ for $i = 1, 2$. Then the storage costs of the amended query Q'_j are

$$stor(Q'_j) = stor(Q_j) + s_{\pi_1} + p_1 \cdot s_{E_1} + s_{E_1} + s_{\pi_2} + p_2 \cdot s_{E_2} + s_{E_2} - p \cdot s_E - s_E.$$

Assume that $db(E_i)$ is allocated to N_{b_i}. Then the effect of vertical fragmentation on the transportation costs is as follows:

- If $N_{b_s} \neq N_t$, then $trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_\ell t} \cdot s + c_{b_s b_\ell} \cdot s_s$.
- If $N_{b_s} = N_t$, then $trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_\ell b_s} \cdot s_\ell$.

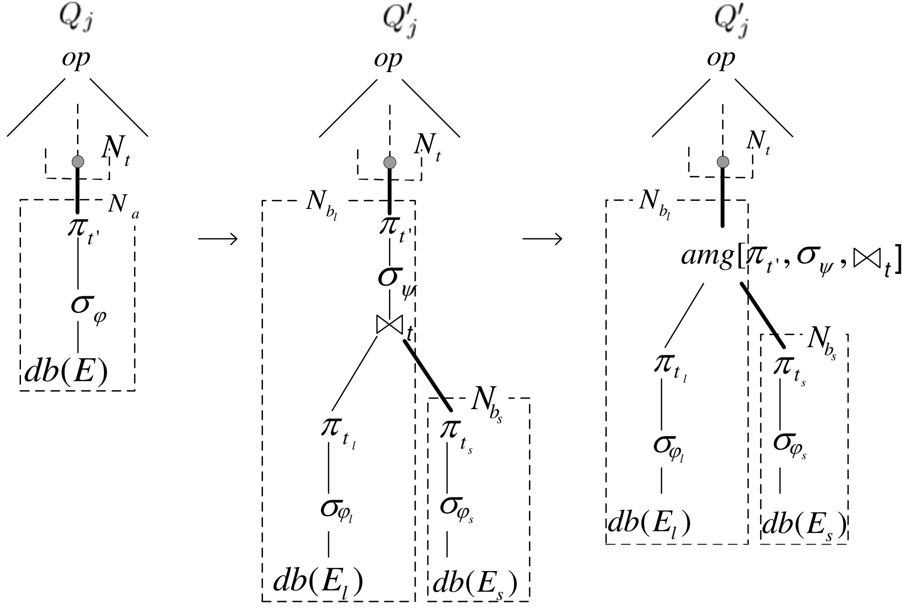

Fig. 5.8. Scenario II for Query Tree Rewriting in Case of Vertical Fragmentation

5.3.3 Scenario III

Now assume that, in addition to the assumptions made in scenario II, the sizes of $\pi_{t_i}(\sigma_{\varphi_i}(db(E_i)))$ for $i = 1, 2$ are almost equal. In this case it is advantageous to transport both these intermediate results to the target node N_t and to perform the amalgam operation at that node. The optimized distributed query tree is illustrated in Figure 5.9.

If $N_{b_1} \neq N_t$ and $N_{b_2} \neq N_t$ storage costs will be the same as discussed in scenario II. The transportation costs of query Q'_j are changed to

$$trans(Q'_j) = trans(Q_j) - c_{at} \cdot s + c_{b_1 t} \cdot s_{\pi_1} + c_{b_2 t} \cdot s_{\pi_2}.$$

The following fact summarizes our discussion for the case of vertical fragmentation.

Fact 5.3 *Let Q be an optimized query and λ an optimal allocation of network nodes to the nodes in the query tree of Q. If the leaf $db(E)$ is vertically fragmented into $db(E_1)$ and $db(E_2)$ over schema E_1, E_2, respectively, then an optimal allocation for the resulting query tree will at most change the allocation of the two predecessors of $db(E)$ labeled by a selection σ_φ and a projection π_t. This is because all selection and projection operations in the query tree have been pushed down to the leaf $db(E)$ by query optimization before the fragmentation. After vertical fragmentation, a join operation is introduced above the leaves. Then another round of query optimization might only shift the selection and projection inside the newly introduced join operation. The upper part of the query tree does not change.*

5.4 Summary

With the discussions above showing how fragmentation operations affect query costs for an optimized query with optimal allocation of network to nodes, a conclusion can be drawn

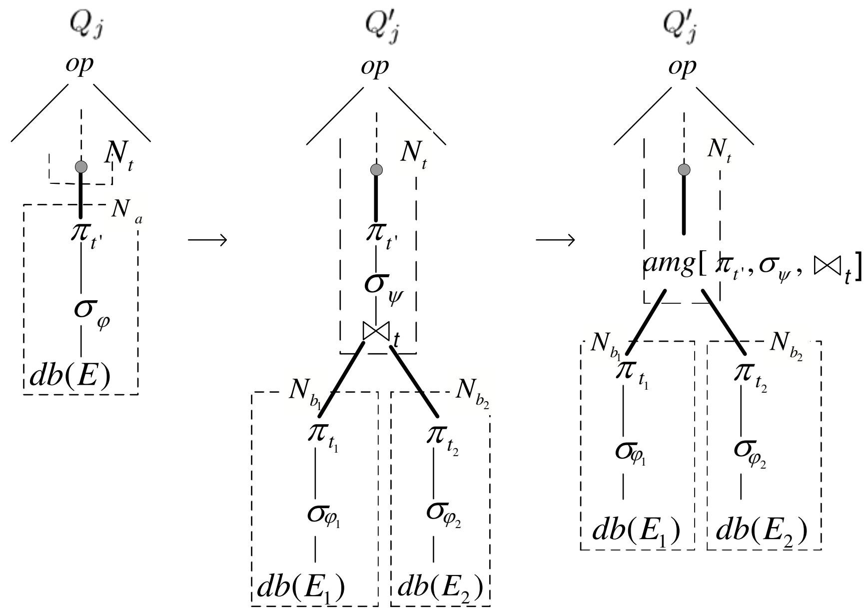

Fig. 5.9. Scenario III for Query Tree Rewriting in Case of Vertical Fragmentation

that the problem of distributed query optimization can be isolated from the problem of fragmentation and allocation by considering the subqueries with the form $\pi_t(\sigma_\varphi(db(E)))$. In addition, it is observed that the impact of fragmentation on query trees is of little relevance to the data model considered. The change of data models results in only marginal changes to query trees and intermediate nodes.

Chapter 6

Heuristics for Horizontal Fragmentation and Allocation

In this chapter we investigate how to perform horizontal fragmentation based on the cost model presented in Chapter 3. The general goal of fragmentation and allocation is to optimize total query processing costs. For this we adopt the recommended rule of thumb to consider only the 20% most frequent queries, as these usually account for most of the data access [93]. As before let $Q^m = \{Q_1, \ldots, Q_m\}$ denote the set of these queries. Then we can base decisions about which fragments to build and where to allocate them on the static analysis of these selected queries using the results from the previous chapter. In this chapter I will concentrate on horizontal fragments. Based on the discussion of query costs in the preceding chapter I now investigate three problems:

- Assuming that a set of horizontal fragments are given, the allocation of these fragments to network nodes will be addressed. For this a simple naive algorithm will be presented.
- Assuming that an allocation of fragments is given, if another simple selection predicate is taken into account how this allocation would change will be addressed. In particular, this will tell us whether this additional fragmentation would make sense, i.e. further decrease query processing costs.
- Assuming that we start from scratch, we address which selection formulae should be considered for fragmentation. For this a heuristic algorithm will be presented.

Some work of this chapter can be found in the literature. [75] uses an ad hoc heuristic approach to achieve an optimized fragmentation for object-oriented databases, which has been refined in [76, 77, 81] by taking a closer look at the impact of fragmentation on optimized query trees and allocation heuristics.

6.1 Primary Horizontal Fragmentation and Allocation

We are particularly interested in those horizontal fragments that arise from the simple selection predicates that are involved in the selected set of queries Q^m. So let us assume that Φ^w contains those predicates, i.e. each φ_i appears inside a subquery of the form $(*)$ in some $Q_j \in Q^m$. According to our analysis in the preceding section – summarized in Facts 5.1 - 5.3 – we may concentrate on these subqueries.

The first step in the design of fragmentation is acquiring application information. With application information we can retrieve a set of simple predicates. Taking into consideration complex value databases, *simple selection predicates* (or *simple predicates* for short) are different from the predicates defined in the context of the relational data model.

Definition 6.1. *simple selection predicates* φ_i take the form

$$path\ \ \theta\ \ v$$

with a path expression *path* as defined in Section 3.2.2, a value of the corresponding type, and a comparison operator θ, which can be one of $=, \neq, \leq, <, \geq, >, \supseteq, \subset, \supseteq, \supset, \in, \ni, \notin$, and $\not\ni$. □

Definition 6.2. Let $\Phi^w = \{\varphi_1, \ldots, \varphi_w\}$ denote a set of simple selection predicates defined on database type E. Then the set of *normal predicates* $\mathcal{N}^m = \{\mathcal{N}_1, \ldots, \mathcal{N}_n\}$ over Φ^w is the set of all satisfiable predicates of the form

$$\mathcal{N}_g \equiv \varphi_1^* \wedge \cdots \wedge \varphi_w^*,$$

where φ_i^* is either φ_i or $\neg\varphi_i$, $1 \leq i \leq w$ and $1 \leq g \leq 2^w$. □

Of course, the disjointness property implies that we must have

$$\mathcal{N}_j \wedge \mathcal{N}_k \Leftrightarrow false \quad \text{for all} \quad j \neq k.$$

Similarly, the completeness property implies the requirement that

$$\bigvee_{g=1}^{2^w} \mathcal{N}_g \Leftrightarrow true$$

Definition 6.3. An *atomic horizontal fragment* F_g of database type E is a fragment that is defined by a normal predicate over E:

$$db(F_g) = \sigma_{\mathcal{N}_g}(db(E)), \qquad 1 \leq g \leq 2^w.$$

□

Normal predicates, which are the conjunctions of simple predicates either in their positive or negative form, can be represented in the following form:

$$\mathcal{N}_g = \bigwedge_{i \in J} \varphi_i \wedge \bigwedge_{i \notin J} \neg\varphi_i.$$

with $J \subseteq \{1, \ldots, w\}$ as a set of indices of a subset of all simple predicates. Let f_i be the frequency of predicate φ_i, $J_\theta = \{i | i \in J \wedge \varphi_i \text{ executed at site } \theta\}$ be a subset of indices of all simple predicates, executed at site N_θ.

Definition 6.4. Let $P^M = \{P_1, \ldots, P_M\}$ denote the set of elementary queries of the form $\pi_t(\sigma_\varphi(db(E)))$. Then the frequency f_j of query P_j is the sum of the frequencies of queries Q_j that contain P_j. □

Definition 6.5. The *target node* $N_{\lambda(Q_j)}$ of a query Q_j or sub-query P_j is a network node to which the result of the query will be transported. □

Definition 6.6. The *request* of an atomic fragment at site θ is the sum of frequencies of predicates that occur in their positive form and are issued at site θ:

$$request_\theta(F_g) = \sum_{i=1, i \in J_\theta}^{w} f_i.$$

□

Definition 6.7. The *pay* of allocating an atomic horizontal fragment F_g at a site θ is the costs of accessing the atomic horizontal fragment F_g by all queries from sites other than θ:

$$pay_\theta(F_g) = \sum_{\theta'=1, \theta' \neq \theta}^{k} request_{\theta'}(F_g) \cdot c_{\theta\theta'}.$$

□

Definition 6.8. Let s_{ji} be the size of the data volume required by P_j from fragment F_g ($j = 1, \ldots, M$, $g = 1, \ldots, m$). Then the *need* of data from F_g by queries P_j at node N_θ can be expressed by:

$$need_\theta(F_g) = \sum_{\lambda(P_j)=\theta} s_{jg} \cdot f_j. \tag{3}$$

□

Furthermore, the portion of the total transportation costs for all queries in P^M that is due to fragment F_g being allocated to network node θ, i.e. $\lambda(F_g) = \theta$, amounts to

$$cost_{\lambda(F_g)=\theta}(P^M) = \sum_{j=1}^{M} s_{jg} \cdot f_j \cdot c_{\lambda(P_j)\theta}. \tag{$**$}$$

Here the $c_{\lambda(P_j)\theta}$ denotes the transportation cost factors between nodes $N_{\lambda(P_j)}$ and N_θ. Further, looking at Formula ($**$) we analyze relationships between *cost*, *pay* and *need*.

$$\begin{aligned}
cost_{\lambda(F_g)=\theta}(P^M) &= \sum_{\theta'=1, \theta' \neq \theta}^{k} \sum_{\lambda(P_j)=\theta'} s_{jg} \cdot f_j \cdot c_{\lambda(P_j)\theta} \\
&= \sum_{\theta'=1, \theta' \neq \theta}^{k} need_{\theta'}(F_g) \cdot c_{\theta'\theta} \\
&= \sum_{\lambda(P_j)=\theta} pay_\theta(F_g) \cdot s_{jg}
\end{aligned}$$

The above formulae give rise to two alternative heuristics for the allocation of given fragments F_g.

– The first heuristic allocates F_g to N_θ such that $need_\theta(F_g)$ is maximal, i.e. we choose the network node with the highest *need* of data from the fragment F_g. This heuristic is based on the assumption of the simple query environment, as assumed in [24], with unit transportation costs being the same between any pair of network nodes. This guarantees that there are no transport costs associated with data from fragment F_g for those queries that need most of the fragment. In addition, the availability of data from fragment F_g will be maximized.
– The second heuristic allocates F_g to N_θ such that $cost_{\lambda(F_g)=\theta}(P^M)$ will be minimal. For an atomic fragment it is sufficient to allocate F_g to N_θ such that $pay_\theta(F_g)$ is minimal, i.e. we choose the network node in such a way that the *pay* for all queries arising from this allocation are minimized. While this truly leads to lower costs for all queries, it may be less advantageous in terms of data availability.

It is easy to formulate the algorithm for the first heuristic. The following algorithm describes fragment allocation based on the second of these heuristics.

With the terms defined above, I now present an algorithm for horizontal fragmentation, shown as Algorithm 6.9. The algorithm first finds the site that has the biggest value of *pay* of each atomic fragment and then allocates the atomic fragment to the site. A fragmentation schema and fragment allocation schema can be obtained simultaneously.

Algorithm 6.9. [Cost-Based Primary Horizontal Fragmentation Algorithm]
Input: E /* a database type
$\Phi^y = \{\varphi_1, \ldots, \varphi_y\}$ /* a set of simple predicates defined on E
a set of network nodes $N = \{1, \ldots, k\}$ with cost factors
Output: Horizontal fragmentation schema and allocation schema$\{E_{H1}, \ldots, E_{Hk}\}$
Method: `for each` $\theta \in \{1, \ldots, k\}$
$E_{H\theta} = \emptyset$
`endfor`
`define` a set of normal predicates $\mathcal{N}^y$ using Φ^y
`define` a set of atomic horizontal fragments $\mathcal{F}^y$ using $\mathcal{N}^y$
`for each` atomic horizontal fragment $F_g \in \mathcal{F}^y$, $1 \leq i \leq 2^y$ `do`
`for each` node $\theta \in \{1, \ldots, k\}$ do
`calculate` $request_\theta(F_g)$
`calculate` $pay_\theta(F_g)$
`endfor`
`choose` w such that $pay_w(F_g) = \min(pay_1(F_g), \ldots, pay_k(F_g))$
/* find the minimum value
$\lambda(F_g) = \mathcal{N}_w$ /* allocate E_j to the site of the smallest *pay*
`define` $E_{H\theta}$ with $E_{H\theta} = \bigcup\{F_g : \lambda(F_g) = N_\theta\}$
`endfor`

□

Obviously, the complexity of Algorithm 6.9 is in $\mathbf{O}(k \cdot 2^x)$, where k is the number of network nodes and x is the number of simple selection predicates considered, which gives 2^x as an upper bound for the number of fragments. Note that the naive approach of comparing costs for all possible allocations of these fragments would give a complexity in $\mathbf{O}(k^{2^x})$ which definitely is intractable.

EXAMPLE 6.1. We now illustrate Algorithm 6.9 using an example. Given a set of simple predicates:

- $\varphi_1 \equiv$ in.dname = 'information', issued at site 1 with frequency 20 and site 2 with frequency 30,
- $\varphi_2 \equiv$ name.titles $\ni$ 'Dr', issued at site 2 with frequency 40,
- $\varphi_3 \equiv$ phone.areacode = 06, issued at site 3 with frequency 100.

We now fragment instance db(LECTURER) in Example 3.3 with the following steps:

1. Define a set of normal predicates as below:
   $N_1 = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$
   $N_2 = \varphi_1 \wedge \varphi_2 \wedge \neg\varphi_3$
   $N_3 = \varphi_1 \wedge \neg\varphi_2 \wedge \varphi_3$
   $N_4 = \varphi_1 \wedge \neg\varphi_2 \wedge \neg\varphi_3$
   $N_5 = \neg\varphi_1 \wedge \varphi_2 \wedge \varphi_3$
   $N_6 = \neg\varphi_1 \wedge \varphi_2 \wedge \neg\varphi_3$
   $N_7 = \neg\varphi_1 \wedge \neg\varphi_2 \wedge \varphi_3$
   $N_8 = \neg\varphi_1 \wedge \neg\varphi_2 \wedge \neg\varphi_3$
2. Define a set of atomic fragments using the set of normal predicates from previous step:
   $$db(\text{LECT}_g) = \sigma_{N_g}(db(\text{LECTURER})), 1 \leq g \leq 8.$$
3. Calculate $request_\theta(\text{LECT}_g)$ for each site to get an Atomic fragment Request Matrix as shown in Table 6.1.

Table6.1. Atomic Fragment Request Matrix

request	LECT$_1$	LECT$_2$	LECT$_3$	LECT$_4$	LECT$_5$	LECT$_6$	LECT$_7$	LECT$_8$
$request_1(\text{LECT}_g)$	20	20	20	20	0	0	0	0
$request_2(\text{LECT}_g)$	70	70	0	0	40	40	0	0
$request_3(\text{LECT}_g)$	50	0	50	0	50	0	50	0

4. Calculate $pay_\theta(\text{LECT}_g)$ for each site to get an Atomic Fragment Pay Matrix as shown in Table 6.2.

Table6.2. Atomic Fragment Pay Matrix

pay	LECT$_1$	LECT$_2$	LECT$_3$	LECT$_4$	LECT$_5$	LECT$_6$	LECT$_7$	LECT$_8$
$pay_1(\text{LECT}_g)$	1950	700	1250	0	1650	400	1250	0
$pay_2(\text{LECT}_g)$	2200	200	2200	200	2000	0	2000	0
$pay_3(\text{LECT}_g)$	3300	3300	500	500	1600	1600	0	0

5. Allocate atomic fragments to the site of the smallest *pay* to get atomic fragment allocation as shown in Table 6.3.

Table6.3. Atomic Fragment Allocation

Fragment	LECT_1	LECT_2	LECT_3	LECT_4	LECT_5	LECT_6	LECT_7	LECT_8
site N_θ	2	2	3	1	3	2	3	2

6. The last step unions atomic fragments by sites that the atomic fragments are allocated to obtain fragmentation of LECTURER. Table 6.4 below shows three fragments of *db*(LECTURER), allocated to site 1, 2 and 3, respectively.

Table6.4. Horizontal Fragmentation of *db*(LECTURER)

db(LECTURER_1)

in	id	name			email	phone		homepage
dname		fname	lname	titles		areacode	number	
				title				
Information	2010	Shirley	Churchill	Lecturer	s.churchill	04	4983677	churchill.com
Accounting	2618	James	Hooks	Lecturer	j.hooks	04	4663365	hooks.com

db(LECTURER_2)

in:	id	name			email	phone		homepage
dname		fname	lname	titles		areacode	number	
				title				
Accounting	1001	John	Dever	Professor Dr	j.dever	09	8556677	dever.com
Information	3203	Jerry	Hubbard	HoD Professor Dr	j.hubbard	06	3569988	hubbard.com

db(LECTURER_3)

in	id	name			email	phone		homepage
dname		fname	lname	titles		areacode	number	
				title				
Marketing	1002	Allan	Barry	Senior Lecturer Dr	a.barry	06	3556688	barry.com

We now compare total query costs before and after fragmentation. Assuming selection is performed locally, total query costs before fragmentation is 9,316,000 for the optimized allocation of instance *db*(LECTURER), which is site 1. After horizontal fragmentation using Algorithm 6.9, total query costs for queries retrieving data using the three predicates is 7,535,000. Obviously horizontal fragmentation using our approach can indeed improve the system performance by reducing data transportation cost. It is intuitive that the correctness rules presented in Chapter 4 are satisfied by the resulting fragmentation schema. □

If we use the cost model to do allocation for the set of input atomic fragments, we need

to try all the combinations of atomic fragments and allocate them to all possible allocations to find the optimized allocation. In the experiment evaluation in Section 6.1.4 we will prove that Algorithm 6.9 can indeed lead to optimized allocation for atomic fragments.

6.1.1 Fragmentation and Allocation Refinement for Horizontal Fragments

We may, on the one hand, assume that not all simple selection formulae in Φ^w are used for building normal predicates and hence also atomic fragments, but that only simple predicates φ_y with $y \in \mathcal{M} \subseteq \{1, \dots, m\}$ are taken into account. This makes sense, as the number of atomic fragments can be up to 2^x with $x = |J|$ being the number of simple predicates used for horizontal fragmentation. On the other hand, the number k of network nodes is a reasonable upper bound for the number of horizontal fragments. Therefore, the recombination of atomic horizontal fragments will become necessary.

Nevertheless, it would be desirable to further reduce the complexity of an allocation algorithm, which leads us to the second problem mentioned above. For this let us apply the *cost minimization heuristic*, which allocates a fragment F to a network node N_θ such that $cost_{\lambda(F)=\theta}(P^M)$ will be minimal. Now assume that we are given an allocation λ and that F is some fragment that can be further fragmented into F_1 and F_2.

According to our discussion of how horizontal fragmentation affects the query costs, summarized in Fact 5.2, we have some transport costs associated with transporting a portion of F to the target node of the corresponding query P_j. Assuming a fully connected network, if the allocation of F to $N_{\lambda(F)}$ is optimal, then not both fragments F_1 and F_2 need to be reallocated. Otherwise there would also be a better allocation for F. This leads to the following fact.

Fact 6.1 *Assume transportation costs dominate the total query costs. Fragmenting F horizontally into F_1 and F_2 gives rise to one of the following two cases:*

1. *One of the two fragments F_1 and F_2 will reside at the same location as F before fragmentation, whereas the other fragment will be moved to a new location.*
2. *Both fragments F_1 and F_2 will reside at the same node, which then must be also the same location as F before fragmentation.*

Proof. Consider F of size s being accessed by two queries, Q_1 and Q_2, which are executed at sites N_1 and N_2, with frequency f_1 and f_2, respectively. Assuming $f_1 \geq f_2$, the optimal allocation of F is site N_1 because transportation costs dominate the total query costs. Given c_{12} and c_{21} as transportation cost factors between site N_1 and N_2. Generally, c_{12} should be equal to c_{21}. Thus we have

$$cost_{\lambda(F)=1}(Q^2) \approx s \cdot f_2 \cdot c_{12}.$$

Let s_1 and s_2 be the size of F_1 and F_2, respectively. Assume that, after fragmentation, the optimal allocation of both fragments are at the site N_2, then total query costs are

$$cost_{\lambda(F)=2,\lambda(F)=2}(Q^2) = (s_1 + s_2) \cdot f_1 \cdot c_{21} = s \cdot f_1 \cdot c_{21}.$$

The objective of fragmentation and allocation is to improve system performance by reducing total query costs, therefore we should have

$$cost_{\lambda(F)=2,\lambda(F)=2}(Q^2) < Cost_{\lambda(F)=1}(Q^2).$$

That means, $s \cdot f_1 \cdot c_{21} < s \cdot f_2 \cdot c_{12}$ or $f_1 < f_2$. This conflicts with the assumption $f_1 > f_2$. Therefore, it is impossible that both fragments F_1 and F_2 will be moved to a new site after fragmentation. If both fragment are still reside at the same site as before, that means the fragmentation is not necessary because the total query costs does not reduced. Therefore, at least one fragment will be moved to a new location. This proves Fact 6.1. □

While in the second case in Fact 6.1 the transportation costs remain the same, in the first case the transportation costs will be reduced. This suggests the need to take a total order on the elements of $\Phi^w = \{\varphi_1, \dots, \varphi_w\}$ with $\varphi_i < \varphi_j$ if and only if $i < j$ such that the following property holds:

> If F is a fragment resulting from a normal predicate in $\mathcal{N}^y$ for $\Phi^y = \{\varphi_1, \dots, \varphi_y\}$ and further fragmentation of F with φ_{y+1} will not reduce the query costs according to case 2 in Fact 6.1, then none of the fragmentation predicates in $\{\varphi_{y+1}, \dots, \varphi_n\}$ is likely to reduce the query costs.

Given such an order on Φ allows us to apply binary search to determine an "optimal" fragmentation and allocation in the following way:

1. Choose y and consider the fragmentation according to normal predicates in $\mathcal{N}^y$.
2. Using Fact 6.1 φ_{y+1} divides the fragments into those that are subject to a search for a finer fragmentation, i.e. those for which case 1 in Fact 6.1 applies, and those for which the given fragmentation will already be too fine.
3. Of course, for some fragments we may already deduce that we have already found the best fragmentation.
4. For the first case choose $y' > y$ and repeat the process, for the second case take $y' < y$ and redo the process.

It is reasonable to choose the ordering of the selection predicates according to the data volume generated by the fragment. Each query P_j requires a particular portion of size s_{ji} from fragment F_i. Therefore, define the associated data *volume* as

$$vol(F_i) = \sum_{j=1}^{m} f_j \cdot s_{ji},$$

where the sum runs over all queries P_j and f_j is the frequency of P_j. So $vol(F_i)$ indicates the total data volume due to fragment F_i. Therefore, order the selection predicates according to decreasing data volume.

Finally, note again that this refinement of fragmentation and allocation is not necessarily restricted to horizontal fragments as Facts 5.1 and 5.3 allow us to generalize the result to splitting and vertical fragmentation.

6.1.2 An Example

Assume that a network has three nodes, a, b and c, with transportation cost factors c_{ij} between each pair of nodes as:

Site	a	b	c
a	0	10	25
b	10	0	20
c	25	20	0

Assume there are queries Q_j ($j \in \{0, \dots, 4\}$) that retrieve information from the instance of type E such that the frequencies f_j of the queries and the sizes s_j of the data retrieved are given in the following table:

Queries Q_j	Q_0	Q_1	Q_2	Q_3	Q_4
Node $N_{\lambda(Q_j)}$	a	b	c	b	b
Frequency f_j	20	15	10	5	1
Size s_j	100	200	50	50	100

If the instance of database type E is stored at site a then the total query costs of all the queries are:

$$cost_{\lambda(E)=a}(Q^5) = \sum_{j=0}^{4} s_j \cdot f_j \cdot c_{a\lambda(Q_j)} = 46,000$$

with $c_{a\lambda(Q_j)}$ as transportation factors between site a and the target node $N_{\lambda(Q_j)}$ of query Q_j.

If the fragment E is stored at site b, then query Q_0 and Q_2 need to access $db(E)$ remotely while other queries, Q_1, Q_3 and Q_4, are processed locally. This allocation is illustrated in Figure 6.1.

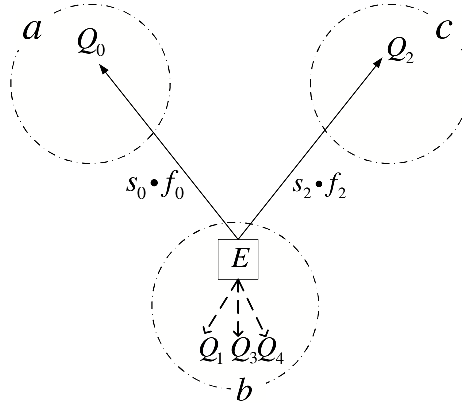

Fig. 6.1. Allocation without Fragmentation

The total query costs under the above allocation are:

$$cost_{\lambda(E)=b}(Q^5) = \sum_{j=0}^{4} s_j \cdot f_j \cdot c_{b\lambda(Q_j)} = 30,000$$

Similarly, if the fragment E is stored at site c we get the total query costs as:

$$cost_{\lambda(E)=c}(Q^5) = \sum_{j=0}^{4} s_j \cdot f_j \cdot c_{c\lambda(Q_j)} = 117,000$$

We conclude that allocating E to node b leads to minimal total query costs. Hence, E should be allocated to site b.

Fragmentation Step 1 If fragment E is horizontally fragmented into E_1 and E_2 according to φ_0 in Q_0, which is the most frequently executed query, then $db(E) = db(E_1) \cup db(E_2)$.

Assume the sizes of fragments are: $s_{E_1} = 100$ and $s_{E_2} = 150$. Let us use s_{j1}, s_{j2} to indicate the amount of data retrieved by query Q_j from E_1, E_2, respectively. The following table shows the target site, frequencies, and the volume of data retrieved by each of the queries.

Queries Q_j	Q_0	Q_1	Q_2	Q_3	Q_4
Node $\lambda(Q_j)$	a	b	c	b	b
Frequency f_j	20	15	10	5	1
Size s_{j1}	100	50	0	25	50
Size s_{j2}	0	150	50	25	50

Query Q_0 only retrieves information from fragment E_1. Therefore we allocate fragment E_1 at site a and leave the fragment E_2 at site b. This is illustrated in Figure 6.2.

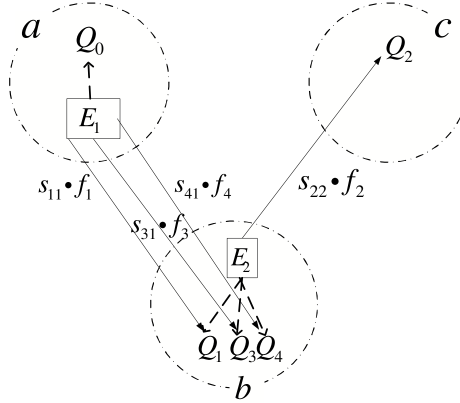

Fig. 6.2. Reallocation of Fragments: Step 1

For situations where fragment E_1 is allocated to each of the three available network sites a, b and c, the total costs of all queries accessing fragment E_1 can be calculated as following:

$$cost_{\lambda(E_1)=a}(Q^5) = \sum_{i=0}^{4} s_{j1} \cdot f_j \cdot c_{a\lambda(Q_j)} = 9,250$$

$$cost_{\lambda(E_1)=b}(Q^5) = \sum_{i=0}^{4} s_{j1} \cdot f_j \cdot c_{b\lambda(Q_j)} = 20,000$$

$$cost_{\lambda(E_1)=c}(Q^5) = \sum_{j=0}^{4} s_{j1} \cdot f_j \cdot c_{c\lambda(Q_j)} = 68,500$$

We can also calculate the total costs of all queries accessing fragment E_2 regarding different allocations:

$$cost_{\lambda(E_2)=a}(Q^5) = \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{a\lambda(Q_j)} = 36,750$$

$$cost_{\lambda(E_2)=b}(Q^5) = \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{b\lambda(Q_j)} = 10,000$$

$$cost_{\lambda(E_2)=c}(Q^5) = \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{c\lambda(Q_j)} = 48,500$$

The results of the above calculation show that allocation that leads to lowest query costs is to store fragment E_1 at a and E_2 at site b, as shown in Figure 6.2.

The calculation above is based on the assumption that all queries access data from fragment E_1 and E_2 separately. Now let us apply Scenario II, that is, if an instance of a database type has been split into two fragments with some queries accessing both fragments, the smaller fragment should be transferred to the site allocated to the bigger fragments and then the data from the union of the fragments be transferred to the site issued the queries.

If both fragments E_1 and E_2 are allocated at site a then it should lead to the same costs as when the instance E is not fragmented and allocated at site a.

$$cost_{\lambda(E_1)=a,\lambda(E_2)=a}(Q^5) = \sum_{j=0}^{4} s_{j1} \cdot f_j \cdot c_{a\lambda(Q_j)} + \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{a\lambda(Q_j)} = 46,000$$

It will be the same as before applying the fragmentation, when both fragments E_1 and E_2 are allocated to site b or site c:

$$cost_{\lambda(E_1)=b,\lambda(E_2)=b}(Q^5) = \sum_{j=0}^{4} s_{j1} \cdot f_j \cdot c_{a\lambda(Q_j)} + \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{a\lambda(Q_j)} = 30,000$$

$$cost_{\lambda(E_1)=c,\lambda(E_2)=c}(Q^5) = \sum_{j=0}^{4} s_{j1} \cdot f_j \cdot c_{c\lambda(Q_j)} + \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{c\lambda(Q_j)} = 117,000$$

So how are query costs affected if the two fragments are allocated to two different sites? First we allocate fragment E_1 to site a and E_2 to site b or c. Figure 6.3 illustrates remote data transportation under the allocation $\lambda(E_1) = a$ and $\lambda(E_2) = c$.

The total query costs of accessing E_1 and E_2 under the above allocations should be

$$cost_{\lambda(E_1)=a,\lambda(E_2)=b}(Q^5) = \\ s_{01} \cdot f_0 \cdot c_{aa} + s_{11} \cdot f_1 \cdot c_{ab} + s_{31} \cdot f_3 \cdot c_{ab} + s_{41} \cdot f_4 \cdot c_{ab} + s_{22} \cdot f_2 \cdot c_{bc} = 19,250$$

$$cost_{\lambda(E_1)=a,\lambda(E_2)=c}(Q^5) = s_{01} \cdot f_0 \cdot c_{aa} + s_{11} \cdot f_1 \cdot c_{ac} + (s_{11} + s_{12}) \cdot f_1 \cdot c_{cb} + s_{31} \cdot f_3 \cdot c_{ab} \\ + s_{32} \cdot f_3 \cdot c_{cb} + s_{41} \cdot f_4 \cdot c_{ab} + s_{41} \cdot f_4 \cdot c_{cb} + s_{22} \cdot f_2 \cdot c_{cc} = 84,000$$

Then we allocate fragment E_1 at site b and fragment E_2 at site a or c and calculate the

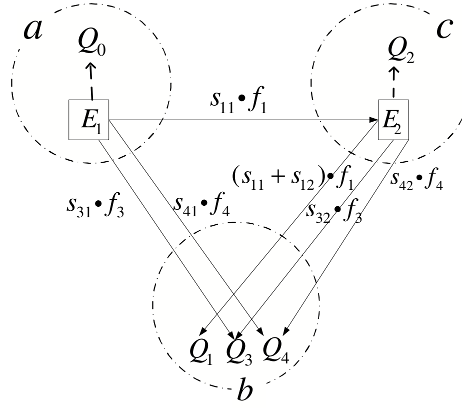

Fig. 6.3. Reallocation of Fragments: Step 2

query costs as below

$$cost_{\lambda(E_1)=b,\lambda(E_2)=a}(Q^5) = \\ s_{01} \cdot f_0 \cdot c_{ba} + s_{12} \cdot f_1 \cdot c_{ab} + s_{32} \cdot f_3 \cdot c_{ab} + s_{42} \cdot f_4 \cdot c_{ab} + s_{22} \cdot f_2 \cdot c_{ac} = 56,750$$

$$cost_{\lambda(E_1)=b,\lambda(E_2)=c}(Q^5) = \\ s_{01} \cdot f_0 \cdot c_{ba} + s_{12} \cdot f_1 \cdot c_{cb} + s_{32} \cdot f_3 \cdot c_{cb} + s_{42} \cdot f_4 \cdot c_{cb} + s_{22} \cdot f_2 \cdot c_{cc} = 68,500$$

Now we allocate E_1 at site c and E_2 at a and b, and the costs of queries to access all the required data from E_1 and E_2 are

$$cost_{\lambda(E_1)=c,\lambda(E_2)=a}(Q^5) = s_{01} \cdot f_0 \cdot c_{ca} + s_{11} \cdot f_1 \cdot c_{ca} + (s_{11} + s_{12}) \cdot f_1 \cdot c_{ab} + s_{31} \cdot f_3 \cdot c_{ab} \\ + s_{32} \cdot f_3 + \cdot c_{ab} + s_{41} \cdot f_4 \cdot c_{cb} + s_{42} \cdot f_4 \cdot c_{ab} + s_{22} \cdot f_2 \cdot c_{ac} = 116,000$$

$$cost_{\lambda(E_1)=c,\lambda(E_2)=b}(Q^5) = \\ s_{01} \cdot f_0 \cdot c_{ca} + s_{11} \cdot f_1 \cdot c_{cb} + s_{31} \cdot f_3 \cdot c_{cb} + s_{41} \cdot f_4 \cdot c_{cb} + s_{22} \cdot f_2 \cdot c_{bc} = 68,500$$

From the results of the above calculations we find that allocation of E_1 at site a, and E_2 at site b, leads to the lowest total query costs.

Using the formula 3 we can calculate needs of each fragments at the three different sites:

$$need_a(E_1) = \sum_{\lambda(Q_j)=a} s_{j1} \cdot f_j = 2000 \qquad need_a(E_2) = \sum_{\lambda(Q_j)=a} s_{j2} \cdot f_j = 0$$

$$need_b(E_1) = \sum_{\lambda(Q_j)=b} s_{j1} \cdot f_j = 925 \qquad need_b(E_2) = \sum_{\lambda(Q_j)=b} s_{j2} \cdot f_j = 2425$$

$$need_c(E_1) = \sum_{\lambda(Q_j)=c} s_{j1} \cdot f_j = 0 \qquad need_c(E_2) = \sum_{\lambda(Q_j)=c} s_{j2} \cdot f_j = 500$$

The results of the above calculations show that the *need* of fragment E_1 is biggest at site b while the *need* of E_2 is biggest at site a. Therefore allocating fragments to the sites that need them the most will lead to the lowest total query costs.

Fragmentation Step 2 Assume query Q_1 retrieves the highest data volume $s_1 \cdot f_1$ from $db(E)$. Then we fragment $db(E)$ horizontally according to the selection condition φ_1 in Q_1. The sizes of the resulting fragments are $s_{E_1} = 200$, $s_{E_2} = 200$. Let s_{ji} indicate the size of data volume retrieved by query Q_j from fragment E_i. The sizes of data volumes retrieved by each of the query Q_j are shown in the following table.

Queries Q_j	Q_0	Q_1	Q_2	Q_3	Q_4
Node $\lambda(Q_j)$	a	b	c	b	b
Frequency f_j	20	15	10	5	1
Size s_{j1}	50	200	25	0	100
Size s_{j2}	50	0	25	50	0

Then total costs of queries are calculated for the three different allocations of fragment E_1:

$$cost_{\lambda(E_1)=a}(Q^5) = \sum_{j=0}^{4} s_{j1} \cdot f_j \cdot c_{a\lambda(Q_j)} = 39,750$$

$$cost_{\lambda(E_1)=b}(Q^5) = \sum_{j=0}^{4} s_{j1} \cdot f_j \cdot c_{b\lambda(Q_j)} = 15,000$$

$$cost_{\lambda(E_1)=c}(Q^5) = \sum_{j=0}^{4} s_{i1} \cdot f_j \cdot c_{c\lambda(Q_j)} = 89,000$$

Total costs of queries accessing fragment E_2 are:

$$cost_{\lambda(E_2)=a}(Q^5) = \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{a\lambda(Q_j)} = 8,750$$

$$cost_{\lambda(E_2)=b}(Q^5) = \sum_{j=0}^{4} s_{i2} \cdot f_j \cdot c_{b\lambda(Q_j)} = 15,000$$

$$cost_{\lambda(E_2)=c}(Q^5) = \sum_{j=0}^{4} s_{j2} \cdot f_j \cdot c_{c\lambda(Q_j)} = 30,000$$

The results of the cost calculations show that allocating fragment E_1 to site b and fragment E_2 at site a leads to the least total query costs. This allocation is illustrated in Figure 6.4.

In this example there are only queries that either retrieve data from one fragment or retrieve the same volume of data from the two fragments. We only apply the calculation formula described as scenario III. There is no need to transfer the smaller fragments to join with the larger one and transfer the results to the target site.

Again, by using Formula 3, presented in Section 6.1, we can calculate needs of each fragments at the three different sites:

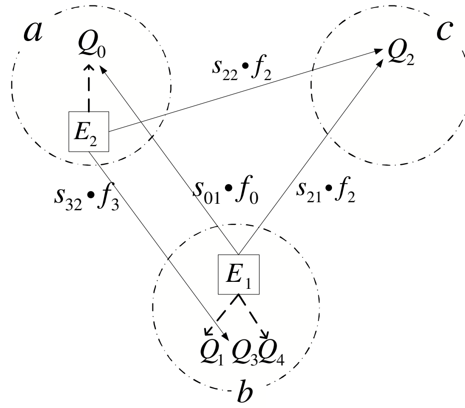

Fig. 6.4. Reallocation of Fragments: Step 3

$$need_a(E_1) = \sum_{\lambda(Q_j)=a} s_{j1} \cdot f_j = 1000 \qquad need_a(E_2) = \sum_{\lambda(Q_j)=a} s_{j2} \cdot f_j = 1000$$

$$need_b(E_1) = \sum_{\lambda(Q_j)=b} s_{j1} \cdot f_j = 3100 \qquad need_b(E_2) = \sum_{\lambda(Q_j)=b} s_{j2} \cdot f_j = 250$$

$$need_c(E_1) = \sum_{\lambda(Q_j)=c} s_{j1} \cdot f_j = 250 \qquad need_c(E_2) = \sum_{\lambda(Q_j)=c} s_{j2} \cdot f_j = 250$$

The results from the above calculations show that the *need* of fragment E_1 is biggest at site a while the *need* of E_2 is biggest at site b, in which case total query costs are minimal.

It can be concluded that we can apply an allocation heuristic, allocating fragments according to the needs of the fragments at each site. Each of the fragments should be allocated to the sites that need them the most so the total query costs are minimal.

This example also shows that if an instance of a database type is optimally allocated to a site, after fragmentation the optimal allocation only leads to one fragment being reallocated to another site while the other fragment remains at the same site before fragmentation.

6.1.3 Simple Selection Predicates for Horizontal Fragmentation

Both allocation procedures presented above involve an unspecified number of simple selection predicates. We now investigate how to determine in advance a reasonably low number of such predicates following a procedure proposed in [75]. The procedure leads to a reasonable fragmentation schema by looking at most frequently used simple predicates. Based on the cost model introduced in Section 3.3, this heuristic procedure for horizontal fragmentation consists of the following steps:

1. Sort queries in Q^m by decreasing frequency to get a list of queries $[Q_1, \ldots, Q_m]$.
2. Optimize all the queries and extract simple predicates from the queries to get a list of Φ of simple selection predicates.
3. Construct a predicate usage matrix based on Φ to obtain a list of simple predicates $\Phi^w = [\varphi_1, \ldots, \varphi_w]$ for each database type E.
4. Determine y with $1 \leq y \leq w$ such that fragmentation with the first y predicates in Φ^w leads to a reallocation of a fragment, whereas the fragmentation with elements in φ_{y+1}

does not add changes. This number y can be determined by binary search using Algorithm 6.11 below.

5. Take the first y simple predicates in Φ^w to get a subset of simple predicates Φ^y.
6. Perform horizontal fragmentation with Φ^y using Algorithm 6.10 below. This results in a fragmentation schema for database type E to which the allocation approach in the previous two subsections can be applied.

We first introduce Algorithm 6.10 for calculating total query costs according to the cost model from Section 3.3. In the algorithm we use $Q^m = \{Q_1, \dots, Q_m\}$, the set of the most frequent queries, and $\Phi^w = [\varphi_1, \dots, \varphi_w]$, the list of simple selection predicates contained in Q^m, to determine the total query costs $cost_x$ in case E is fragmented by using the first x simple selection predicates in Φ^w.

Algorithm 6.10. [Computation of Query Costs]
Input: E /*a database type
$Q^m = \{Q_1, \dots, Q_m\}$ /* a set of global queries
$\Phi^w = [\varphi_1, \dots, \varphi_w]$ /* a set of simple predicates defined on E
$x \in \{1, \dots, m\}$
Output: $cost_x(Q^m)$
Method:

```
take the first x simple predicates in Φ^w to get Φ^x
apply Algorithm 6.9 to get a horizontal fragmentation schema F_E = {E_H1, ..., E_Hk}
for each query tree Q_j ∈ Q^m do
    replace db(E) by db(E_H1) ∪ ··· ∪ db(E_Hk) with s_ji > 0
    allocate network nodes to intermediate nodes of the query tree
    calculate the query costs cost_λ(Q_j) using the cost model
endfor
calculate total query costs cost_x(Q^m) = Σ_{j=1}^{m} cost_λ(Q_j) · f_j
```

$$cost_x(Q^m) = \sum_{j=1}^{m} cost_\lambda(Q_j) \cdot f_j$$

□

Using Algorithm 6.10 above, the following algorithm, Algorithm 6.11, determines the number y of simple predicates that should be used for horizontal fragmentation.

Algorithm 6.11. [Determination of Simple Selection Predicates]
Input: $Q^m = \{Q_1, \dots, Q_m\}$ /* a set of global queries
, $\Phi^w = [\varphi_1, \dots, \varphi_w]$ /* a set of simple predicates defined on E
$X = [0, \dots, w]$ /* a list of indices
Output: y
Method:

```
set a = 0, b = w
while b − a ≥ 3 do
    choose x_1, x_2 from X such that 0 < x_1 < x_2 < b
    for each x ∈ {a, x_1, x_2, b} do
        apply Algorithm 6.10 to determine cost_x
    endfor
    min := Min{cost_a, cost_x1, cost_x2, cost_b}            /* find the minimum costs
    if cost_y1 = cost_y2 and y_1 < y_2 with y_1, y_2 ∈ {a, b, x_1, x_2}
```

```
        then min := cost_{y1}
        endif
        if cost_a = min then b := x_1
        elsif cost_{x1} = min then b := x_2
        elseif cost_{x2} = min then a := x_1
        elsif cost_b = min then a := x_2
        endif
    enddo
    choose y ∈ {a,... b} such that cost_y = Min{cost_x : x ∈ {a,...,b}}
```

□

Basically, the above algorithm takes as input a list of indices of simple predicates, iteratively chooses four numbers a, b, x_1, x_2 with $a < x_1 < x_2 < b$; starting with $a = 0$ and $b = w$, calculates the corresponding total query costs, and compares these costs to decide a reasonable number y of simple predicates for horizontal fragmentation.

6.1.4 Experimental Evaluation

This section presents the results of some experiments that were conducted to validate the heuristics underlying Algorithms 6.9 and 6.11 proposed in the previous section. For these experiments a database schema $\mathcal{S}$ was designed and populated to get a database $db(\mathcal{S})$. Then four network sites were assumed and 30 representative queries were defined following a similar pattern of queries as in the OO7 project [22]. For one database type $E \in \mathcal{S}$ a set Φ of 14 simple selection predicates were obtained from the 30 queries.

Before testing Algorithm 6.11 Algorithm 6.9 was first validated because Algorithm 6.9 is used by Algorithm 6.11. The instance $db(E)$ of type E was fragmented, using all 14 simple predicates, into 42 atomic fragments which contains tuples. It is infeasible to get an optimal allocation schema for these atomic fragments because the complexity of an exhaustive comparison of costs involving $4^{42} \approx 10^{26}$ possible allocations is too high. Therefore, to allocate the resulting set of atomic fragments the following two different allocation strategies were tried and the total query costs arising from them were compared:

- Each atomic fragment was allocated according to Algorithm 6.9 and the corresponding total query costs were computed.
- Several allocation schemata were chosen randomly and total query costs for each of them were calculated. Then the lowest query cost among them was considered.

The experimental results showed that the total query costs for both strategies were the same. This means that the heuristic Algorithm 6.9 can indeed lead to a semi-optimal allocation schema. It can be concluded that given a set of simple selection predicates for horizontal fragmentation, allocating each atomic fragment to a site that requests it most frequently leads to a fragment allocation schema with nearly minimal total query costs. This justifies the chosen heuristics in Algorithm 6.9.

To valid the Algorithm 6.11 total query costs were first calculated with respect to every value $x \in \{1, 2, \ldots, 14\}$, the number of simple predicates from Φ^w that is used for fragmenting a database type E, on three different database instances that have the same selectively with respect to each simple predicate in Φ^w but with different sizes. The results are shown in Figure

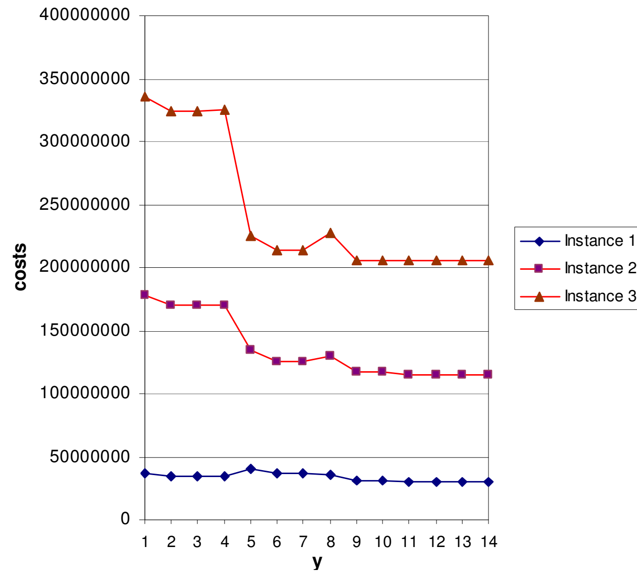

Fig. 6.5. Simple Predicate and Query Costs

6.5. Note that the trend in all three cases is similar, i.e. there is no more improvement for $y \geq 11$.

Algorithm 6.11 were then tested on three different instances and obtained the following results:

test	1	2	3
instance size	1,000	5,000	10,000
y	11	11	11

From these results the following can be concluded:

– Given a set of simple predicates Φ^w, we only need a subset Φ^y of simple predicates to perform horizontal fragmentation such that the system performance can be improved in the same way as using all the simple predicates in Φ^w.
– The heuristic is efficient in the sense of rapidly getting the value of y.
– With the increase in sizes of the instances, the query costs obviously vary with the change of the number of simple predicates for fragmentation, i.e. the more simple predicates the less total query costs. But total query costs stop changing from a certain point (see Figure 6.5).

This result suggests that to fragment an instance $db(E)$ we can first get a value of y by applying Algorithm 6.11 on a small sample instance, which has the same selectivity as the original instance regarding the set of simple predicates. Then we use the first y simple predicates to perform fragmentation and fragment allocation on the original database instance. Note that with the reduced number of simple predicates for fragmentation the complexity of the follow-up allocation problem can be reduced from $\mathbf{O}(k^{2^m})$ to $\mathbf{O}(k^{2^y})$ with m being the number of simple predicates abstracted from the most frequent 20% queries. Using Algorithm 6.9 further reduces complexity to $\mathbf{O}(k \cdot 2^y)$.

6.2 Derived Horizontal Fragmentation

As with other distribution design techniques, the aim of derived horizontal fragmentation is to improve the performance of applications accessing the database. Therefore, a cost model should be employed to evaluate the total query costs of the global queries while making decisions on derived horizontal fragmentation. Ceri *et al.* [25] stated that the important parameter needed for horizontal fragmentation is the number of accesses performed by the applications to different portions of data. However, the parameter is not used while performing derived horizontal fragmentation. Further, in the literature, derived horizontal fragmentation is performed in an ad hoc way without considering how it will affect the system performance [15, 26, 45, 93]. Only at the later stage of allocation, system performance is evaluated. It is argued here that once the decision on derived horizontal fragmentation has been made, the possibilities of minimizing total query costs are restricted at the stage of allocation. In this section, the problems of designing derived horizontal fragmentation and allocating fragments in a way such that the overall performance of the distributed database system is better than the one of an equivalent centralized one are addressed. In the meantime, it will be shown that this approach can further improve system performance beyond traditional approach (as used in [93]). That is, with the cost model introduced in Chapter 3, a heuristic approach minimizing query costs for the case of derived horizontal fragmentation will be present. The minimization of transportation costs is decisive, and can be achieved by refining derived horizontal fragmentation using all the candidate fragmentation schemata.

In the following subsections an example is presented to show the problems of existing approaches of derived horizontal fragmentation. We will attempt to solve the problems by first analyzing the cost model and then consider a heuristic method based on the result of the analysis. We will see how the heuristic method is applied with a simple example. Then it will be proved that the proposed heuristic is correct with regard to the criteria of correctness of fragmentation in Section 4.4.

6.2.1 A Motivating Example

Assume there are three relations A, B, C in a database. Relation C is accessed by four queries $Q_1, \ldots, Q_4$ with different frequencies $f_1, \ldots, f_4$. Relation A is accessed by Q_1 and Q_2. Relation B is accessed by Q_3 and Q_4. Relation A and B have been horizontally fragmented using predicates. Relation A has been fragmented into A_1 and A_2 which are allocated to sites 1 and 2, respectively, i.e., $\lambda(A_1) = 1$, $\lambda(A_2) = 2$. Relation B has been fragmented into two fragments, B_3 and B_4, which are allocated to sites 3 and 4, respectively. Assume that there is no predicate defined on C, which is accessed by all four queries, and that $\mathrm{MAX}(f_1, f_2, f_3, f_4) = f_1$. Performing only primary horizontal fragmentation we will have Scenario I depicted in Figure 6.6, in which C is allocated to site 1, i.e., $\lambda_1(C) = 1$ according to the cost optimization rule. In the figures below (Figure 6.6, 6.7 and 6.8), all remote transactions and their frequencies are depicted with solid lines and values on the lines, while local transactions are depicted with broken, dashed lines with their frequencies marked on the lines.

Assuming that the transportation cost factors among all sites are the same, the total query costs of Scenario I is computed as:

$$cost_{\lambda_1}(Q^4) = s_C \cdot f_2 + s_C \cdot f_3 + s_C \cdot f_4$$

When a member relation has more than one owner relation, there will be more than one possible derived horizontal fragmentation schema. In this case the choice of a fragmentation

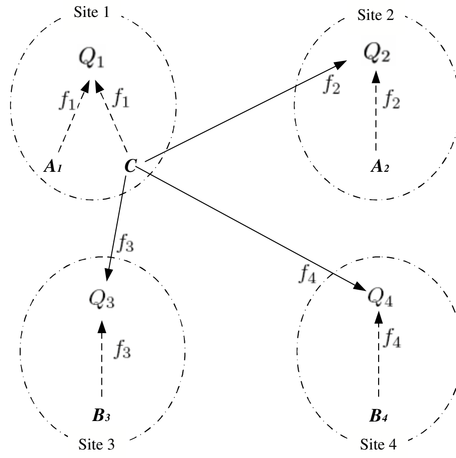

Fig. 6.6. Scenario I - Primary Fragmentation Only

that is used in more applications is recommended [93]. Assume $f_1 + f_2 > f_3 + f_4$, relation C is therefore derived horizontally fragmented by semijoin with fragments of A. The resulting fragmentation and fragment allocation is depicted in Scenario II in Figure 6.7. The total query costs for Scenario II are:

$$cost_{\lambda_2}(Q^4) = s_{C_1} \cdot f_3 + s_{C_2} \cdot f_3 + s_{C_1} \cdot f_4 + s_{C_2} \cdot f_4.$$

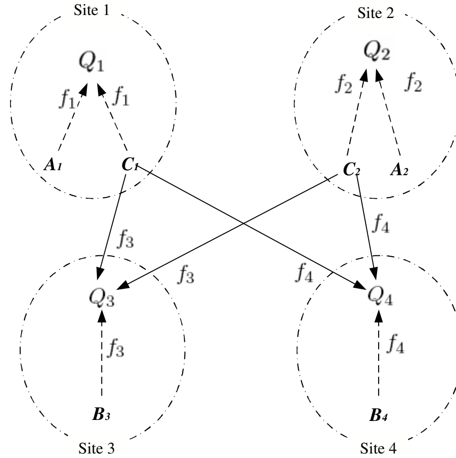

Fig. 6.7. Scenario II - Derived Fragmentation according to One Fragmentation Schema

However, performing derived fragmentation based on fragmentation schema of relation A can only improve the performance of the queries which access both the member relation C and the owner relation A. The performance of the queries that access the member relation C together with another owner relation B cannot be improved. That is, the chance of optimizing the system performance is restricted if the fragmentation of C is only based on fragmentation

of one owner relation. Now we look at what happens if we take into consideration fragmentation of both owner relations. In this case, we have two fragmentation schemata for C, i.e., $F_C = \{C_{1a}, C_{2a}\}$ and $F'_C = \{C_{3b}, C_{4b}\}$ with:

$$C_{1a} = C \ltimes A_1, C_{2a} = C \ltimes A_2,$$

$$C_{3b} = C \ltimes B_3, C_{4b} = C \ltimes B_4.$$

Applying intersection operation on $C_{ia} \in F_C, (1 \leq i \leq 2)$ and $C_{jb} \in F'_C, (3 \leq j \leq 4)$ we get the following finer fragmentation:

$$C_{1a3b} = C_{1a} \cap C_{3b}, C_{1a4b} = C_{1a} \cap C_{4b},$$

$$C_{1a3b} = C_{2a} \cap C_{3b}, C_{1a3b} = C_{2a} \cap C_{3b}.$$

Assuming $f_1 > f_3, f_1 > f_4, f_2 < f_3, f_2 > f_4$, with the cost model introduced in Section 3.3 we get the optimized allocation of the four atomic fragments as following:

$$\lambda(C_{1a3b}) = 1, \lambda(C_{1a4b}) = 1, \lambda(C_{2a3b}) = 3, \lambda(C_{2a4b}) = 2.$$

Scenario III in Figure 6.8 depicts the finer derived horizontal fragmentation and fragment allocation.

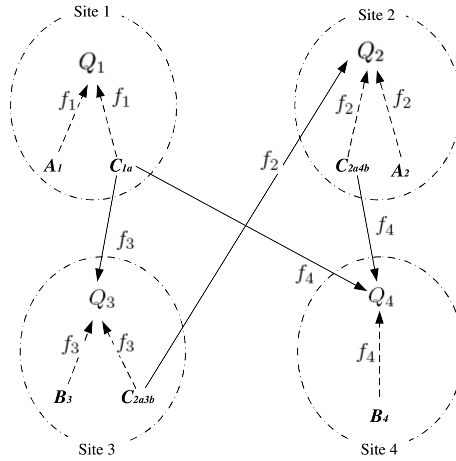

Fig. 6.8. Scenario III - Derived Fragmentation according to Two Fragmentation Schemata

In the case of Scenario III, the total query costs are:

$$cost_{\lambda_3}(Q^4) = s_{C_{2a3b}} \cdot f_2 + s_{C_{1a}} \cdot f_3 + s_{C_{1a}} \cdot f_4 + s_{C_{2a4b}} \cdot f_4$$

Comparing with the costs in Scenario I and Scenario II we get:

$$\begin{aligned} & cost_{\lambda_1}(Q^4) - cost_{\lambda_2}(Q^4) \\ = & s_C \cdot f_2 + s_C \cdot f_3 + s_C \cdot f_4 - (s_{C_1} \cdot f_3 + s_{C_2} \cdot f_3 + s_{C_1} \cdot f_4 + s_{C_2} \cdot f_4) \\ = & s_C \cdot f_2 + s_C \cdot f_3 + s_C \cdot f_4 - (s_{C_1} + s_{C_2}) \cdot f_3 - (s_{C_1} + s_{C_2}) \cdot f_4 \\ > & 0 \end{aligned}$$

We can conclude that the derived fragmentation using fragmentation of an owner relation can indeed further reduce the total query costs and therefore should be employed when doing database distribution design.

$$\begin{aligned} & cost_{\lambda_2}(Q^4) - cost_{\lambda_3}(Q^4) \\ = & s_{C_1} \cdot f_3 + s_{C_2} \cdot f_3 + s_{C_1} \cdot f_4 + s_{C_2} \cdot f_4 - (s_{C_{2a3b}} \cdot f_2 + s_{C_{1a}} \cdot f_3 + s_{C_{1a}} \cdot f_4 + s_{C_{2a4b}} \cdot f_4) \\ = & s_{C_2} \cdot f_3 + s_{C_2} \cdot f_4 - (s_{C_{2a3b}} \cdot f_2 + s_{C_{C2a4b}} \cdot f_4) \\ > & 0 \end{aligned}$$

The above formula proves that $cost_{\lambda_2}(Q^4) > cost_{\lambda_3}(Q^4)$. This result shows that a finer derived fragmentation approach can lead to better system performance than derived fragmentation based on a fragmentation schema of one owner relation.

6.2.2 Some Terms

We shall now define some terms to facilitate our discussion of derived horizontal fragmentation. Let $db(E_{ji}) = \{t | t \in \sigma_{\varphi_j}(db(E_i))\}$ denote the set of tuples of E_i accessed by query Q_j.

As we do not restrict derived fragmentation to be performed on a member relation according to only one of its owner relation, in the complex data model we introduce the terms *target type* and *related type*, which have broader meanings than owner relations.

Definition 6.12. A *target type* is a database type that is not primarily horizontally fragmented and is accessed together with other database types that are either at its lower level or at its high level and have been fragmented using predicates. □

Definition 6.13. A *related type* of a *target* type is a type that has been horizontally fragmented and is accessed by queries together with the target type. □

Definition 6.14. The *request* of a fragment E_i^k at a site θ over a network is the sum of the frequencies of all the queries that are issued at site θ and access E_i^k:

$$request_\theta(E_i^k) = \sum_{j=1, \lambda(Q_j)=\theta, db(E_{ji}) \cap db(E_i^k) \neq \emptyset}^{m} f_j$$

□

Definition 6.15. The *affinity* between a target type E_d and one fragment E_i^k of its related type is the sum of the frequencies of all the queries Q_j accessing E_d and the fragment E_i^k together at site θ:

$$aff_\theta(E_d, E_i^k) = \sum_{j=1, \lambda(Q_j)=\theta, db(E_{ji}) \cap db(E_i^k) \neq \emptyset, db(E_{ji}) \cap db(E_d) \neq \emptyset}^{m} f_j$$

□

When there is more than one fragmentation schema of a given target type, we define *atomic derived horizontal fragments* (or *atomic fragments* for short).

Definition 6.16. *Atomic derived horizontal fragments*, or *atomic fragments*, are the intersections of fragments of one fragmentation schema with fragments of another fragmentation schema. For example, $F_E = \{E_1, \dots, E_m\}$ and $F'_E = \{E'_1, \dots, E'_n\}$, then $db(E_i) \cap db(E'_j)$ is an atomic fragment. □

According to our discussion of how horizontal fragmentation affects query costs, the allocation of fragments to network nodes, following the cost minimization heuristics, already determine the location assignment provided that an optimal location assignment for the queries was given prior to the fragmentation. Horizontal fragmentation will only change the subqueries in the form of $(*)$. In this case we evaluate a derived horizontal fragmentation by comparing the total query costs before and after the fragmentation. After derived horizontal fragmentation the instance of a database type will be replaced by a set of atomic fragments which are allocated to network nodes that lead to the least query costs.

Taking the cost model introduced in Chapter 3, we now investigate how total query costs are affected by derived horizontal fragmentation. Assume there are two related types A and B, and one target type C. Let $C_{iai'b}$ be the atomic fragment, λ_1 indicate a distribution design without derived horizontal fragmentation, λ_2 indicate a distribution design with derived horizontal fragmentation and fragment allocation, $\lambda(Q_j)$ be the optimal allocation of the root of subqueries in the form $(*)$, x indicate the site that C is allocated to before it is derived horizontal fragmented, and y denote an optimal allocation of atomic fragment $C_{iai'b} = C_{ia} \cap C_{i'b}$. As the transportation costs dominate the total query costs, we get the following formulae:

$$
\begin{aligned}
& cost_{\lambda_1}(Q^m) - cost_{\lambda_2}(Q^m) \\
= & \sum_{j=1}^{m} cost_{\lambda_1}(Q_j) \cdot f_j - \sum_{j=1}^{m} cost_{\lambda_2}(Q_j) \cdot f_j \\
= & \sum_{j=1}^{m} (\sum_{h_1} \sum_{h'_1} c_{\lambda(h'_1)\lambda(h_1)} \cdot s(h'_1)) \cdot f_j \\
& - \sum_{j=1}^{m} (\sum_{h_2} \sum_{h'_2} c_{\lambda(h'_2)\lambda(h_2)} \cdot s(h'_2)) \cdot f_j \\
= & \sum_{j=1}^{m} c_{\lambda(Q_j)x} \cdot s_C \cdot f_j - \sum_{j=1}^{m} (\sum_{i=1}^{k} \sum_{i'=1}^{k} s_{C_{iai'b}} \cdot c_{\lambda(Q_j)y}) \cdot f_j
\end{aligned}
$$

In order to maximize the value of $cost_{\lambda_1}(Q^m) - cost_{\lambda_2}(Q^m)$ we need to minimize the value of $\sum_{j=1}^{m} (\sum_{i=1}^{k} \sum_{i'=1}^{k} s_{C_{iai'b}} \cdot c_{\lambda(Q_j)y}) \cdot f_j$. For a single atomic fragment $C_{iai'b}$, we need to minimize the value of $\sum_{j=1}^{m} c_{\lambda(Q_j)y} \cdot f_j$. This leads to a heuristic which allocates atomic fragments $C_{iai'b}$ to a site that accesses it most often by queries Q_j together with a related fragment, either A_i or $B_{i'}$. The optimal allocation is $y = \lambda(Q_j)$ in which case $c_{\lambda(Q_j)y} = 0$. That is, we allocate the atomic fragment $C_{iai'b}$ to a site that request it most often. This will maximize the local data availability for the most frequent queries. The accesses of $C_{iai'b}$ by queries are either with A_i, $B_{i'}$ or none of them. Therefore the difference between $\mathit{aff}(C, A_i)$ and $\mathit{aff}(C, B_{i'})$ will reflect the local *request* at sites i and i', and indicate the difference between $request_i(C)$ and

$request_{i'}(C)$. In other words, we should allocate an atomic fragment to the same site of the related fragment which has the highest *affinities* with the target type. In the following section we shall investigate a heuristic procedure for derived horizontal fragmentation.

6.2.3 Heuristics for Derived Horizontal Fragmentation

We perform derived horizontal fragmentation with the following steps. Read and write queries are not distinguished because replication is not considered at this stage.

1. Take the most frequently used 20% queries Q^m.
2. Process primary horizontal fragmentation using the heuristic in Section 6.1 to get a set of primary horizontal fragmentation schemata.
3. Get a set of target types, those which are database types that have not been fragmented primarily but are accessed together with some related fragments.
4. For each of the target types find the set of queries that accessed both the target type and corresponding related types and get the frequencies of each query.
5. For each of the target types get the *request* of each fragment of the related data types.
6. Use fragmentation of each of the related types to perform derived fragmentation of the target type. Allocate resulting fragments to the same site of the corresponding related fragment involved in the semijoin. Remove overlaps between each pair of the resulting fragments. A overlap part is allocated to the same site as the related fragment that is requested the most by queries.
7. If there is more than one derived fragmentation schema from step 3 perform derived fragmentation refinement by performing intersection between every pair of fragments from two different schemata to get a set of atomic fragments.
8. Allocate atomic fragments to the same site as the related fragments that have the highest affinity with the target type.

This procedure is formally described by the algorithm below.

Algorithm 6.17. [Cost-Based Derived Horizontal Fragmentation Algorithm]
Input: $Q^m = \{Q_1, \dots, Q_m\}$ /* a set of global queries
E_d /* a type with a set of components and attributes
a set of network nodes $N = \{1, \dots, k\}$ with cost factors
a set of related types $\{E_1, \dots, E_i, \dots, E_c\}$ with $F_{E_i} = \{E_i^1, \dots, E_i^k\}$
Output: derived horizontal fragmentation schema and fragment allocation schema
Method

```
for each θ ∈ {1,...,k} let db(E_d^θ) = ∅ endfor
for each related type E_i ∈ {E_1,...,E_c} do
  db(E_di^θ) = db(E_d) ⋉ db(E_i^θ)
endfor
for each fragment db(E_di^θ) /* remove overlaps
  for each fragment db(E_di^θ') do
    db(E_di^θθ') = db(E_di^θ) ∩ db(E_di^θ')
    choose y such that request(E_i^y) = min{request(E_i^θ), request(E_i^θ')}
    db(E_di^y) = db(E_di^y) − db(E_di^θθ') /* remove intersection from the less request node
  endfor
endfor
```

```
for each db(E^θ_di) do
   for each db(E^θ'_di') do
      db(E^θθ'_dii') = db(E^θ_di) ∩ db(E^θ'_di')
      aff_θ(E_d, E^z_x) = max{aff_θ(E_d, E^θ_i), aff_θ(E_d, E^θ'_i')}
      db(E^z_d) = db(E^z_d) ∪ db(E^θθ'_dij)
   endfor
endfor
```
□

6.2.4 An Example

Recollect that in [24] member relations and owner relations are defined as the relation at the tail of a join link and a relation at the head of the link, respectively. In our complex data model the link between a member database type and a owner database type is simply a link between a database type and one of its components. For example, if $E_1 \in comp(E_2)$, then E_1 is the owner database type and E_2 is the member database type.

EXAMPLE 6.2. Taking again the database schema in Example 3.1, we now assume for a target database type LECTURER, there are two related database types, DEPARTMENT and TEACH, each of which has been horizontally fragmented into three fragments that are allocated to network nodes, 1, 2, and 3, respectively, i.e., $F_{\text{DEPARTMENT}} = \{\text{DEPARTMENT}_1, \text{DEPARTMENT}_2, \text{DEPARTMENT}_3\}$, $F_{\text{TEACH}} = \{\text{TEACH}_1, \text{TEACH}_2, \text{TEACH}_3\}$. To perform derived horizontal fragmentation of type LECTURER we go through the following procedure:

1. With each related type semijoin is performed to get a set of horizontal fragments. Remove the overlap between each pair of fragments.
   Type LECTURER can then be derived horizontally fragmented according to the fragmentation schemata of DEPARTMENT and TEACH. The fragments resulting from semijoin with fragments of DEPARTMENT are:
   $db(\text{LECTURER}_{1D'}) = db(\text{LECTURER}) \ltimes db(\text{DEPARTMENT}_1)$,
   $db(\text{LECTURER}_{2D'}) = db(\text{LECTURER}) \ltimes db(\text{DEPARTMENT}_2)$,
   $db(\text{LECTURER}_{3D'}) = db(\text{LECTURER}) \ltimes db(\text{DEPARTMENT}_3)$,
   $db(\text{LECTURER}_{4D'}) = db(\text{LECTURER}) - (db(\text{LECTURER}) \ltimes db(\text{DEPARTMENT}))$
   Because DEPARTMENT is a component of LECTURER, the above fragments are disjoint. Also, all objects for DEPARTMENT in LECTURER must be in one of fragments of DEPARTMENT. Hence, $\text{LECTURER}_{4D'}$ is always an empty set. Therefore, we can directly get the following disjoint fragments:
   $$\text{LECTURER}_{1D}, \text{LECTURER}_{2D}, \text{LECTURER}_{3D}.$$
   Similarly, fragmenting LECTURER by performing semijoin with fragments of TEACH we get
   $db(\text{LECTURER}_{1T'}) = db(\text{LECTURER}) \ltimes db(\text{TEACH}_1)$,
   $db(\text{LECTURER}_{2T'}) = db(\text{LECTURER}) \ltimes db(\text{TEACH}_2)$,
   $db(\text{LECTURER}_{3T'}) = db(\text{LECTURER}) \ltimes db(\text{TEACH}_3)$,
   $db(\text{LECTURER}_{4T'}) = db(\text{LECTURER}) - (db(\text{LECTURER}) \ltimes db(\text{TEACH}))$.
   Because LECTURER is a component of TEACH there might be overlaps between the above fragments. Removing any overlap we get the following disjoint fragments:
   $\text{LECTURER}_{1T}, \text{LECTURER}_{2T}, \text{LECTURER}_{3T}, \text{LECTURER}_{4T}$.

2. Perform intersection between all fragments resulted from semijoin with DEPARTMENT and fragments resulted from semijoin with fragments of TEACH, we now have 12 intersections:

   $db(\text{LECTURER}_{1D1T}) = db(\text{LECTURER}_{1D}) \cap db(\text{LECTURER}_{1T})$,
   $db(\text{LECTURER}_{1D2T}) = db(\text{LECTURER}_{1D}) \cap db(\text{LECTURER}_{2T})$,
   $db(\text{LECTURER}_{1D3T}) = db(\text{LECTURER}_{1D}) \cap db(\text{LECTURER}_{3T})$,
   $db(\text{LECTURER}_{1D4T}) = db(\text{LECTURER}_{1D}) \cap db(\text{LECTURER}_{4T})$,
   $db(\text{LECTURER}_{2D1T}) = db(\text{LECTURER}_{2D}) \cap db(\text{LECTURER}_{1T})$,
   $db(\text{LECTURER}_{2D2T}) = db(\text{LECTURER}_{2D}) \cap db(\text{LECTURER}_{2T})$,
   $db(\text{LECTURER}_{2D3T}) = db(\text{LECTURER}_{2D}) \cap db(\text{LECTURER}_{3T})$,
   $db(\text{LECTURER}_{2D4T}) = db(\text{LECTURER}_{1D}) \cap db(\text{LECTURER}_{4T})$,
   $db(\text{LECTURER}_{3D1T}) = db(\text{LECTURER}_{3D}) \cap db(\text{LECTURER}_{1T})$,
   $db(\text{LECTURER}_{3D2T}) = db(\text{LECTURER}_{3D}) \cap db(\text{LECTURER}_{2T})$,
   $db(\text{LECTURER}_{3D3T}) = db(\text{LECTURER}_{3D}) \cap db(\text{LECTURER}_{3T})$,
   $db(\text{LECTURER}_{3D4T}) = db(\text{LECTURER}_{3D}) \cap db(\text{LECTURER}_{4T})$.

3. For each of the intersections we decide its allocation based on the allocation of corresponding related fragments and their frequencies. Two situations may occur:
   - *Intersections result from fragments at the same site.* Among the above intersections, $\text{LECTURER}_{1D1T}, \text{LECTURER}_{2D2T}$, LECTURE
     -R_{3D3T} are resulted from the semijoin of the fragments at the same network node. Therefore we allocate them at the same site as these related fragments.
     $\lambda(\text{LECTURER}_{1D1T}) = 1$, $\lambda(\text{LECTURER}_{2D2T}) = 2$, $\lambda(\text{LECTURER}_{3D3T}) = 3$,
   - *Intersections result from fragments at different sites.* In this case the affinities between related types involved in the intersections and the target type will be used to decide the allocation of the atomic fragment. The related fragments that have highest affinity with the target type will decide the allocation of the atomic derived fragment.
     For example, the allocation of LECTURER_{1D2T} will be decided by the values of $\mathit{aff}(\text{LECTURER}, \text{DEPARTMENT}_1)$ and $\mathit{aff}(\text{LECTURER}, \text{TEACH}_2)$. If $\mathit{aff}(\text{LECTURER}, \text{DEPARTMENT}_1) = \text{MAX}\{(\mathit{aff}(\text{LECTURER}, \text{DEPARTMENT}_1), \mathit{aff}(\text{LECTURER}, \text{TEACH}_2)\}$ then allocate C_{1D2T} to site 1. Otherwise allocate it to site 2.

□

EXAMPLE 6.3. Looking back at the motivation example in Section 6.2.1, there are four atomic fragments, $C_{1a3b}, C_{1a4b}, C_{2a3b}, C_{2a4b}$. To allocate these atomic fragments we compare affinities. For example, to allocate C_{1a3b} we compare affinities $\mathit{aff}(C, A_1), \mathit{aff}(C, B_3)$. As $\mathit{aff}(C, A_1) = f_1$ and $\mathit{aff}(C, B_3) = f_3$ and $f_1 > f_3$. Hence, we allocate C_{1a3b} to site 1.
In the same way we have:

$\lambda(C_{1a4b}) = 1$ because $\mathit{aff}(C, A_1) = f_1$, $\mathit{aff}(C, B_4) = f_4$ and $f_1 > f_4$.
$\lambda(C_{2a3b}) = 3$ because $\mathit{aff}(C, A_2) = f_2$, $\mathit{aff}(C, B_3) = f_3$ and $f_3 > f_2$.
$\lambda(C_{2a4b}) = 2$ because $\mathit{aff}(C, A_2) = f_2$,$\mathit{aff}(C, B_4) = f_4$ and $f_2 > f_4$.

The allocation resulted from the heuristic using affinities is the same as the optimized allocation in the example in Section 6.2.1.

□

6.2.5 Discussion

In this section I will prove that the proposed approach for derived horizontal fragmentation is correct with regard to the criteria in Section 4.4. In addition I will analyze the complexity of the proposed approach.

Let C be an instance of a target database type, A and B be the instances of two related types, which are fragmented as $F_A = \{A_1, \ldots, A_m\}$, $F_B = \{B_1, \ldots, B_n\}$. Database type C has attributes in common with A and B. That means that C is either a component of A and B or has A or B as a component. The following shows that criteria of fragmentation, disjointness, reconstruction, completeness, are satisfied.

- **Completeness:** As shown in the definition of derived horizontal fragmentation, there is always a remainder which contains all instances of C which do not match instances in A or B. In other words, if an object cannot be selected using semijoin with fragments of A or B, it will be in the remainder fragment.
- **Disjointness:** To check disjointness between each pair of atomic derived fragments, C_{iajb} and $C_{i'aj'b}$, we can check whether $C_{iajb} \cap C_{i'aj'b} = \emptyset$ for the following three situations: $i = i'$, $j = j'$ and $i \neq i' \wedge j \neq j'$.
  For the first two situations the proofs of disjointness are straightforward. Because C_{jb} and $C_{j'b}$ are disjoint, the disjointness between C_{iajb} and $C_{iaj'b}$ is guaranteed:

$$\begin{aligned} C_{iajb} \cap C_{iaj'b} &= (C_{ia} \cap C_{jb}) \cap (C_{ia} \cap C_{j'b}) \\ &= C_{ia} \cap (C_{jb} \cap C_{j'b}) \\ &= \emptyset \end{aligned}$$

  Similarly, because C_{ia} and $C_{i'a}$ are disjoint C_{iajb} and $C_{i'ajb}$ is disjoint.

$$\begin{aligned} C_{iajb} \cap C_{i'ajb} &= (C_{ia} \cap C_{jb}) \cap (C_{i'a} \cap C_{jb}) \\ &= C_{jb} \cap (C_{ia} \cap C_{i'a}) \\ &= \emptyset \end{aligned}$$

  For general cases $i \neq i' \wedge j \neq j'$.

$$\begin{aligned} C_{iajb} \cap C_{i'aj'b} &= (C_{ia} \cap C_{jb}) \cap (C_{i'a} \cap C_{j'b}) \\ &= (C_{ia} \cap (C_{i'a}) \cap (C_{jb} \cap C_{j'b}) \\ &= \emptyset \end{aligned}$$

- **Reconstruction:** The formulae below show that the union of all atomic derived fragments reconstruct the original instance C.

$$\begin{aligned} \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} C_{iajb} &= \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} (C_{ia} \cap C_{jb}) \\ &= \bigcup_{i=1}^{m} (C_{ia} \cap \bigcup_{j=1}^{n} C_{jb}) \\ &= \bigcup_{i=1}^{m} (C_{ia} \cap C) \\ &= C \cap \bigcup_{i}^{m} C_{ia} \\ &= C \cap C \\ &= C \end{aligned}$$

This approach is of similar computation complexity as traditional approaches using fragmentation of one owner relation but can lead to better system performance. Let c be the number of related types of a given target type, n be the number of records of the target type, n' be the average number of records of related types, k be the number of network nodes, m be the number of queries. The complexity of our approach, which deals with derived fragmentation and allocation, is $O(c \cdot n' \cdot \log n' + k \cdot n \cdot \log n + c \cdot m \cdot k^2)$. The first *for* loop does not affect the computational complexity. The second *for* loop, performing semijoins, takes time $O(c \cdot n' \cdot \log n' + n \cdot \log n)$. The third *for* loop, removing overlaps between fragments, takes time $O(k \cdot n \cdot \log n + c \cdot m \cdot k)$. The third *for* loop, allocating atomic fragments, takes time $O(c \cdot m \cdot k^2)$. For example, if there are two related types for a target type, the complexity of the traditional approach is $O(n' \cdot \log n' + n \cdot \log n + m \cdot k^2)$ while our approach is $O(2 \cdot n' \cdot \log n' + n \cdot \log n + 2 \cdot m \cdot k^2)$. The complexity for the one time design procedure does not change very much while the system performance can indeed be improved, for the long-term using of the system.

6.2.6 Experimental Evaluation

This subsection presents some experiments conducted to verify Algorithm 6.17, proposed above. They used the same testbed as in Section 6.1.4, a testbed designed with a database schema $\mathcal{S}$ populated with records to get $db(\mathcal{S})$. Assume that from four sites over a network, there are 30 queries, either the 20% most frequently queries or those used by most critical transactions. These 30 queries were designed by applying the similar pattern of queries as in the OO7 project [22]. According to the well-known 20/80 rule, the system performance is assessed by the total query costs of these 30 queries. Some of the types were accessed by queries with predicates while other types, *target types*, were accessed by queries though joining with *related types* or directly. To test the heuristic queries were designed such that there were two target types, each of which had two related types. The types that had predicates defined on them had been horizontally fragmented using the heuristics proposed in Section 6.1. With these fragmented related types derived horizontal fragmentation was performed on the target types using the following two approaches:

- case I: using Algorithm 6.17 introduced above (Section 6.2.3).
- case II: using the traditional approach based on fragments of one owner type as in [93].

Tests were run on three different instances of different sizes. Comparing the results produced the following table:

case	I	II
Instance 1	$21079 \cdot 10^6$	$21152 \cdot 10^6$
Instance 2	$78111 \cdot 10^6$	$78456 \cdot 10^6$
Instance 3	$136565 \cdot 10^6$	$136724 \cdot 10^6$

The experimental results show that the total query costs for case I is smaller than that for case II on all three different database instances. This means that the heuristic approach for derived horizontal fragmentation can lead to better system performance than can traditional approaches that use fragmentation of one owner relation. This validates the proposed heuristic approach for derived horizontal fragmentation. Furthermore, the time to process the tests using this approach was of similar length to that using the traditional approach.

Chapter 7

Heuristics for Vertical Fragmentation and Fragment Allocation

In the literature, fragmentation is treated as independent design procedure from allocation. As discussed in Section 2.2, for distributed databases vertical fragmentation are mainly affinity-based [88, 89, 93]. Allocation takes as input the output of the vertical fragmentation. However, once the fragmentation decision has been made, the possibility of finding an optimal allocation schema of the fragments is restricted. Therefore, a cost model should come into play while making decision of fragmentation. In this chapter I concentrate on vertical fragmentation.

7.1 A Motivating Example

We first look at the following example to see the restriction of finding an optimal allocation for a given fragmentation schema which has been decided at the fragmentation stage using attribute affinities with no cost models involved.

Example 7.1. Consider a relation being fragmented into several fragments according to the affinities between each pair of attributes. Among these fragments there is one fragment F_u having four attributes a_1, a_2, a_3, a_4, of length $\ell_1, \ell_2, \ell_3, \ell_4$, respectively, and being accessed by two queries with accessing frequencies f_1, f_2 respectively. Query Q_1 needs to access attributes a_1, a_2, a_3, a_4 while Q_2 needs to access a_3, a_4. First we assume both Q_1 and Q_2 are issued at site N_1. The optimal allocation is to allocate F_u to site N_1, in which case there are no transportation costs needed to execute both Q_1 and Q_2. This scenario is illustrated in Figure 7.1.
Now assume that Q_2 is issued at site N_2. The change in the issuing site of Q_2 does not affect the weights of edges on the affinity graph. Therefore the fragmentation schema would be the same. For F_u the optimal allocation depends on the values of f_1 and f_2. There are two situations that may occur:

- if $f_2 \leq (1 + \frac{\ell_1 + \ell_2}{\ell_3 + \ell_4}) \cdot f_1$ then $\lambda(F_u) = N_1$
- if $f_2 > (1 + \frac{\ell_1 + \ell_2}{\ell_3 + \ell_4}) \cdot f_1$ then $\lambda(F_u) = N_2$

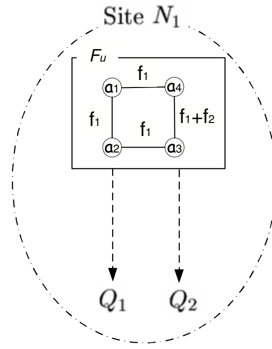

Fig. 7.1. Scenario I: Fragmentation with Two Queries at One Location

The first situation is illustrated in Figure 7.2 as Scenario II. In this scenario, transportation costs are involved to execute Query Q_2. As the transportation costs dominate the total query costs, we get:

$$costs_{\lambda_1}(Q^2) \approx trans_{\lambda_1}(Q_2) = f_2 \cdot n \cdot (\ell_3 + \ell_4) \cdot c_{12}$$

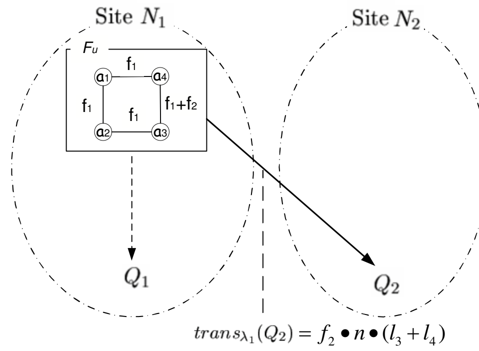

Fig. 7.2. Scenario II: Fragmentation with Two Queries at Different Locations

The second situation is illustrated in Figure 7.3 as Scenario III. In this scenario, transportation costs are involved to execute query Q_1. Then, we get:

$$costs_{\lambda_2}(Q^2) \approx trans_{\lambda_1}(Q_1) = f_1 \cdot n \cdot (\ell_1 + \ell_2 + \ell_3 + \ell_4) \cdot c_{21}$$

Assuming $f_2 > f_1$, if we do not perform vertical fragmentation using affinities but rather consider both vertical fragmentation and allocation together to find an optimal solution, we can have two fragments $F_{u1} = \{a_1, a_2\}$ $F_{u2} = \{a_3, a_4\}$ and allocate F_{u2} to site N_2. This fragmentation and allocation are illustrated in Scenario IV (Figure 7.4). In this case, total query costs are

$$cost_{\lambda_3}(Q^2) \approx trans_{\lambda_2}(Q_1) = f_1 \cdot n \cdot (\ell_3 + \ell_4) \cdot c_{21}$$

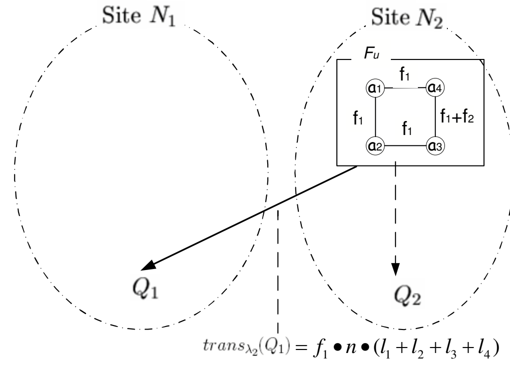

Fig. 7.3. Scenario III: Fragmentation with Two Queries at Different Locations

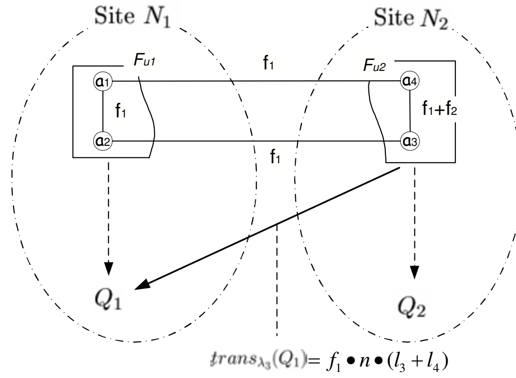

Fig. 7.4. Scenario IV: Fragmentation with Two Queries at Different Locations

Assuming $c_{12} = c_{21} = 1$, we compare query costs in Scenario IV to those in Scenario II and get

$$\begin{aligned} cost_{\lambda_1}(Q^2) &= cost_{\lambda_3} + f_2 \cdot n \cdot (\ell_3 + \ell_4) - f_1 \cdot n \cdot (\ell_3 + \ell_4) \\ &= cost_{\lambda_3} + n \cdot (\ell_3 + \ell_4) \cdot (f_2 - f_1) \\ &> cost_{\lambda_3} \end{aligned}$$

Comparing query costs in Scenario IV to those in Scenario III, we get

$$\begin{aligned} cost_{\lambda_2}(Q^2) &= cost_{\lambda_3} + f_1 \cdot n \cdot (\ell_1 + \ell_2 + \ell_3 + \ell_4) - f_1 \cdot n \cdot (\ell_3 + \ell_4) \\ &= cost_{\lambda_3} + f_1 \cdot n \cdot (\ell_1 + \ell_2) \\ &> cost_{\lambda_3} \end{aligned}$$

□

Obviously, the fragmentation and allocation in Scenario IV, which results from comparing with query costs while making decision of fragmentation, is the best solution. That is, to get an optimal solution of fragmentation and allocation we need to employ a cost model with which we can achieve an optimal fragmentation and allocation simultaneously. With the cost model, any change of the query information (including the site information of queries) will

be reflected in the design of fragmentation and allocation. But the approaches of vertical fragmentation using the value of affinities cannot reflect this change.

However, if a relation has m non-primary key attributes, the possible fragments are given by the *Bell* number which is approximately $B(m) \approx m^m$. With this number of possible fragments, the following up allocation, using the cost model introduced previously, is of complexity $O(k^{(m^m)})$ with k as the number of network nodes. Therefore, it is impossible to get optimal solutions to the vertical fragmentation and allocation problems. We can only expect to find a heuristic solution.

In the remainder of this chapter a heuristic approach to vertical fragmentation and allocation will be introduced. This approach adapts a simplified cost model while making decision of vertical fragmentation. It adopts the vertical fragmentation approach in [78, 79] for the RDM to the complex value data model described in Chapter 3. Research work in this chapter can also be found in [55, 73]. In the following, some terms will be defined before the presentation of the algorithm for vertical fragmentation for the complex value data model and examples illustrating how the algorithm is applied and the system performance is changed.

7.2 Some Terms

If $db(E)$ is vertically fragmented into a set of fragments with fragmentation schema $F_E = \{E_{V1}, \dots, E_{Vu}, \dots, E_{Vk_i}\}$, each of the fragments will be allocated to one of the network nodes $N_1, \dots, N_\theta, \dots, N_k$. Note that the maximum number of fragments is k, i.e. $k_i \leq k$. Let $\lambda(Q_j)$ indicate the site issuing query Q_j and $atomic_j = \{a_i | f_{ji} = f_j\}$ indicate the set of atomic attributes accessed by Q_j, with f_{ji} as the frequency of the query Q_j accessing a_i. Here, $f_{ji} = f_j$ if the attribute a_i is accessed by Q_j. Otherwise, $f_{ji} = 0$. Each attribute a_i of E is of average length ℓ_i.

From the discussion in the previous section, Section 7.1, we know that to get an optimal vertical fragmentation we need to employ a cost model which takes input information as:

- the frequency of queries that access the object (taking that when the same query is issued at different sites, it is treated as different queries);
- the subset of the attributes used by queries;
- the size of each attribute of the object.
- the site that issue the queries

Definition 7.1. The *request* of an attribute a_i at a site θ is the sum of the frequencies of all queries at the site accessing the attribute:

$$request_\theta(a_i) = \sum_{j=1,\lambda(Q_j)=\theta}^{m} f_{ji}.$$

□

Definition 7.2. With the length ℓ_i of an attribute a_i and average number of tuples n in a $db(E)$, we can calculate the *need* of an attribute as the total data volume involved to retrieve a_i by all the queries:

$$need_\theta(a_i) = \sum_{j=1,\lambda(Q_j)}^{m} s_i \cdot f_{ji}.$$

□

We can also calculate the *need* of an attribute using the *request* of the attribute.

$$\begin{aligned} need_\theta(a_i) &= n \cdot \ell_i \cdot \sum_{j=1,\lambda(Q_j)=\theta}^{m} f_{ji} \\ &= n \cdot \ell_i \cdot request_\theta(a_i) \end{aligned}$$

Finally, a term *pay* is introduced to measure the costs of accessing a single attribute once it is allocated to a network node.

Definition 7.3. The *pay* of allocating an attribute a_i to a site θ measures the costs of accessing attribute a_i by all queries from sites θ' other than site θ:

$$pay_\theta(a_i) = \sum_{\theta'=1,\theta'\neq\theta}^{k} \sum_{j=1,\lambda(Q_j)=\theta'}^{m} f_{ji} \cdot c_{\theta\theta'}.$$

□

Note that attribute length is not included in the formula because when we compare the *pay* of an attribute at different sites, attribute length will always be the same.

It can also be calculated using the following formula:

$$pay_\theta(a_i) = \sum_{\theta'=1,\theta\neq\theta'}^{k} request_{\theta'}(a_i) \cdot c_{\theta\theta'}$$

Definition 7.4. Let f_{ji} be the frequency of a query accessing an attribute a_i of a fragment F_u of a type E, ℓ_i be the length of the attribute. The *need* of a fragment at a site θ is calculated with the following formula:

$$\begin{aligned} need_\theta(F_u) &= \sum_{j=1,\lambda(Q_j)=\theta}^{m} s_j \cdot f_j \\ &= \sum_{j=1,\lambda(Q_j)=\theta}^{m} n \cdot (\sum_{i=1,a_i\in A_j}^{n} \ell_i) \cdot f_j \\ &= \sum_{j=1,\lambda(Q_j)=\theta}^{m} n \cdot (\sum_{i=1}^{n} \ell_i \cdot f_{ji}) \\ &= \sum_{i=1}^{n} n \cdot \ell_i \cdot \sum_{j=1,\lambda(Q_j)=\theta}^{m} \cdot f_{ji} \\ &= \sum_{i=1}^{n} need_\theta(a_i) \end{aligned}$$

with s_j as the size of data volume required by query Q_j from fragment F_u, A_j as the set of attributes accessed by Q_j. □

In distributed databases, costs of queries are dominated by the costs of data transportation from a remote site to the site that issued the queries. To compare different vertical fragmentation schemata we would like to compare how it affects the transportation costs. So we can simplify the cost model in Section 3.3 as following:

$$cost_{\lambda}(Q^m) = \sum_{\theta=1}^{k} \sum_{u=1}^{v} need_{\theta}(F_u) \cdot c_{\theta\theta'}. \tag{4}$$

Note that the cost factor $c_{\theta\theta'} = 0$, if $\theta = \theta'$.

EXAMPLE 7.2. Assume a fragment F being accessed by three queries from three different sites, a, b and c, respectively. If we allocate fragment F to site c, $\lambda(F) = c$, then the costs of all queries that access this attribute can be calculated by summing up the *need* at site a multiplied by the cost factor c_{ca} and the *need* at site b multiplied by the cost factor c_{cb}. Using Formula (4) we have:

$$cost_{\lambda(F)=c}(Q^m) = need_a(F) \cdot c_{ca} + need_b(F) \cdot c_{cb}$$

□

7.3 A Heuristic Approach for Vertical Fragmentation

As in [88], we assume a simple transaction model where the system collects the information at the issuing site of the query and executes the query there. In this model we can evaluate the costs of allocating a single attribute to network nodes and then make decisions by choosing a site that leads to the least query costs. Also, according to our discussion of how fragmentation affects query costs, the allocation of fragments to network nodes following the cost minimization heuristics already determines the location assignment, provided that an optimal location assignment for the queries was given prior to the fragmentation.

Taking the simplified cost model introduced above we now analyze the relationships between the *cost*, the *pay* and the *request* of an attribute. We compute the following formulae:

$$\begin{aligned}
cost_{\lambda(a_i)=\theta}(Q^m) &= \sum_{\theta'=1,\theta\neq\theta'}^{k} need_{\theta'}(a_i) \cdot c_{\theta\theta'} \\
&= \sum_{\theta'=1,\theta'\neq\theta}^{k} \sum_{j=1,\lambda(Q_j)=\theta'}^{m} n \cdot \ell_i \cdot f_{ji} \cdot c_{\theta\theta'} \\
&= n \cdot \ell_i \cdot \sum_{\theta'=1,\theta'\neq\theta}^{k} \sum_{j=1,\lambda(Q_j)=\theta'}^{m} f_{ji} \cdot c_{\theta\theta'} \\
&= n \cdot \ell_i \cdot \sum_{\theta'=1,\theta'\neq\theta}^{k} request_{\theta'}(a_i) \cdot c_{\theta\theta'} \\
&= n \cdot \ell_i \cdot pay_{\theta}(a_i).
\end{aligned}$$

The above formulae gives rise to two alternative heuristics for the allocation of an attribute a_i $(i = 1, \ldots, n)$.

– The first heuristic allocates a_i to a network node N_w such that $pay_w(a_i)$ is minimal, i.e., we choose a network node in such a way that the total transport costs for all queries arising from the allocation are minimized.
– The second heuristic allocates a_i to a network node N_w such that $request_w(a_i)$ is maximal. i.e., we choose the network node with the highest *request* of the attribute a_i. This guarantees that there are no transport costs associated with the data of attribute a_i for those queries that need the data of a_i most frequently. In addition, the availability of data of attribute a_i will be maximized.

Taking the first heuristic, first occurring in [78], we perform vertical fragmentation using six steps below. Read and write queries are distinguished because replication is not considered at this stage. The second heuristic is easily formulated. It is actually a special case of the first heuristic when a simple query environment is assumed.

1. Take the most frequently used 20% queries Q^m.
2. Optimize all the queries and construct an $AUFM$ for each database type E based on the queries.
3. Calculate the *request* at each site for each attribute to construct an Attribute Request Matrix.
4. Calculate the *pay* at each site for each attribute to construct an Attribute Pay Matrix.
5. Cluster all attributes to the site which has the lowest value for *pay*.
6. Attach the primary key to each of the fragments.

In order to record query information, an *Attribute Usage Frequency Matrix* ($AUFM$) is used to record frequencies of queries, the set of atomic attributes accessed by the queries and the sites that issue the queries. Each row in the $AUFM$ represents one query Q_j; and the head of each column contains the set of atomic attributes of a given representation type t_E, the site issuing the queries and the frequency of the queries. We do not distinguish between references and attributes, but record them in the same matrix. The values on a column indicate the frequencies f_{ji} of the queries Q_j that use the corresponding atomic attribute a_i grouped by the site that issues the queries. We treat any two queries issued at different sites as different queries, even if the queries themselves are the same. The $AUFM$ is constructed according to optimized queries in order to record all the attribute requirements returned by queries as well as all the attributes used in some join predicates. If a query returns all the information of an attribute then all its sub-attributes are accessed by the query.

This procedure is formally described as the algorithm below. With the $AUFM$ as an input, we now present a vertical fragmentation algorithm (as described in Algorithm 7.5). For each atomic attribute at each site, the algorithm first calculates the *request* and then calculates the *pay*. At last, all atomic attributes are clustered to the site that has the lowest value of the *pay*. Meanwhile, a set of path expressions for each vertical fragment are obtained. Vertical fragmentation is performed by using the sets of paths. A vertical fragmentation schema and an allocation schema are obtained simultaneously.

Algorithm 7.5. [Cost-Based Vertical Fragmentation Algorithm]
Input: the $AUFM$ of database type E
$atomic(E) = \{a_1, \dots, a_n\}$ /* a set of atomic attributes
$PATH(E) = \{path_i, \dots, path_n\}$ /* a set of path of all atomic attributes
a set of network nodes $N = \{1, \dots, k\}$ with cost factors c_{ij}

Output: vertical fragmentation and fragment allocation schema $F_E = \{E_{V1}, \ldots, E_{Vk}\}$
Method:
```
for each θ ∈ {1,...,k}
    let atomic(E_Vθ) = k_E
endfor
for each attribute a_i ∈ atomic(E), 1 ≤ i ≤ n do
    for each node θ ∈ {1,...,k}
        do calculate request_θ(a_i)
    endfor
    for each node θ ∈ {1,...,k}
        do calculate pay_θ(a_i)
    endfor
    choose w such that pay_w(a_i) = min_{θ=1}^{k} pay_θ(a_i)
    atomic(E_Vw) = atomic(E_Vw) ∪ {a_i}      /* add a_i to E_Vw
    PATH(E_Vw) = PATH(E_Vw) ∪ {path_i}      /* add path_i to PATH(E_Vw)
endfor
for each θ ∈ {1,...,k}
    db(E_Vw) = π_PATH(E_Vw)(db(E))
endfor
```
□

7.4 Examples

EXAMPLE 7.3. We now illustrate the algorithm using an example. Assume there are five queries that constitute the 20% most frequently executed queries which access an instance of database type LECTURER from three different sites:

- Query 1 $\pi_{t_{\text{LECTURER}}}(\sigma_{\text{name.titles}\ni\text{'Professor'}}(\text{LECTURER}))$ issued at site 1 with $f_1 = 20$,
- Query 2 $\pi_{\text{name.titles, homepage}}(\text{LECTURER})$ issued at site 2 with $f_2 = 30$,
- Query 3 $\pi_{\text{name.lname, phone}}(\text{LECTURER})$ issued at site 3 with $f_3 = 100$,
- Query 4 $\pi_{\text{name.fname, email}}(\text{LECTURER})$ issued at site 1 with $f_4 = 50$, and
- Query 5 $\pi_{\text{name.titles, phone.areacode}}(\text{LECTURER})$ issued at site 2 with $f_5 = 70$.

In order to perform vertical fragmentation using the design procedure as introduced in Section 7.3, we first construct an Attribute Usage Frequency Matrix ($AUFM$) as shown in Table 7.1. Secondly, we compute the *request* for each attribute at each site, the results of which are shown in the Attribute Request Matrix in Table 7.2. Thirdly, assuming the values of transportation cost factors are: $c_{12} = c_{21} = 10$, $c_{13} = c_{31} = 25$, $c_{23} = c_{32} = 20$, we can now calculate the *pay* for each attribute at each site using the values of the *request* from Table 7.2. The results are shown in an Attribute Pay Matrix in Table 7.3.
Once atomic attributes are grouped and allocated to sites, we get a set of paths for each site to be used for vertical fragmentation:

- $db(E_{V1}) = \pi_{\text{id, in.dname, name.fname, email}}(db(\text{LECTURER}))$,
- $db(E_{V2}) = \pi_{\text{id, name.titles.title, homepage}}(db(\text{LECTURER}))$ and
- $db(E_{V3}) = \pi_{\text{id, name.lname, phone}}(db(\text{LECTURER}))$.

We now look at how the system performance is changed due to the outlined fragmentation by using the cost model presented above. Assume that the average number of lecturers is

Table7.1. Attribute Usage Frequency Matrix for Example 7.3

Site	Query	Frequency	in	id	name			email	phone		homepage
			dname		fname	lname	titles		areacode	number	
							title				
Length			30	20	$20 \cdot 8$	$20 \cdot 8$	$2 \cdot 15 \cdot 8$	$30 \cdot 8$	10	20	$50 \cdot 8$
1	Q_1	20	20	20	20	20	20	20	20	20	20
	Q_4	50	0	0	50	0	0	50	0	0	0
2	Q_2	30	0	0	0	30	30	0	0	0	30
	Q_5	70	0	0	0	0	70	0	70	0	0
3	Q_3	100	0	0	0	100	0	0	100	100	0

Table7.2. Attribute Request Matrix for Example 7.3

request	in	id	name			email	phone		homepage
	dname		fname	lname	titles		areacode	number	
					title				
$request_1(a_i)$	20	20	70	20	20	70	20	20	20
$request_2(a_i)$	0	0	0	30	100	0	70	0	30
$request_3(a_i)$	0	0	0	100	0	0	100	100	0

Table7.3. Attribute Pay Matrix for Example 7.3

pay	in	id	name			email	phone		homepage
	dname		fname	lname	titles		areacode	number	
					title				
$pay_1(a_i)$	0	0	0	2800	1000	0	3200	2500	300
$pay_2(a_i)$	200	200	700	2200	200	700	2200	2200	200
$pay_3(a_i)$	500	500	1750	1100	2500	1750	1900	500	1100
site N_θ	1	1,2,3	1	3	2	1	3	3	2

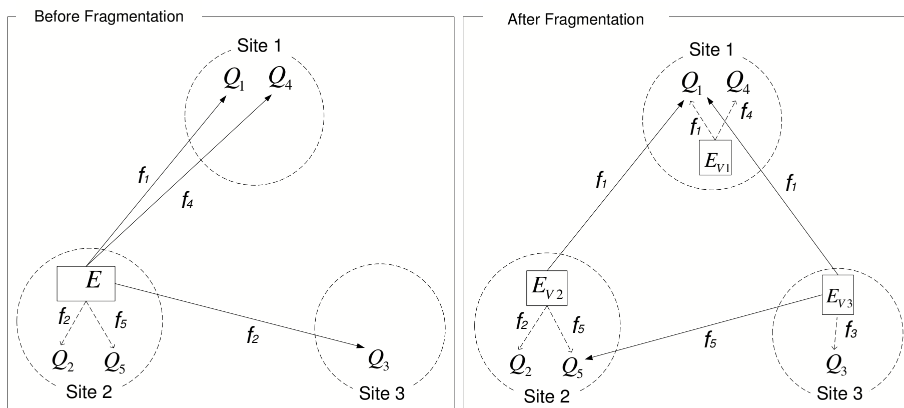

Fig. 7.5. Allocation of Fragments for Example 7.3

20 and the average number of titles for each lecturer is 2. With the average length of each attribute given in Table 7.1, we can compute the total query costs. Assume that distributed query processing and optimization are supported, then selection and projection should be processed first locally to reduce the size of data transported among different sites. In this case, the optimized allocation of db(LECTURER) is site 2, which leads to total query costs of 16,680,000 while the total query costs after the vertical fragmentation and allocation are 4,754,000, which is about one fourth of the costs before the fragmentation. This shows that vertical fragmentation can indeed improve system performance. □

EXAMPLE 7.4. We will take the same example problem as issued in [88, 89] to illustrate how the cost-based approach works and to compare resulting fragmentation schema with that in [88, 89]. Firstly, we take the Attribute Usage Matrix and Attribute Access Matrix in [88] to construct an $AUFM$ grouped by site that issues the queries. The $AUFM$ is shown in Table 7.4. Secondly, we compute the *request* for each attribute at each site and get the Attribute Request Matrix shown in Table 7.5.

Table7.4. Attribute Usage Frequency Matrix for Example 7.4

Site	Query	Frequency	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
1	T_1	10	10	0	0	0	10	0	10	0	0	0
1	T_2	10	0	10	10	0	0	0	0	10	10	0
1	T_4	10	0	10	0	0	0	0	10	10	0	0
1	T_6	10	10	0	0	0	10	0	0	0	0	0
1	T_5	5	5	5	5	0	5	0	5	5	5	0
1	T_7	5	0	0	5	0	0	0	0	0	5	0
1	T_8	5	0	0	5	5	0	5	0	0	5	5
2	T_2	20	0	20	20	0	0	0	0	20	20	0
2	T_1	15	15	0	0	0	15	0	15	0	0	0
2	T_5	10	10	10	10	0	10	0	10	10	10	0
2	T_7	10	0	0	10	0	0	0	0	0	10	0
2	T_6	5	5	0	0	0	5	0	0	0	0	0
2	T_8	5	0	0	5	5	0	5	0	0	5	5
3	T_3	15	0	0	0	15	0	15	0	0	0	15
3	T_4	15	0	15	0	0	0	0	15	15	0	0
3	T_2	10	0	10	10	0	0	0	0	10	10	0
3	T_5	5	5	5	5	0	5	0	5	5	5	0
3	T_6	5	5	0	0	0	5	0	0	0	0	0
3	T_7	5	0	0	5	0	0	0	0	0	5	0
3	T_8	3	0	0	3	3	0	3	0	0	3	3
4	T_2	10	0	10	10	0	0	0	0	10	10	0
4	T_3	10	0	0	0	10	0	10	0	0	0	10
4	T_4	10	0	10	0	0	0	0	10	10	0	0
4	T_5	5	5	5	5	0	5	0	5	5	5	0
4	T_6	5	5	0	0	0	5	0	0	0	0	0
4	T_7	5	0	0	5	0	0	0	0	0	5	0
4	T_8	2	0	0	2	2	0	2	0	0	2	2
Length			10	8	4	6	15	14	3	5	9	12

Table7.5. Attribute Request Matrix for Example 7.4

$request$	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
$request_1(a_i)$	25	25	25	5	25	5	25	25	25	5
$request_2(a_i)$	30	30	45	5	30	5	25	30	45	5
$request_3(a_i)$	10	30	23	18	10	18	20	30	23	18
$request_4(a_i)$	10	25	22	12	10	12	15	25	22	12

Thirdly, assuming we have been given the values of transportation cost factors in Table 7.6, we can now calculate the *pay* of each attribute at each site using the values of the *request* in Table 7.5 and values of cost factors in Table 7.6. The results are shown in an Attribute Pay Matrix in Table 7.7.

Table7.6. Transportation Cost Factors for Example 7.4

site	1	2	3	4
1	0	10	25	20
2	10	0	20	15
3	25	20	0	15
4	20	15	15	0

Table7.7. Attribute Pay Matrix for Example 7.4

pay	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
$pay_1(a_i)$	750	1550	1465	740	750	740	1050	1550	1465	740
$pay_2(a_i)$	600	1225	1040	590	600	590	875	1225	1040	590
$pay_3(a_i)$	1375	1600	1855	405	1375	405	1350	1600	1855	405
$pat_4(a_i)$	1100	1400	1520	445	1100	445	1175	1400	1520	445

Finally, for each attribute we compare values of the *pay* at all sites to find the minimal one. We then allocate attribute a_i to the site that of the minimal *pay*. The allocation of attributes is shown in Table 7.8.

Relation R has been fragmented into two fragments with $F_1 = \{a_1, a_2, a_3, a_5, a_7, a_8, a_9\}$ and $F_2 = \{a_4, a_6, a_{10}\}$, which have been allocated to site 2 and 3 respectively.

Let us now compare our fragmentation schema with the fragmentation decision in [88, 89]. In [88, 89] the relation is first fragmented into three fragments: $F_1 = \{a_1, a_5, a_7\}$, $F_2 = \{a_2, a_3, a_8, a_9\}$ and $F_3 = \{a_4, a_6, a_{10}\}$. Then, at the refinement stage, F_1 and F_2 are allocated at the same site. That means that the final fragmentation schema is $\{a_1, a_2, a_3, a_5, a_7, a_8, a_9\}, \{a_4, a_6, a_{10}\}$, which is same as our results. However, the cost-based approach presented in this chapter is of less complexity.

Take the example problem in [88] again and move all queries T_2 from site 2 to site 1. In this case, the affinity graph does not change; therefore the resulting fragmentation schema is not changed. But using Algorithm 7.5 results a different fragmentation schema: $\{a_2, a_8\}, \{a_1, a_3, a_5, a_7\}, \{a_4, a_6, a_{10}\}$.

Table7.8. Attribute Allocation for Example 7.4

attribute a_i	1	2	3	4	5	6	7	8	9	10
site N_θ	2	2	2	3	2	3	2	2	2	3

To compare the two solutions the simplified cost model introduced in Chapter 3 is used. The total query costs result from fragmentation schema $\{a_2, a_8\}, \{a_1, a_3, a_5, a_7\}$, $\{a_4, a_6, a_{10}\}$ is 64255 and the optimal allocation of fragmentation schema $\{a_1, a_5, a_7\}$, $\{a_2, a_3, a_8, a_9\}$, $\{a_4, a_6, a_{10}\}$ is 64580. Obviously, Algorithm 7.5 leads to a better design because the change of input query information is reflected in the decision of fragmentation to reduce the total query costs. □

It can be concluded that the cost-based heuristic approach to vertical fragmentation presented in this chapter addresses the deficiencies of affinity-based vertical fragmentation approaches, e.g., [38, 57, 86, 88, 89, 93] (which make decisions according to the affinities between each pair of attributes), by introducing a simplified cost model at the stage of vertical fragmentation. Also, the Cost-Base Vertical Fragmentation Algorithm is of lower complexity than are vertical fragmentation algorithms using affinities.

7.5 Discussion

In order to obtain optimized fragmentation and allocation schemata for complex value databases, a cost model should be involved to evaluate the system performance. However, due to the complexity of fragmentation and allocation it is practically impossible to achieve an optimal fragmentation allocation schema by exhaustively comparing different fragmentation schemata and allocation schemata using the cost model. From the analysis of the cost model above, we observe that the less the value of the *pay* of allocating an attribute or an atomic fragment to a site the less the total costs to access it will be [78]. This explains how the proposed cost-based fragmentation approach above can at least determine a semi-optimal vertical fragmentation schema.

Using the Cost-Based Vertical Fragmentation Algorithm, Algorithm 7.5, it can be always guaranteed that the resulting vertical fragmentation schema meets the criteria of correctness rules. Disjointness and completeness are satisfied because all non-key atomic attributes occur and only occur in one of the fragments. Reconstruction is guaranteed because all fragments are composed of key attributes. In addition, if an attribute with a domain of a type inside a set type is decomposed, an index is attached to the attribute before vertical fragmentation. This index will then will be attached to each of fragments.

The advantages of our heuristic approach for vertical fragmentation and fragment allocation are:

- Except key attributes, there is no overlap among all the vertical fragments. Therefore, we do not need extra procedures to remove overlaps.
- The change of queries will be reflected by the fragmentation solution. Query information may reflect the need to retain attributes from some sites more often than from some other sites even though on the affinity graph the cutting edges will be the same.

- Let m be the number of queries, n be the number of attributes, k be the number of network nodes. The complexity of our vertical fragmentation and allocation approach is $O(m \cdot n + k^2 \cdot n)$. The first *for* loop does not affect the computational complexity. The second *for* loop is executed n times. At each iteration, computing *request* for all sites takes time $O(m)$, computing *pay* for all sites takes time $O(k^2)$, and choosing a site of lowest value of *pay* takes time $O(k \cdot \log k)$. The last *for* loop takes time $O(k \cdot n \cdot \log n)$.
  However, the complexity of the graphical approach in [89] is $O(n^2 \cdot m + k^{n'})$ for the whole design procedure, where the complexity of building the affinity matrix is $O(n^2 \cdot m)$, the complexity of vertical fragmentation is $O(n^2)$, and complexity of fragment allocation is $O(k^{n'})$ with n' as the number of resulting fragments.
- This approach suits the situation where for each database type the number of attributes is small and the number of queries is big. Usually, the number of queries taken into consideration is bigger than the number of attributes of a database type.

Chapter 8

Conclusions

8.1 Summary

This thesis addressed the distribution design for complex value databases, which are characterized by the presence of several type constructors following the abstract model in [101]. It focused particularly on the problem of query performance gain that is expected from a distribution. Therefore, it first investigated optimized query trees utilizing a simplified query algebra. The reason for choosing this algebra is that it gives a reasonable handle to estimate total query costs by first estimating the sizes of intermediate results, then summing up storage and transportation costs under the assumption that nodes in the query trees are allocated to network nodes. This defines a rather accurate query cost model. In addition, the use of a query algebra gives us a handle to take query optimization into account, which amounts to a reorganization of query trees.

On this basis we analyzed what would happen to query costs if first an optimized query tree would be rewritten by fragmenting a leaf node, then optimized again. A discussion of horizontal, vertical and splitting fragmentation following previous work in [80, 102] disclosed that only subqueries involving a selection followed by a projection are affected by fragmentation. Further, it was found that if an optimal allocation is assumed before fragmentation, changes will only become necessary to leaf nodes and the selection/projection predecessors. This result – summarized in Facts 5.1-5.3 – is a significant improvement on previous work in the context of the RDM (e.g. [5], [7] and [64]), in which query trees were only considered in a static way, while present work dynamically reorganizes these query trees after fragmentation. Furthermore, the result of the investigation of this study implies that query optimization including optimal allocation of intermediate nodes in query trees to network nodes is largely orthogonal to the problem of fragmentation and allocation. In other words, finding an adequate fragmentation and a suitable allocation of the fragments to network nodes will result in at least sub-optimal total query costs if the queries as such are optimized before fragmentation.

This result was then used on the impact of fragmentation on query costs to develop a heuristic approach to primary horizontal fragmentation and allocation. The resulting heuristic approach consists of three parts. The first part determines a sub-optimal allocation for given fragments placing fragments either at nodes, where they are most needed, or at nodes that will minimize transport costs arising from the given fragment. At this point the corresponding heuristic algorithm still has a rather high complexity. The second part refines a given fragmentation and allocation, and checks whether further fragmentation may lead to further

improvement of query costs including the determination of an optimal allocation of the new fragments. This gives rise to a binary search which either refines fragments or removes unnecessary fragmentation. The third part determines a reasonable number of selection predicates following a heuristic according to which it can be expected that, below a certain threshold of query frequency, selection predicates will no longer contribute to improving query performance. These integrated heuristics are validated by a number of experiments, the positive results of which confirm the validity of the approach.

For derived horizontal fragmentation, the deficiencies of the approaches that perform derived horizontal fragmentation on a member relation or a non-leaf class based on fragmentation of one owner relation or leaf class were discussed. To overcome the deficiencies, in the context of the complex value data model, a heuristic procedure was proposed to allow derived fragmentation to be performed by using horizontal fragmentation schemata of more than one owner relation or leaf class. By doing this, derived horizontal fragmentation could be refined to further improve the system performance. In addition, the derived horizontal fragmentation algorithm in this thesis allowed derived horizontal fragmentation to be performed not only on the owner type of a link but also on the member type of the link. The release of the restriction that derived fragmentation can only be performed on the instances of a member type according to fragmentation of one of its owner types rendered derived horizontal fragmentation more applicable to general cases. The cost-based derived fragmentation heuristic takes into consideration data localization and the fact that the biggest number of fragments finally generated is the number of network notes. Experimental evaluation confirmed the validity of the heuristic on derived horizontal fragmentation.

The allocation heuristics and the fragmentation and allocation refinement were generalized to vertical fragmentation. It was shown that affinity-based approaches cannot really improve the data locality and thus improve system performance. A cost-based heuristics which takes into consideration of query information, including not only frequencies but also where the queries are issued, was proposed. In the meantime the network information, captured by transportation cost factors, were also counted. A heuristic approach was proposed based on the analysis of the cost model with the transportation costs dominating the total query costs, and the consideration that database distribution design aims at improving system performance by increasing data local availability. To compare this approach with the existing affinity-based vertical fragmentation algorithms (see e.g., [88][89]), the vertical fragmentation algorithm proposed in this thesis is applied to the same example problem (as in [88] and [89]), on which many other affinity-based vertical fragmentation algorithms have been based. It is shown that the approach presented in this study out-performs the affinity-based approaches in [88] and [89] with regard to both efficiency and computation complexity. In addition, the approach can be applied not only to relational databases but also to complex value databases.

8.2 Open Problems and Challenges

Open problems existed include consideration of concurrency in the cost model. The cost model is employed to predict the execution costs of alternative fragmentation and allocation designs. The main objective of database distribution design is to improve system performance and throughput. The problem here is how to factor the knowledge of workload of concurrent queries within the cost model. This may lead to the consideration of multiple query optimization, where a set of queries executed in the same time period are optimized together.

Another open problem relates to global query optimization. Current optimization techniques focus only on a subset of queries, namely select-project-join (SPJ) queries with conjunctive predicates. As a result, a large number of theories have been proposed for join and semijoin ordering. However, for queries with other operations, such as aggregations and unions, there is no complete theory one can apply with respect to distributed query optimization for complex value databases. Therefore, it is natural to study the affect of fragmentation on queries that are not restricted only to SPJ queries.

Further, it is worthwhile to consider disjoint union type in the type system because it can be used to represent alternatives, which are used in XML. Reconstructivity of fragmentation may not, however, be guaranteed once vertical fragmentation is performed on a disjoint union type. For example, if a vertical fragmentation is applied to an attribute of type $(a_1 : t_2, a_2 : (t_{21} \oplus t_{22}))$ with two fragments of type $(a_1 : t_1, a_2 : (t_{21}))$ and $(a_1 : t_1, a_2 : (t_{22}))$, respectively, using a join operation cannot reconstruct the original instance. That means, semantic information is lost when vertical fragmentation is applied on attributes of disjoint union type. Some design techniques need to be developed to assure the reconstructivity when disjoint type is considered.

Furthermore, distributed databases are often used to support e-commerce, for which security is a critical issue. When encryption is used before data is transported and decryption is used after data is transported, the sizes of data to be transported and the time needed to process queries would be greater than when encryption is not used. An open problem arises regarding how to evaluate query costs when encryption is engaged while processing distributed queries accessing a distributed database.

At last, one can also investigate how to achieve combined gain of fragmentation, allocation and indexing, especially for XML documents, the storage of which heavily relies on the index mechanisms used. For example, the latest version of IBM DB2 [90] supports the storage of XML into tables using the new XML data type. Internally, XML and relational data are stored in different formats which match their corresponding data models. XML columns are stored on disk pages in tree structures matching the XML data model. For that, index structures are needed to allow efficient storage of global context for XML documents. Therefore, the estimation of sizes of XML documents or its fragments depends on how XML label nodes and data nodes are indexed and stored.

Bibliography

1. Abiteboul, S., Buneman, P., and Suciu, D. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.
2. Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.
3. Abiteboul, S., and Kanellakis, P. C. Object identity as a query language primitive. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (1989), ACM Press, pp. 159–173.
4. Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. The LOREL query language for semi-structured data. *International Journal on Digital Libraries 1*, 1 (1997), 68–88.
5. Ahmad, I., Karlapalem, K., Kwok, Y.-K., and So, S.-K. Evolutionary algorithms for allocating data in distributed database systems. *Distributed Parallel Databases 11*, 1 (2002), 5–32.
6. Andrade, A., Ruberg, G., Baião, F. A., Braganholo, V. P., and Mattoso, M. Efficiently processing xml queries over fragmented repositories with partix. In *International Conference on Extending Database Technology (EDBT) Workshops* (2006), Lecture Notes in Computer Science, Springer, pp. 150–163.
7. Apers, P. M. G. Data allocation in distributed database systems. *ACM Transactions on Database Systems 13* (1988), 263–304.
8. Baião, F., and Mattoso, M. A mixed fragmentation algorithm for distributed object oriented databases. In *Proceedings of the International Conference Computing and Information (ICCI)* (1998), pp. 141–148.
9. Baião, F., Mattoso, M., and Zaverucha, G. Horizontal fragmentation in object dbms: New issues and performance evaluation. In *Proceedings of the 19th IEEE International Performance, Computing and Communications Conference* (February 2000), IEEE Computer Society Press, pp. 108–114.
10. Baião, F., Mattoso, M., and Zaverucha, G. A distribution design methodology for object dbms. *Distributed and Parallel Databases 16*, 1 (2004), 45–90.
11. Barker, K., and Bhar, S. A graphical approach to allocating class fragments in distributed objectbase systems. *Distributed and Parallel Databases 10*, 3 (2001), 207–239.
12. Bellatreche, L., Karlapalem, K., and Basak, G. Horizontal class partitioning for queries in object oriented databases. Tech. rep., HKUST-CS98-6, 1998.
13. Bellatreche, L., Karlapalem, K., and Basak, G. K. Query-driven horizontal class partitioning for object-oriented databases. In *Database and Expert Systems Applications* (1998), pp. 692–701.

14. Bellatreche, L., Karlapalem, K., and Li, Q. Complex methods and class allocation in distributed object-oriented database systems. In *Object Oriented Information Systems* (1998), pp. 239–256.
15. Bellatreche, L., Karlapalem, K., and Simonet, A. Horizontal class partitioning in object-oriented databases. In *Proceedings of the 8th International Conference on Database and Expert Systems Applications (DEXA)* (1997), Springer-Verlag, pp. 58–67.
16. Bellatreche, L., Karlapalem, K., and Simonet, A. Algorithms and support for horizontal class partitioning in object-oriented databases. *Distributed and Parallel Databases 8*, 2 (2000), 155–179.
17. Bellatreche, L., Simonet, A., and Simonet, M. Vertical fragmentation in distributed object database systems with complex attributes and methods. In *Seventh International Workshop on Database and Expert Systems Applications (DEXA)* (1996), H. T. R. Wagner, Ed., IEEE Computer Society Press, pp. 15–21.
18. Blankinship, R., Hevner, A. R., and Yao, S. B. An iterative method for distributed database design. In *Proceedings of the 17th International Conference on Very Large Data Bases* (1991), G. M. Lohman, A. Sernadas, and R. Camps, Eds., Morgan Kaufmann, pp. 389–400.
19. Bremer, J.-M., and Gertz, M. On distributing xml repositories. In *International Workshop on Web and Databases (WebDB)* (2003), pp. 73–78.
20. Buneman, P., Davidson, S., Fernandez, M., and Suciu, D. Adding structure to unstructured data. In *International Conference on Database Theory (ICDT)* (1997), vol. 1186 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 336–350.
21. Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (1996), ACM Press, pp. 505–516.
22. Carey, M. J., DeWitt, D. J., and Naughton, J. F. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data) 22*, 2 (1993), 12–21.
23. Casey, R. G. Allocation of copies of files in an information network. In *Proceedings of AFIPS SJCC* (1972), vol. 40, AFIPS Press, pp. 617–625.
24. Ceri, S., Navathe, S. B., and Wiederhold, G. Distribution design of logical database schemas. *IEEE Transactions on Software Engineering (TSE) 9*, 4 (1983), 487–504.
25. Ceri, S., Negri, M., and Pelagatti, G. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data* (1982), ACM Press, pp. 128–136.
26. Ceri, S., and Pelagatti, G. *Distributed Databases Principles and System.* McGraw-Hill, New York, 1984.
27. Ceri, S., Pernici, B., and Wiederhold, G. Distributed database design methodologies. *Proceedings of the IEEE 75*, 5 (1987), 533–546.
28. Chakravarthy, S., Muthuraj, J., Varadarajan, R., and Navathe, S. B. An objective function for vertically partitioning relations in distributed databases and its analysis. *Distributed and Parallel Databases 2*, 2 (1994), 183–207.
29. Chang, S.-K. Data base decomposition in a hierarchical computer system. In *Proceedings of the 1975 ACM SIGMOD international conference on Management of data* (1975), ACM Press, pp. 48–53.

30. Chang, S.-K., and Cheng, W.-H. A methodology for structured database decomposition. *IEEE Transactions on Software Engineering (TSE) 6*, 2 (1980), 205–218.
31. Cheng, C.-H., Lee, W.-K., and Wong, K.-F. A genetic algorithm-based clustering approach for database partitioning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C 32*, 3 (2002), 215–230.
32. Chinchwadkar, G. S., and Goh, A. An overview of vertical partitioning in object oriented databases. *The Computer Journal 42*, 1 (1999), 39–50.
33. Chu, P.-C. A transaction oriented approach to attribute partitioning. *Information Systems 17*, 4 (1992), 329–342.
34. Chu, W. W. Optimal file allocation in a multiple computer system. *IEEE Transactions on Computers 18*, 10 (1969), 885–889.
35. Cluet, S., Delobel, C., Siméon, J., and Smaga, K. Your mediators need data conversion! In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data* (1998), ACM Press, pp. 177–188.
36. Colby, L. S. A recursive algebra and query optimization for nested relations. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (1989), ACM Press, pp. 273–283.
37. Corcoran, A. L., and Hale, J. A genetic algorithm for fragment allocation in a distributed database system. In *Proceedings of the 1994 ACM symposium on Applied computing (SAC)* (1994), ACM Press, pp. 247–250.
38. Cornell, D., and Yu, P. A vertical partitioning algorithm for relational databases. In *International Conference on Data Engineering* (1987), pp. 30–35.
39. Cornell, D. W., and Yu, P. S. Site assignment for relations and join operations in the distributed transaction processing environment. In *Proceedings of the Fourth International Conference on Data Engineering* (1988), IEEE Computer Society Press, pp. 100–108.
40. Cornell, D. W., and Yu, P. S. On optimal site assignment for relations in the distributed database environment. *IEEE Transactions on Software Engineering 15*, 8 (1989), 1004–1009.
41. Cornell, D. W., and Yu, P. S. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Transactions on Software Engineering 16*, 2 (1990), 248–258.
42. Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. A query language for XML. *Computer Networks 31*, 11-16 (1999), 1155–1169.
43. Ezeife, C., and Zheng, J. Measuring the performance of database object horizontal fragmentation schemes. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)* (1999), IEEE Computer Society Press, pp. 408–414.
44. Ezeife, C. I., and Barker, K. Horizontal class fragmentation in distributed object based systems. In *Proceedings of the Second Biennial European Joint Conference on Engineering Systems Design and Analysis* (1994), pp. 225–235.
45. Ezeife, C. I., and Barker, K. A comprehensive approach to horizontal class fragmentation in a distributed object based system. *Distributed and Parallel Databases 3*, 3 (1995), 247–272.
46. Ezeife, C. I., and Barker, K. Vertical fragmentation for advanced object models in a distributed object based system. In *Proceedings of the 7th International Conference on Computing and Information* (1995), pp. 613–632.

47. Ezeife, C. I., and Barker, K. Distributed object based design: Vertical fragmentation of classes. *Distributed and Parallel Databases 6*, 4 (1998), 317–350.
48. Frankhauser, P., Fernández, M., Malhotra, A., Rys, M., Siméon, J., and Wadler, P. The XML query algebra. `http://www.w3.org/TR/2001/WD-query-algebra-20010215`, 2001.
49. Fuchs, M., Maloney, M., and Milowski, A. Schema for object oriented XML. `http://www.w3c.org/TR/NOTE-sox`, 1998.
50. Fung, C.-W., Karlapalem, K., and Li, Q. Cost-driven evaluation of vertical class partitioning in object-oriented databases. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA)* (1997), R. W. Topor and K. Tanaka, Eds., vol. 6 of *Advanced Database Research and Development Series*, World Scientific, pp. 11–20.
51. Fung, C.-W., Karlapalem, K., and Li, Q. An evaluation of vertical class partitioning for query processing in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering 14*, 5 (2002), 1095–1118.
52. Fung, C.-W., Karlapalem, K., and Li, Q. Cost-driven vertical class partitioning for methods in object oriented databases. *The VLDB Journal 12*, 3 (2003), 187–210.
53. Gavish, B., and Pirkul, H. Computer and database location in distributed computer systems. *IEEE Transactions on Computers C-35*, 7 (1986), 583–590.
54. Goldfarb, C. F., and Prescod, P. *The XML Handbook*. Prentice Hall, New Jersey, 1998.
55. Hartmann, S., Ma, H., and Schewe, K.-D. Cost-based vertical fragmentation for xml. In *International Workshop on Database Management and Application over Networks - DBMAN* (2007), vol. 4537 of *Lecture Notes in Computer Science*, Springer, pp. 12–24.
56. Hevner, A. R., and Yao, S. B. Query processing in distributed database systems. *IEEE Transactions on Software Engineering (TSE) 5*, 3 (1979), 177–187.
57. Hoffer, J. A., and Severance, D. G. The use of cluster analysis in physical database design. In *Proceedings of the First International Conference on Very Large Data Bases (VLDB)* (1975), pp. 69–86.
58. Huang, Y.-F., and Chen, J.-H. Fragment allocation in distributed database design. *Information Science and Engineering 17* (2001), 491–506.
59. Karlapalem, K., and Li, Q. Partitioning schemes for object oriented databases. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM)* (1995), IEEE Computer Society Press, pp. 42–49.
60. Karlapalem, K., and Li, Q. A framework for class partitioning in object-oriented databases. *Distributed and Parallel Databases 8*, 3 (2000), 333–366.
61. Karlapalem, K., Li, Q., and Vieweg, S. Method induced partitioning schemes for object-oriented databases. In *International Conference on Distributed Computing Systems* (1996), pp. 377–384.
62. Karlapalem, K., and Navathe, S. Materialization of redesigned distributed relational databases. Tech. Rep. HKUST-CS94-14, Department of Computer Science, HKUST, September 1994.
63. Karlapalem, K., Navathe, S. B., and Morsi, M. M. A. Issues in distribution design of object-oriented databases. In *IWDOM* (1992), pp. 148–164.

64. Karlapalem, K., and Pun, N. M. Query-driven data allocation algorithms for distributed database systems. In *Database and Expert Systems Applications* (1997), pp. 347–356.
65. Khalil, N., Eid, D., and Khair, M. Availability and reliability issues in distributed databases using optimal horizontal fragmentation. In *Database and Expert Systems Applications (DEXA)* (1999), T. J. M. Bench-Capon, G. Soda, and A. M. Tjoa, Eds., vol. 1677 of *Lecture Notes in Computer Science*, Springer, pp. 771–780.
66. Kirchberg, M., Riaz-ud-Din, F., Schewe, K.-D., and Tretiakov, A. Towards algebraic query optimisation for xquery. *Journal on Data Semantics VII 4244* (2006), 165–195.
67. Kossmann, D. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR) 32*, 4 (December 2000), 422–469.
68. Levin, K. D., and Morgan, H. L. Optimizing distributed data bases - a framework for research. In *Proceedings of the AFIPS* (1975), vol. 44 of *NCC*, pp. 473–478.
69. Lin, X., Orlowska, M., and Zhang, Y. A graph based cluster approach for vertical partitioning in database design. *Data and Knowledge Engineeering 11*, 2 (1993), 151–169.
70. Lin, X., and Zhang, Y. A new graphical method of vertical partitioning in database design. In *Proceedings of the 4th Australian Database Conference (ADC)* (1993), M. P. P. Maria E. Orlowska, Ed., World Scientific, pp. 131–144.
71. Liu, H.-C., and Yu, J. X. Algebraic equivalences of nested relational operators. *Information Systems 30*, 3 (2005), 167–204.
72. Ma, H. Distribution design in object oriented databases. Master's thesis, Department of Information Systems, Massey University, 2003.
73. Ma, H., and Kirchberg, M. Cost-based fragmentation for complex value databases. In *Proceedings of the 26th International Conference on Conceptual Modeling (ER)* (2007), vol. 4802 of *Lecture Notes in Computer Science*, Springer, pp. 72–86.
74. Ma, H., and Schewe, K.-D. Fragmentation of XML documents. In *Proceedings of the 18th Brazilian Symposium on Databases (SBBD)* (2003), pp. 200–214.
75. Ma, H., and Schewe, K.-D. A heuristic approach to horizontal fragmentation in object oriented databases. In *Proceedings of the 2004 Baltic Conference on Databases and Information Systems* (2004), pp. 31–46.
76. Ma, H., and Schewe, K.-D. A heuristic approach to cost-efficient horizontal fragmentation of XML documents. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE)* (2005), O. Belo, J. Eder, O. Pastor, and J. F. e Cunha, Eds., Lecture Notes in Computer Science, pp. 131–136.
77. Ma, H., and Schewe, K.-D. Query optimisation as part of distribution design for complex value databases. In *Proceedings 15th European - Japanese Conference on Information Modelling and Knowledge Bases (EJC)* (2005), pp. 269–276.
78. Ma, H., Schewe, K.-D., and Kirchberg, M. A heuristic approach to vertical fragmentation incorporating query information. In *Proceedings of the 7th International Baltic Conference on Databases and Information Systems (DB&IS)* (2006), O. Vasilecas, J. Eder, and A. Caplinskas, Eds., IEEE Computer Society Press, pp. 69–76.
79. Ma, H., Schewe, K.-D., and Kirchberg, M. A heuristic approach to fragmentation incorporating query information. In *Databases and Information Systems IV*, O. Vasilecas, J. Eder, and A. Caplinskas, Eds., vol. 155 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2007, pp. 103–116.

80. Ma, H., Schewe, K.-D., and Wang, Q. A heuristic approach to cost-efficient fragmentation and allocation of complex value databases. In *Proceedings of the 17th Australian Database Conference* (2006), CRPIT 49, pp. 119–128.
81. Ma, H., Schewe, K.-D., and Wang, Q. Distribution design for higher-order data models. *Data and Knowledge Engineering 60*, 2 (2007), 400–434.
82. Mahmoud, S., and Riordon, J. S. Optimal allocation of resources in distributed information networks. *ACM Transactions on Database Systems (TODS) 1*, 1 (1976), 66–78.
83. Malinowski, E., and Chakravarthy, S. Fragmentation techniques for distributing object-oriented databases. In *International Conference on Conceptual Modeling (ER)* (1997), D. W. Embley and R. C. Goldstein, Eds., vol. 1331 of *Lecture Notes in Computer Science*, Springer, pp. 347–360.
84. March, S. T., and Rho, S. Allocating data and operations to nodes in distributed database design. *IEEE Transactions on Knowledge and Data Engineering 7*, 2 (1995), 305–317.
85. Menon, M.-S. Allocating fragments in distributed databases. *IEEE Transactions on Parallel and Distributed Systems 16*, 7 (2005), 577–585.
86. Muthuraj, J., Chakravarthy, S., Varadarajan, R., and Navathe, S. B. A formal approach to the vertical partitioning problem in distributed database design. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems* (Jan 1993), pp. 26–34.
87. Navathe, S., Karlapalem, K., and Ra, M. A mixed fragmentation methodology for initial distributed database design. *Journal of Computer and Software Engineering 3*, 4 (1995), 395–426.
88. Navathe, S. B., Ceri, S., Wiederhold, G., and Dour, J. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS) 9*, 4 (1984), 680–710.
89. Navathe, S. B., and Ra, M. Vertical partitioning for database design: A graphical algorithm. *SIGMOD Record 14*, 4 (1989), 440–450.
90. Nicola, M., and van der Linden, B. Native xml support in db2 universal database. In *Proceedings of the 31st international conference on Very large data bases (VLDB)* (2005), VLDB Endowment, pp. 1164–1174.
91. Nwosu, K. On complex object distribution technique for distributed computing systems. In *Proceedings of the 6th International Conference on Computing and Information (ICCI)* (1994).
92. Özsu, M. T., and Valduriez, P. *Distributed Data Management: Unsolved Problems and New Issues*. IEEE Computer Society Press, 1994.
93. Özsu, M. T., and Valduriez, P. *Principles of Distributed Database Systems*. Prentice-Hall, New Jersey, 1999.
94. Ra, M. Horizontal partitioning for distributed database design. In *Advances in Database Research* (1993), M. Orlowska and M. Papazoglou, Eds., World Scientific Publishing, pp. 101–120.
95. Ra, M., and Park, Y.-S. Data fragmentation and allocation for pc-based distributed database design. In *Proceedings of the 3rd International Conference on Database Systems for Advanced Applications (DASFAA)* (1993), World Scientific Press, pp. 90–96.
96. Ramakrishnan, R., and Gehrke, J. *Database Management Systems*. McGra- Hill, Boston, 1998.

97. Rathakrishnan, B., and Kim, J. An extended recursive algebra for nested relations and its optimization. In *Proceedings of Seventeenth Annual International Computer Software and Applications Conference (COMPSAC)* (1993), pp. 145–151.
98. Rothnie, J. B., and Goodman, N. An overview of the preliminary design of sdd-1: A system for distributed databases. In *Berkeley Workshop* (1977), pp. 39–57.
99. Sacca, D., and Wiederhold, G. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems (TODS) 10*, 1 (1985), 29–56.
100. Sarathy, R., Shetty, B., and Sen, A. A constrained nonlinear 0-1 program for data allocation. *European Journal of Operational Research 102*, 3 (November 1997), 626–647.
101. Schewe, K.-D. On the unification of query algebras and their extension to rational tree structures. In *Proceedings of the 12th Australasian Conference on Database Technologies* (2001), M. Orlowska and J. Roddick, Eds., IEEE Computer Society Press, pp. 52–59.
102. Schewe, K.-D. Fragmentation of object oriented and semi-structured data. In *Databases and Information Systems II*, H.-M. Haav and A. Kalja, Eds. Kluwer Academic Publishers, 2003, pp. 1–14.
103. Schewe, K.-D., and Thalheim, B. Fundamental concepts of object oriented databases. *Acta Cybernetica 11*, 4 (1993), 49–84.
104. Shin, D. G., and Irani, K. B. Partitioning a relational database horizontally using a knowledge-based approach. *SIGMOD Record 14*, 4 (1985), 95–105.
105. Shin, D.-G., and Irani, K. B. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Transactions on Software Engineering (TSE) 17*, 9 (1991), 872–883.
106. Son, J. H., and Kim, M. H. An adaptable vertical partitioning method in distributed systems. *Journal of Systems and Software 73*, 3 (2004), 551–561.
107. Tamhankar, A. M., and Ram, S. Database fragmentation and allocation: An integrated methodology and case study. *IEEE Transactions on Systems Management 28*, 3 (1998), 194–207.
108. Teorey, T. J. Distributed database design: a practical approach and example. *SIGMOD Record 18*, 4 (1989), 23–39.
109. Thalheim, B. *Entity-Relationship Modeling: Foundations of Database Technology.* Springer-Verlag, 2000.
110. Ullman, J. D. *Principles of Database Systems.* W. H. Freeman & Co., New York, NY, USA, 1983.
111. World Wide Web Consortium (W3C). XQuery. `http://www.w3c.org/TR/xquery`.
112. World Wide Web Consortium (W3C). XML Schema. `http://www.w3c.org/TR/xmlschema-0`, `http://www.w3c.org/TR/xmlschema-1`, `http://www.w3c.org/TR/xmlschema-2`, 2001.
113. Yao, S. B., Navathe, S. B., and Weldon, J.-L. An integrated approach to database design. In *Data Base Design Techniques I: Requirement and Logical Structures* (New York, 1982), Springer-Verlag, pp. 1–30. Lecture Notes in Computer Science 132.
114. Zhang, Y. On horizontal fragmentation of distributed database design. In *Advances in Database Research* (1993), M. Orlowska and M. Papazoglou, Eds., World Scientific Publishing, pp. 121–130.