Chapter 2
Distributed and Parallel Database Design

A typical database design is a process which starts from a set of requirements and results in the definition of a schema that defines the set of relations. The distribution design starts from this global conceptual schema (GCS) and follows two tasks: *partitioning* (*fragmentation*) and *allocation*. Some techniques combine these two tasks in one algorithm, while others implement them in two separate tasks as depicted in Fig. 2.1. The process typically makes use of some auxiliary information that is depicted in the figure although some of this information is optional (hence the dashed lines in the figure).

The main reasons and objectives for fragmentation in distributed versus parallel DBMSs are slightly different. In the case of the former, the main reason is *data locality*. To the extent possible, we would like queries to access data at a single site in order to avoid costly remote data access. A second major reason is that fragmentation enables a number of queries to execute concurrently (through *interquery parallelism*). The fragmentation of relations also results in the parallel execution of a single query by dividing it into a set of subqueries that operate on fragments, which is referred to as *intraquery parallelism*. Therefore, in distributed DBMSs, fragmentation can potentially reduce costly remote data access and increase inter and intraquery parallelism.

In parallel DBMSs, data localization is not that much of a concern since the communication cost among nodes is much less than in geo-distributed DBMSs. What is much more of a concern is load balancing as we want each node in the system to be doing more or less the same amount of work. Otherwise, there is the danger of the entire system thrashing since one or a few nodes end up doing a majority of the work, while many nodes remain idle. This also increases the latency of queries and transactions since they have to wait for these overloaded nodes to finish. Inter and intraquery parallelism are both important as we discuss in Chap. 8,

---

The original version of this chapter was revised. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-26253-2_13

© Springer Nature Switzerland AG 2020

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_2

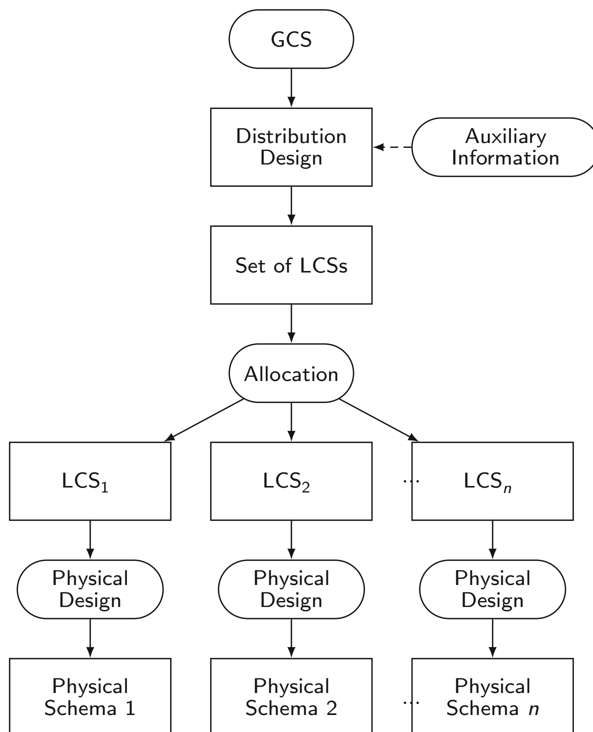


Fig. 2.1 Distribution design process

although some of the modern big data systems (Chap. 10) pay more attention to interquery parallelism.

Fragmentation is important for system performance, but it also raises difficulties in distributed DBMSs. It is not always possible to entirely localize queries and transactions to only access data at one site—these are called *distributed queries* and *distributed transactions*. Processing them incurs a performance penalty due to, for example, the need to perform distributed joins and the cost of distributed transaction commitment (see Chap. 5). One way to overcome this penalty for read-only queries is to replicate the data in multiple sites (see Chap. 6), but that further exacerbates the overhead of distributed transactions. A second problem is related to semantic data control, specifically to integrity checking. As a result of fragmentation, attributes participating in a constraint (see Chap. 3) may be decomposed into different fragments that are allocated to different sites. In this case, integrity checking itself involves distributed execution, which is costly. We consider the issue of distributed data control in the next chapter. Thus, the challenge is to partition[1] and allocate

[1]A minor point related to terminology is the use of terms "fragmentation" and "partitioning": in distributed DBMSs, the term fragmentation is more commonly used, while in parallel DBMSs, data partitioning is preferred. We do not prefer one over the other and will use them interchangeably in this chapter and in this book.

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Fig. 2.2 Example database

the data in such a way that most user queries and transactions are local to one site, minimizing distributed queries and transactions.

Our discussion in this chapter will follow the methodology of Fig. 2.1: we will first discuss fragmentation of a global database (Sect. 2.1), and then discuss how to allocate these fragments across the sites of a distributed database (Sect. 2.2). In this methodology, the unit of distribution/allocation is a fragment. There are also approaches that combine the fragmentation and allocation steps and we discuss these in Sect. 2.3. Finally we discuss techniques that are adaptive to changes in the database and the user workload in Sect. 2.4.

In this chapter, and throughout the book, we use the engineering database introduced in the previous chapter. Figure 2.2 depicts an instance of this database.

2.1 Data Fragmentation

Relational tables can be partitioned either *horizontally* or *vertically*. The basis of horizontal fragmentation is the select operator where the selection predicates determine the fragmentation, while vertical fragmentation is performed by means of the project operator. The fragmentation may, of course, be nested. If the nestings are of different types, one gets *hybrid fragmentation*.

Example 2.1 Figure 2.3 shows the PROJ relation of Fig. 2.2 divided horizontally into two fragments: $\texttt{PROJ}_1$ contains information about projects whose budgets are less than $200,000, whereas $\texttt{PROJ}_2$ stores information about projects with larger budgets. ♦

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

Fig. 2.3 Example of horizontal partitioning

$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

$PROJ_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Fig. 2.4 Example of vertical partitioning

Example 2.2 Figure 2.4 shows the PROJ relation of Fig. 2.2 partitioned vertically into two fragments: $PROJ_1$ and $PROJ_2$. $PROJ_1$ contains only the information about project budgets, whereas $PROJ_2$ contains project names and locations. It is important to notice that the primary key to the relation (PNO) is included in both fragments. ♦

Horizontal fragmentation is more prevalent in most systems, in particular in parallel DBMSs (where the literature prefers the term *sharding*). The reason for the prevalence of horizontal fragmentation is the *intraquery parallelism*[2] that most recent big data platforms advocate. However, vertical fragmentation has been successfully used in *column-store* parallel DBMSs, such as MonetDB and Vertica, for analytical applications, which typically require fast access to a few attributes.

The systematic fragmentation techniques that we discuss in this chapter ensure that the database does not undergo semantic change during fragmentation, such as losing data as a consequence of fragmentation. Therefore, it is necessary to be able to argue about the *completeness* and *reconstructability*. In the case of horizontal fragmentation, *disjointness* of fragments may also be a desirable property (unless we explicitly wish to replicate individual tuples as we will discuss later).

1. *Completeness*. If a relation instance R is decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$, each data item that is in R can also be found in one or more of R_i's. This property, which is identical to the *lossless decomposition* property of

[2]In this chapter, we use the terms "query" and "transaction" interchangeably as they both refer to the system workload that is one of the main inputs to distribution design. As highlighted in Chap. 1 and as will be discussed in length in Chap. 5, transactions provide additional guarantees, and therefore their overhead is higher and we will incorporate this into our discussion where needed.

normalization (Appendix A), is also important in fragmentation since it ensures that the data in a global relation is mapped into fragments without any loss. Note that in the case of horizontal fragmentation, the "item" typically refers to a tuple, while in the case of vertical fragmentation, it refers to an attribute.

2. *Reconstruction*. If a relation R is decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$, it should be possible to define a relational operator ∇ such that

$$R = \nabla R_i, \quad \forall R_i \in F_R$$

The operator ∇ will be different for different forms of fragmentation; it is important, however, that it can be identified. The reconstructability of the relation from its fragments ensures that constraints defined on the data in the form of dependencies are preserved.

3. *Disjointness*. If a relation R is horizontally decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$ and data item d_i is in R_j, it is not in any other fragment R_k $(k \neq j)$. This criterion ensures that the horizontal fragments are disjoint. If relation R is vertically decomposed, its primary key attributes are typically repeated in all its fragments (for reconstruction). Therefore, in case of vertical partitioning, disjointness is defined only on the nonprimary key attributes of a relation.

2.1.1 *Horizontal Fragmentation*

As we explained earlier, horizontal fragmentation partitions a relation along its tuples. Thus, each fragment has a subset of the tuples of the relation. There are two versions of horizontal partitioning: primary and derived. *Primary horizontal fragmentation* of a relation is performed using predicates that are defined on that relation. *Derived horizontal fragmentation*, on the other hand, is the partitioning of a relation that results from predicates being defined on another relation.

Later in this section, we consider an algorithm for performing both of these fragmentations. However, we first investigate the information needed to carry out horizontal fragmentation activity.

2.1.1.1 Auxiliary Information Requirements

The database information that is required concerns the global conceptual schema, primarily on how relations are connected to one another, especially with joins. One way of capturing this information is to explicitly model primary key–foreign key join relationships in a *join graph*. In this graph, each relation R_i is represented as a vertex and a directed edge L_k exists from R_i to R_j if there is a primary key–foreign key equijoin from R_i to R_j. Note that L_k also represents a one-to-many relationship.

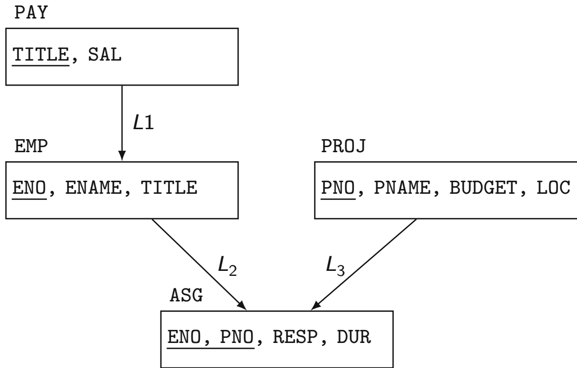


Fig. 2.5 Join graph representing relationships among relations

Example 2.3 Figure 2.5 shows the edges among the database relations given in Fig. 2.2. Note that the direction of the edge shows a one-to-many relationship. For example, for each title there are multiple employees with that title; thus, there is an edge between the PAY and EMP relations. Along the same lines, the many-to-many relationship between the EMP and PROJ relations is expressed with two edges to the ASG relation. ♦

The relation at the tail of an edge is called the *source* of the edge and the relation at the head is called the *target*. Let us define two functions: *source* and *target*, both of which provide mappings from the set of edges to the set of relations. Considering L_1 of Fig. 2.5, $source(L_1) = \texttt{PAY}$ and $target(L_1) = \texttt{EMP}$.

Additionally, the cardinality of each relation R denoted by $card(\texttt{R})$ is useful in horizontal fragmentation.

These approaches also make use of the workload information, i.e., the queries that are run on the database. Of particular importance are the predicates used in user queries. In many cases, it may not be possible to analyze the full workload, so the designer would normally focus on the important queries. There is a well-known "80/20" rule-of-thumb in computer science that applies in this case as well: the most common 20% of user queries account for 80% of the total data accesses, so focusing on that 20% is usually sufficient to get a fragmentation that improves most distributed database accesses.

At this point, we are interested in determining *simple predicates*. Given a relation $\texttt{R}(\texttt{A}_1, \texttt{A}_2, \ldots, \texttt{A}_n)$, where $\texttt{A}_i$ is an attribute defined over domain D_i, a simple predicate p_j defined on R has the form

$$p_j : \texttt{A}_i \; \theta \; Value$$

where $\theta \in \{=, <, \neq, \leq, >, \geq\}$ and *Value* is chosen from the domain of $\texttt{A}_i$ ($Value \in D_i$). We use Pr_i to denote the set of all simple predicates defined on a relation $\texttt{R}_i$. The members of Pr_i are denoted by p_{ij}.

Example 2.4 Given the relation instance PROJ of Fig. 2.2,

$$\text{PNAME} = \text{"Maintenance" and BUDGET} \leq 200000$$

is a simple predicate. ♦

User queries often include more complicated predicates, which are Boolean combinations of simple predicates. One such combination, called a *minterm predicate*, is the conjunction of simple predicates. Since it is always possible to transform a Boolean expression into conjunctive normal form, the use of minterm predicates in the design algorithms does not cause any loss of generality.

Given a set $Pr_i = \{p_{i1}, p_{i2}, \ldots, p_{im}\}$ of simple predicates for relation R_i, the set of minterm predicates $M_i = \{m_{i1}, m_{i2}, \ldots, m_{iz}\}$ is defined as

$$M_i = \{m_{ij} = \bigwedge_{p_{ik} \in Pr_i} p^*_{ik}\},\ 1 \leq k \leq m,\ 1 \leq j \leq z$$

where $p^*_{ik} = p_{ik}$ or $p^*_{ik} = \neg p_{ik}$. So each simple predicate can occur in a minterm predicate in either its natural form or its negated form.

Negation of a predicate is straightforward for equality predicates of the form $Attribute = Value$. For inequality predicates, the negation should be treated as the complement. For example, the negation of the simple predicate $Attribute \leq Value$ is $Attribute > Value$. There are theoretical problems of finding the complement in infinite sets, and also the practical problem that the complement may be difficult to define. For example, if two simple predicates are defined of the form $Lower_bound \leq Attribute_1$, and $Attribute_1 \leq Upper_bound$, their complements are $\neg(Lower_bound \leq Attribute_1)$ and $\neg(Attribute_1 \leq Upper_bound)$. However, the original two simple predicates can be written as $Lower_bound \leq Attribute_1 \leq Upper_bound$ with a complement $\neg(Lower_bound \leq Attribute_1 \leq Upper_bound)$ that may not be easy to define. Therefore, we limit ourselves to simple predicates.

Example 2.5 Consider relation PAY of Fig. 2.2. The following are some of the possible simple predicates that can be defined on PAY.

$$p_1 : \text{TITLE} = \text{"Elect. Eng."}$$

$$p_2 : \text{TITLE} = \text{"Syst. Anal."}$$

$$p_3 : \text{TITLE} = \text{"Mech. Eng."}$$

$$p_4 : \text{TITLE} = \text{"Programmer"}$$

$$p_5 : \text{SAL} \leq 30000$$

The following are *some* of the minterm predicates that can be defined based on these simple predicates.

$$m_1 : \texttt{TITLE} = \text{“Elect. Eng.”} \wedge \texttt{SAL} \leq 30000$$

$$m_2 : \texttt{TITLE} = \text{“Elect. Eng.”} \wedge \texttt{SAL} > 30000$$

$$m_3 : \neg(\texttt{TITLE} = \text{“Elect. Eng.”}) \wedge \texttt{SAL} \leq 30000$$

$$m_4 : \neg(\texttt{TITLE} = \text{“Elect. Eng.”}) \wedge \texttt{SAL} > 30000$$

$$m_5 : \texttt{TITLE} = \text{“Programmer”} \wedge \texttt{SAL} \leq 30000$$

$$m_6 : \texttt{TITLE} = \text{“Programmer”} \wedge \texttt{SAL} > 30000$$

♦

These are only a representative sample, not the entire set of minterm predicates. Furthermore, some of the minterms may be meaningless given the semantics of relation PAY, in which case they are removed from the set. Finally, note that these are simplified versions of the minterms. The minterm definition requires each predicate to be in a minterm in either its natural or its negated form. Thus, m_1, for example, should be written as

$$m_1 : \texttt{TITLE} = \text{“Elect. Eng.”} \wedge \texttt{TITLE} \neq \text{“Syst. Anal.”} \wedge \texttt{TITLE} \neq \text{“Mech. Eng.”}$$
$$\wedge\ \texttt{TITLE} \neq \text{“Programmer”} \wedge \texttt{SAL} \leq 30000$$

This is clearly not necessary, and we use the simplified form.

We also need quantitative information about the workload:

1. *Minterm selectivity*: number of tuples of the relation that would satisfy a given minterm predicate. For example, the selectivity of m_2 of Example 2.5 is 0.25 since one of the four tuples in PAY satisfies m_2. We denote the selectivity of a minterm m_i as $sel(m_i)$.
2. *Access frequency*: frequency with which user applications access data. If $Q = \{q_1, q_2, \ldots, q_q\}$ is a set of user queries, $acc(q_i)$ indicates the access frequency of query q_i in a given period.

Note that minterm access frequencies can be determined from the query frequencies. We refer to the access frequency of a minterm m_i as $acc(m_i)$.

2.1.1.2 Primary Horizontal Fragmentation

Primary horizontal fragmentation applies to the relations that have no incoming edges in the join graph and performed using the predicates that are defined on that relation. In our examples, relations PAY and PROJ are subject to primary horizontal fragmentation, and EMP and ASG are subject to derived horizontal fragmentation. In this section, we focus on primary horizontal fragmentation and devote the next section to derived horizontal fragmentation.

A primary horizontal fragmentation is defined by a selection operation on the source relations of a database schema. Therefore, given relation R its horizontal

fragments are given by

$$\mathtt{R}_i = \sigma_{F_i}(\mathtt{R}),\ 1 \le i \le w$$

where F_i is the selection formula used to obtain fragment $\mathtt{R}_i$ (also called the *fragmentation predicate*). Note that if F_i is in conjunctive normal form, it is a minterm predicate (m_i). The algorithm requires that F_i be a minterm predicate.

Example 2.6 The decomposition of relation PROJ into horizontal fragments $\mathtt{PROJ}_1$ and $\mathtt{PROJ}_2$ in Example 2.1 is defined as follows[3]:

$$\mathtt{PROJ}_1 = \sigma_{\mathtt{BUDGET} \le 200000}(\mathtt{PROJ})$$

$$\mathtt{PROJ}_2 = \sigma_{\mathtt{BUDGET} > 200000}(\mathtt{PROJ})$$

Example 2.6 demonstrates one of the problems of horizontal partitioning. If the domain of the attributes participating in the selection formulas is continuous and infinite, as in Example 2.6, it is quite difficult to define the set of formulas $F = \{F_1,\ F_2,\ \ldots,\ F_n\}$ that would fragment the relation properly. One possible solution is to define ranges as we have done in Example 2.6. However, there is always the problem of handling the two endpoints. For example, if a new tuple with a BUDGET value of, say, $600,000 were to be inserted into PROJ, one would have to review the fragmentation to decide if the new tuple is to go into $\mathtt{PROJ}_2$ or if the fragments need to be revised and a new fragment needs to be defined as

$$\mathtt{PROJ}_2 = \sigma_{200000 < \mathtt{BUDGET} \wedge \mathtt{BUDGET} \le 400000}(\mathtt{PROJ})$$

$$\mathtt{PROJ}_3 = \sigma_{\mathtt{BUDGET} > 400000}(\mathtt{PROJ})$$

Example 2.7 Consider relation PROJ of Fig. 2.2. We can define the following horizontal fragments based on the project location. The resulting fragments are shown in Fig. 2.6.

$$\mathtt{PROJ}_1 = \sigma_{\mathtt{LOC}=\text{“Montreal”}}(\mathtt{PROJ})$$

$$\mathtt{PROJ}_2 = \sigma_{\mathtt{LOC}=\text{“New York”}}(\mathtt{PROJ})$$

$$\mathtt{PROJ}_3 = \sigma_{\mathtt{LOC}=\text{“Paris”}}(\mathtt{PROJ})$$

Now we can define a horizontal fragment more carefully. A horizontal fragment $\mathtt{R}_i$ of relation R consists of all the tuples of R that satisfy a minterm predicate m_i.

[3]We assume that the nonnegativity of the BUDGET values is a feature of the relation that is enforced by an integrity constraint. Otherwise, a simple predicate of the form $0 \le \mathtt{BUDGET}$ also needs to be included in Pr. We assume this to be true in all our examples and discussions in this chapter.

PROJ1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

PROJ2

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

PROJ3

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

Fig. 2.6 Primary horizontal fragmentation of relation PROJ

Hence, given a set of minterm predicates M, there are as many horizontal fragments of relation R as there are minterm predicates. This set of horizontal fragments is also commonly referred to as the set of *minterm fragments*.

We want the set of simple predicates that form the minterm predicates to be *complete* and *minimal*. 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 minterm fragment that is defined according to Pr.[4]

Example 2.8 Consider the fragmentation of relation PROJ given in Example 2.7. If the only query that accesses PROJ wants to access the tuples according to the location, the set is complete since each tuple of each fragment PROJ_i has the same probability of being accessed. If, however, there is a second query that accesses only those project tuples where the budget is less than or equal to $200,000, then Pr is not complete. Some of the tuples within each PROJ_i have a higher probability of being accessed due to this second application. To make the set of predicates complete, we need to add ($\text{BUDGET} \leq 200000$, $\text{BUDGET} > 200000$) to Pr:

$$Pr = \{\text{LOC} = \text{"Montreal"}, \text{LOC} = \text{"New York"}, \text{LOC} = \text{"Paris"},$$
$$\text{BUDGET} \leq 200000, \text{BUDGET} > 200000\}$$

Completeness is desirable because fragments obtained according to a complete set of predicates are logically uniform, since they all satisfy the minterm predicate. They are also statistically homogeneous in the way applications access them. These

[4]Clearly the definition of completeness of a set of simple predicates is different from the completeness rule of fragmentation we discussed earlier.

characteristics ensure that the resulting fragmentation results in a balanced load (with respect to the given workload) across all the fragments.

Minimality states that if a predicate influences how fragmentation is performed (i.e., causes a fragment f to be further fragmented into, say, f_i and f_j), there should be at least one application that accesses f_i and f_j differently. In other words, the simple predicate should be *relevant* in determining a fragmentation. If all the predicates of a set Pr are relevant, Pr is *minimal*.

A formal definition of relevance can be given as follows. Let m_i and m_j be two minterm predicates that are identical in their definition, except that m_i contains the simple predicate p_i in its natural form, while m_j contains $\neg p_i$. Also, let f_i and f_j be two fragments defined according to m_i and m_j, respectively. Then p_i is *relevant* if and only if

$$\frac{acc(m_i)}{card(f_i)} \neq \frac{acc(m_j)}{card(f_j)}$$

Example 2.9 The set Pr defined in Example 2.8 is complete and minimal. If, however, we were to add the predicate PNAME = "Instrumentation" to Pr, the resulting set would not be minimal since the new predicate is not relevant with respect to Pr—there is no application that would access the resulting fragments any differently. ♦

We now present an iterative algorithm that would generate a complete and minimal set of predicates Pr' given a set of simple predicates Pr. This algorithm, called COM_MIN, is given in Algorithm 2.1 where we use the following notation:

Rule 1: each fragment is accessed differently by at least one application.

f_i *of* Pr': fragment f_i defined according to a minterm predicate defined over the predicates of Pr'.

COM_MIN begins by finding a predicate that is relevant and that partitions the input relation. The **repeat-until** loop iteratively adds predicates to this set, ensuring minimality at each step. Therefore, at the end the set Pr' is both minimal and complete.

The second step in the primary horizontal design process is to derive the set of minterm predicates that can be defined on the predicates in set Pr'. These minterm predicates determine the fragments that are used as candidates in the allocation step. Determination of individual minterm predicates is trivial; the difficulty is that the set of minterm predicates may be quite large (in fact, exponential on the number of simple predicates). We look at ways of reducing the number of minterm predicates that need to be considered in fragmentation.

This reduction can be achieved by eliminating some of the minterm fragments that may be meaningless. This elimination is performed by identifying those minterms that might be contradictory to a set of implications I. For example, if $Pr' = \{p_1, p_2\}$, where

Algorithm 2.1: COM_MIN

Input: R: relation; Pr: set of simple predicates
Output: Pr': set of simple predicates
Declare: F: set of minterm fragments
begin
  $Pr' \leftarrow \emptyset$; $F \leftarrow \emptyset$ {initialize}
  find $p_i \in Pr$ such that p_i partitions R according to *Rule* 1
  $Pr' \leftarrow Pr' \cup p_i$
  $Pr \leftarrow Pr - p_i$
  $F \leftarrow F \cup f_i$ {f_i is the minterm fragment according to p_i}
  repeat
    find $p_j \in Pr$ such that p_j partitions some f_k of Pr' according to *Rule* 1
    $Pr' \leftarrow Pr' \cup p_j$
    $Pr \leftarrow Pr - p_j$
    $F \leftarrow F \cup f_j$
    if $\exists p_k \in Pr'$ *which is not relevant* **then**
      $Pr' \leftarrow Pr' - p_k$
      $F \leftarrow F - f_k$
    end if
  until Pr' *is complete*
end

$$p_1 : \ att = value_1$$
$$p_2 : \ att = value_2$$

and the domain of att is $\{value_1, value_2\}$, so I contains two implications:

$$i_1 : \ (att = value_1) \Rightarrow \neg(att = value_2)$$
$$i_2 : \ \neg(att = value_1) \Rightarrow (att = value_2)$$

The following four minterm predicates are defined according to Pr':

$$m_1 : \ (att = value_1) \wedge (att = value_2)$$
$$m_2 : \ (att = value_1) \wedge \neg(att = value_2)$$
$$m_3 : \ \neg(att = value_1) \wedge (att = value_2)$$
$$m_4 : \ \neg(att = value_1) \wedge \neg(att = value_2)$$

In this case the minterm predicates m_1 and m_4 are contradictory to the implications I and can therefore be eliminated from M.

The algorithm for primary horizontal fragmentation, called PHORIZONTAL, is given in Algorithm 2.2. The input is a relation R that is subject to primary horizontal fragmentation, and Pr, which is the set of simple predicates that have been determined according to applications defined on relation R.

Example 2.10 We now consider relations PAY and PROJ that are subject to primary horizontal fragmentation as depicted in Fig. 2.5.

Suppose that there is only one query that accesses PAY, which checks the salary information and determines a raise accordingly. Assume that employee records are managed in two places, one handling the records of those with salaries less than or equal to $30,000, and the other handling the records of those who earn more than $30,000. Therefore, the query is issued at two sites.

The simple predicates that would be used to partition relation PAY are

$$p_1 : \texttt{SAL} \leq 30000$$

$$p_2 : \texttt{SAL} > 30000$$

thus giving the initial set of simple predicates $Pr = \{p_1, p_2\}$. Applying the COM_MIN algorithm with $i = 1$ as initial value results in $Pr' = \{p_1\}$. This is complete and minimal since p_2 would not partition f_1 (which is the minterm fragment formed with respect to p_1) according to Rule 1. We can form the following minterm predicates as members of M:

$$m_1 : \texttt{SAL} < 30000$$

$$m_2 : \neg(\texttt{SAL} \leq 30000) = \texttt{SAL} > 30000$$

Therefore, we define two fragments $F_{\texttt{PAY}} = \{\texttt{PAY}_1, \texttt{PAY}_2\}$ according to M (Fig. 2.7).

Algorithm 2.2: PHORIZONTAL

Input: R: relation; Pr: set of simple predicates
Output: $F_{\texttt{R}}$: set of horizontal fragments of R
begin
  $Pr' \leftarrow$COM_MIN(R, Pr)
  determine the set M of minterm predicates
  determine the set I of implications among $p_i \in Pr'$
  foreach $m_i \in M$ **do**
    if *m_i is contradictory according to I* **then**
      $M \leftarrow M - m_i$
    end if
  end foreach
  $F_{\texttt{R}} = \{\texttt{R}_i | \texttt{R}_i = \sigma_{m_i}\texttt{R}\}, \forall m_i \in M$
end

$\texttt{PAY}_1$

TITLE	SAL
Mech. Eng.	27000
Programmer	24000

$\texttt{PAY}_2$

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000

Fig. 2.7 Horizontal fragmentation of relation PAY

Let us next consider relation PROJ. Assume that there are two queries. The first is issued at three sites and finds the names and budgets of projects given their location. In SQL notation, the query is

```
SELECT PNAME, BUDGET
FROM   PROJ
WHERE  LOC=Value
```

For this application, the simple predicates that would be used are the following:

$$p_1 : \texttt{LOC} = \text{“Montreal”}$$

$$p_2 : \texttt{LOC} = \text{“New York”}$$

$$p_3 : \texttt{LOC} = \text{“Paris”}$$

The second query is issued at two sites and has to do with the management of the projects. Those projects that have a budget of less than or equal to \$200,000 are managed at one site, whereas those with larger budgets are managed at a second site. Thus, the simple predicates that should be used to fragment according to the second application are

$$p_4 : \texttt{BUDGET} \le 200000$$

$$p_5 : \texttt{BUDGET} > 200000$$

Using COM_MIN, we get the complete and minimal set $Pr' = \{p_1, p_2, p_4\}$. Actually COM_MIN would add any two of p_1, p_2, p_3 to Pr'; in this example we have selected to include p_1, p_2.

Based on Pr', the following six minterm predicates that form M can be defined:

$$m_1 : (\texttt{LOC} = \text{“Montreal”}) \wedge (\texttt{BUDGET} \le 200000)$$

$$m_2 : (\texttt{LOC} = \text{“Montreal”}) \wedge (\texttt{BUDGET} > 200000)$$

$$m_3 : (\texttt{LOC} = \text{“New York”}) \wedge (\texttt{BUDGET} \le 200000)$$

$$m_4 : (\texttt{LOC} = \text{“New York”}) \wedge (\texttt{BUDGET} > 200000)$$

$$m_5 : (\texttt{LOC} = \text{“Paris”}) \wedge (\texttt{BUDGET} \le 200000)$$

$$m_6 : (\texttt{LOC} = \text{“Paris”}) \wedge (\texttt{BUDGET} > 200000)$$

As noted in Example 2.5, these are not the only minterm predicates that can be generated. It is, for example, possible to specify predicates of the form

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$$

However, the obvious implications (e.g., $p_1 \Rightarrow \neg p_2 \wedge \neg p_3$, $\neg p_5 \Rightarrow p_4$) eliminate these minterm predicates and we are left with m_1 to m_6.

Looking at the database instance in Fig. 2.2, one may be tempted to claim that the following implications hold:

$$i_8 : \texttt{LOC} = \text{“Montreal”} \Rightarrow \neg(\texttt{BUDGET} > 200000)$$

$$i_9 : \texttt{LOC} = \text{“Paris”} \Rightarrow \neg(\texttt{BUDGET} \leq 200000)$$

$$i_{10} : \neg(\texttt{LOC} = \text{“Montreal”}) \Rightarrow \texttt{BUDGET} \leq 200000$$

$$i_{11} : \neg(\texttt{LOC} = \text{“Paris”}) \Rightarrow \texttt{BUDGET} > 200000$$

However, remember that implications should be defined according to the semantics of the database, not according to the current values. There is nothing in the database semantics that suggest that the implications i_8–i_{11} hold. Some of the fragments defined according to $M = \{m_1, \ldots, m_6\}$ may be empty, but they are, nevertheless, fragments.

The result of the primary horizontal fragmentation of PROJ is to form six fragments $F_{\texttt{PROJ}} = \{\texttt{PROJ}_1, \texttt{PROJ}_2, \texttt{PROJ}_3, \texttt{PROJ}_4, \texttt{PROJ}_5, \texttt{PROJ}_6\}$ of relation PROJ according to the minterm predicates M (Fig. 2.8). Since fragments $\texttt{PROJ}_2$ and $\texttt{PROJ}_5$ are empty, they are not depicted in Fig. 2.8. ♦

2.1.1.3 Derived Horizontal Fragmentation

A derived horizontal fragmentation applies to the target relations in the join graph and is performed based on predicates defined over the source relation of the join graph edge. In our examples, relations EMP and ASG are subject to derived horizontal fragmentation. Recall that the edge between the source and the target relations is defined as an equijoin that can be implemented by means of semijoins.

$\texttt{PROJ}_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

$\texttt{PROJ}_3$

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York

$\texttt{PROJ}_4$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York

$\texttt{PROJ}_6$

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

Fig. 2.8 Horizontal fragmentation of relation PROJ

This second point is important, since we want to partition a target relation according to the fragmentation of its source, but we also want the resulting fragment to be defined *only* on the attributes of the target relation.

Accordingly, given an edge L where $source(L) = \texttt{S}$ and $target(L) = \texttt{R}$, the derived horizontal fragments of R are defined as

$$\texttt{R}_i = \texttt{R} \ltimes \texttt{S}_i, 1 \leq i \leq w$$

where w is the maximum number of fragments that will be defined on R and $\texttt{S}_i = \sigma_{F_i}(\texttt{S})$, where F_i is the formula according to which the primary horizontal fragment $\texttt{S}_i$ is defined.

Example 2.11 Consider edge L_1 in Fig. 2.5, where $source(L_1) = \texttt{PAY}$ and $target(L_1) = \texttt{EMP}$. Then, we can group engineers into two groups according to their salary: those making less than or equal to \$30,000, and those making more than \$30,000. The two fragments $\texttt{EMP}_1$ and $\texttt{EMP}_2$ are defined as follows:

$$\texttt{EMP}_1 = \texttt{EMP} \ltimes \texttt{PAY}_1$$

$$\texttt{EMP}_2 = \texttt{EMP} \ltimes \texttt{PAY}_2$$

where

$$\texttt{PAY}_1 = \sigma_{\texttt{SAL} \leq 30000}(\texttt{PAY})$$

$$\texttt{PAY}_2 = \sigma_{\texttt{SAL} > 30000}(\texttt{PAY})$$

The result of this fragmentation is depicted in Fig. 2.9. ♦

Derived horizontal fragmentation applies to the target relations in the join graph and are performed based on predicates defined over the source relation of the join graph edge. In our examples, relations EMP and ASG are subject to derived horizontal fragmentation. To carry out a derived horizontal fragmentation, three inputs are needed: the set of partitions of the source relation (e.g., $\texttt{PAY}_1$ and $\texttt{PAY}_2$ in Example 2.11), the target relation, and the set of semijoin predicates between

$\texttt{EMP}_1$

ENO	ENAME	TITLE
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E7	R. Davis	Mech. Eng.

$\texttt{EMP}_2$

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E8	J. Jones	Syst. Anal.

Fig. 2.9 Derived horizontal fragmentation of relation EMP

the source and the target (e.g., EMP.TITLE = PAY.TITLE in Example 2.11). The fragmentation algorithm, then, is quite trivial, so we will not present it in any detail.

There is one potential complication that deserves some attention. In a database schema, it is common that there are multiple edges into a relation R (e.g., in Fig. 2.5, ASG has two incoming edges). In this case, there is more than one possible derived horizontal fragmentation of RṪhe choice of candidate fragmentation is based on two criteria:

1. The fragmentation with better join characteristics;
2. The fragmentation used in more queries.

Let us discuss the second criterion first. This is quite straightforward if we take into consideration the frequency that the data is accessed by the workload. If possible, one should try to facilitate the accesses of the "heavy" users so that their total impact on system performance is minimized.

Applying the first criterion, however, is not that straightforward. Consider, for example, the fragmentation we discussed in Example 2.1. The effect (and the objective) of this fragmentation is that the join of the EMP and PAY relations to answer the query is assisted (1) by performing it on smaller relations (i.e., fragments), and (2) by potentially performing joins in parallel.

The first point is obvious. The second point deals with intraquery parallelism of join queries, i.e., executing each join query in parallel, which is possible under certain circumstances. Consider, for example, the edges between the fragments (i.e., the join graph) of EMP and PAY derived in Example 2.9. We have $\mathrm{PAY}_1 \rightarrow \mathrm{EMP}_1$ and $\mathrm{PAY}_2 \rightarrow \mathrm{EMP}_2$; there is only one edge coming in or going out of a fragment, so this is a *simple* join graph. The advantage of a design where the join relationship between fragments is simple is that the target and source of an edge can be allocated to one site and the joins between different pairs of fragments can proceed independently and in parallel.

Unfortunately, obtaining simple join graphs is not always possible. In that case, the next desirable alternative is to have a design that results in a *partitioned* join graph. A partitioned graph consists of two or more subgraphs with no edges between them. Fragments so obtained may not be distributed for parallel execution as easily as those obtained via simple join graphs, but the allocation is still possible.

Example 2.12 Let us continue with the distribution design of the database we started in Example 2.10. We already decided on the fragmentation of relation EMP according to the fragmentation of PAY (Example 2.11). Let us now consider ASG. Assume that there are the following two queries:

1. The first query finds the names of engineers who work at certain places. It runs on all three sites and accesses the information about the engineers who work on local projects with higher probability than those of projects at other locations.
2. At each administrative site where employee records are managed, users would like to access the responsibilities on the projects that these employees work on and learn how long they will work on those projects.

The first query results in a fragmentation of ASG according to the (nonempty) fragments PROJ_1, PROJ_3, PROJ_4, and PROJ_6 of PROJ obtained in Example 2.10:

$$\text{PROJ}_1 : \sigma_{\text{LOC}=\text{“Montreal”} \wedge \text{BUDGET} \leq 200000}(\text{PROJ})$$

$$\text{PROJ}_3 : \sigma_{\text{LOC}=\text{“New York”} \wedge \text{BUDGET} \leq 200000}(\text{PROJ})$$

$$\text{PROJ}_4 : \sigma_{\text{LOC}=\text{“New York”} \wedge \text{BUDGET} > 200000}(\text{PROJ})$$

$$\text{PROJ}_6 : \sigma_{\text{LOC}=\text{“Paris”} \wedge \text{BUDGET} > 200000}(\text{PROJ})$$

Therefore, the derived fragmentation of ASG according to $\{\text{PROJ}_1, \text{PROJ}_3, \text{PROJ}_4, \text{PROJ}_6\}$ is defined as follows:

$$\text{ASG}_1 = \text{ASG} \ltimes \text{PROJ}_1$$

$$\text{ASG}_2 = \text{ASG} \ltimes \text{PROJ}_3$$

$$\text{ASG}_3 = \text{ASG} \ltimes \text{PROJ}_4$$

$$\text{ASG}_4 = \text{ASG} \ltimes \text{PROJ}_6$$

These fragment instances are shown in Fig. 2.10.

The second query can be specified in SQL as

```
SELECT RESP, DUR
FROM   ASG NATURAL JOIN EMP_i
```

where $i = 1$ or $i = 2$, depending on the site where the query is issued. The derived fragmentation of ASG according to the fragmentation of EMP is defined below and depicted in Fig. 2.11.

$$\text{ASG}_1 = \text{ASG} \ltimes \text{EMP}_1$$

$$\text{ASG}_2 = \text{ASG} \ltimes \text{EMP1}_2$$

ASG_1

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24

ASG_2

ENO	PNO	RESP	DUR
E2	P2	Analyst	6
E4	P2	Programmer	18
E5	P2	Manager	24

ASG_3

ENO	PNO	RESP	DUR
E3	P3	Consultant	10
E7	P3	Engineer	36
E8	P3	Manager	40

ASG_4

ENO	PNO	RESP	DUR
E3	P4	Engineer	48
E6	P4	Manager	48

Fig. 2.10 Derived fragmentation of ASG with respect to PROJ

ASG$_1$

ENO	PNO	RESP	DUR
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E7	P3	Engineer	36

ASG$_2$

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E5	P2	Manager	24
E6	P4	Manager	48
E8	P3	Manager	40

Fig. 2.11 Derived fragmentation of ASG with respect to EMP

This example highlights two observations:

1. Derived fragmentation may follow a chain where one relation is fragmented as a result of another one's design and it, in turn, causes the fragmentation of another relation (e.g., the chain PAY $\rightarrow$ EMP $\rightarrow$ ASG).
2. Typically, there will be more than one candidate fragmentation for a relation (e.g., relation ASG). The final choice of the fragmentation scheme is a decision problem that may be addressed during allocation.

2.1.1.4 Checking for Correctness

We now check the fragmentation algorithms discussed so far with respect to the three correctness criteria we discussed earlier.

Completeness

The completeness of a primary horizontal fragmentation is based on the selection predicates used. As long as the selection predicates are complete, the resulting fragmentation is guaranteed to be complete as well. Since the basis of the fragmentation algorithm is a set of *complete* and *minimal* predicates (Pr'), completeness is guaranteed if Pr' is properly determined.

The completeness of a derived horizontal fragmentation is somewhat more difficult to define since the predicate determining the fragmentation involves two relations.

Let R be the target relation of an edge whose source is relation S, where R and S are fragmented as $F_R = \{R_1, R_2, \ldots, R_w\}$ and $F_S = \{S_1, S_2, \ldots, S_w\}$, respectively. Let A be the join attribute between R and S. Then for each tuple t of R_i, there should be a tuple t' of S_i such that $t[A] = t'[A]$. This is the well-known *referential integrity* rule, which ensures that the tuples of any fragment of the target relation are also in the source relation. For example, there should be no ASG tuple which has a project number that is not also contained in PROJ. Similarly, there should be no EMP tuples with TITLE values where the same TITLE value does not appear in PAY as well.

Reconstruction

Reconstruction of a global relation from its fragments is performed by the union operator in both the primary and the derived horizontal fragmentation. Thus, for a relation R with fragmentation $F_R = \{R_1, R_2, \ldots, R_w\}$, $R = \bigcup R_i, \quad \forall R_i \in F_R$.

Disjointness

It is easier to establish disjointness of fragmentation for primary than for derived horizontal fragmentation. In the former case, disjointness is guaranteed as long as the minterm predicates determining the fragmentation are mutually exclusive.

In derived fragmentation, however, there is a semijoin involved that adds considerable complexity. Disjointness can be guaranteed if the join graph is simple. Otherwise, it is necessary to investigate actual tuple values. In general, we do not want a tuple of a target relation to join with two or more tuples of the source relation when these tuples are in different fragments of the source. This may not be very easy to establish, and illustrates why derived fragmentation schemes that generate a simple join graph are always desirable.

Example 2.13 In fragmenting relation PAY (Example 2.10), the minterm predicates $M = \{m_1, m_2\}$ were

$$m_1 : \text{SAL} \leq 30000$$

$$m_2 : \text{SAL} > 30000$$

Since m_1 and m_2 are mutually exclusive, the fragmentation of PAY is disjoint.

For relation EMP, however, we require that

1. Each engineer has a single title.
2. Each title has a single salary value associated with it.

Since these two rules follow from the semantics of the database, the fragmentation of EMP with respect to PAY is also disjoint. ♦

2.1.2 Vertical Fragmentation

Recall that a vertical fragmentation of a relation R produces fragments $R_1, R_2, \ldots, R_r$, each of which contains a subset of R's attributes as well as the primary key of RÅs in the case of horizontal fragmentation, the objective is to partition a relation into a set of smaller relations so that many of the user applications will run on only one fragment. Primary key is included in each fragment to enable reconstruction, as we discuss later. This is also beneficial for integrity enforcement since the primary

key functionally determines all the relation attributes; having it in each fragment eliminates distributed computation to enforce primary key constraint.

Vertical partitioning is inherently more complicated than horizontal partitioning, mainly due to the total number of possible alternatives. For example, in horizontal partitioning, if the total number of simple predicates in Pr is n, there are 2^n possible minterm predicates. In addition, we know that some of these will contradict the existing implications, further reducing the candidate fragments that need to be considered. In the case of vertical partitioning, however, if a relation has m nonprimary key attributes, the number of possible fragments is equal to $B(m)$, which is the mth Bell number. For large values of m, $B(m) \approx m^m$; for example, for $m = 10$, $B(m) \approx 115,000$, for $m = 15$, $B(m) \approx 10^9$, for $m = 30$, $B(m) = 10^{23}$.

These values indicate that it is futile to attempt to obtain optimal solutions to the vertical partitioning problem; one has to resort to heuristics. Two types of heuristic approaches exist for the vertical fragmentation of global relations[5]:

1. *Grouping:* starts by assigning each attribute to one fragment, and at each step, joins some of the fragments until some criteria are satisfied.
2. *Splitting:* starts with a relation and decides on beneficial partitionings based on the access behavior of applications to the attributes.

In what follows we discuss only the splitting technique, since it fits more naturally within the design methodology we discussed earlier, since the "optimal" solution is probably closer to the full relation than to a set of fragments each of which consists of a single attribute. Furthermore, splitting generates nonoverlapping fragments, whereas grouping typically results in overlapping fragments. We prefer nonoverlapping fragments for disjointness. Of course, nonoverlapping refers only to nonprimary key attributes.

2.1.2.1 Auxiliary Information Requirements

We again require workload information. Since vertical partitioning places in one fragment those attributes usually accessed together, there is a need for some measure that would define more precisely the notion of "togetherness." This measure is the *affinity* of attributes, which indicates how closely related the attributes are. It is not realistic to expect the designer or the users to be able to easily specify these values. We present one way they can be obtained from more primitive data.

Let $Q = \{q_1, q_2, \ldots, q_q\}$ be the set of user queries that access relation $\mathtt{R}(\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n)$. Then, for each query q_i and each attribute $\mathtt{A}_j$, we associate an *attribute usage value*, denoted as $use(q_i, \mathtt{A}_j)$:

[5]There is also a third, extreme approach in column-oriented DBMS (like MonetDB and Vertica) where each column is mapped to one fragment. Since we do not cover column-oriented DBMSs in this book, we do not discuss this approach further.

$$use(q_i, \mathtt{A}_j) = \begin{cases} 1 \text{ if attribute } \mathtt{A}_j \text{ is referenced by query } q_i \\ 0 \text{ otherwise} \end{cases}$$

The $use(q_i, \bullet)$ vectors for each query are easy to determine.

Example 2.14 Consider relation PROJ of Fig. 2.2. Assume that the following queries are defined to run on this relation. In each case, we also give the SQL expression.

q_1: Find the budget of a project, given its identification number.

```
SELECT BUDGET
FROM   PROJ
WHERE  PNO=Value
```

q_2: Find the names and budgets of all projects.

```
SELECT PNAME, BUDGET
FROM   PROJ
```

q_3: Find the names of projects located at a given city.

```
SELECT PNAME
FROM   PROJ
WHERE  LOC=Value
```

q_4: Find the total project budgets for each city.

```
SELECT SUM(BUDGET)
FROM   PROJ
WHERE  LOC=Value
```

According to these four queries, the attribute usage values can be defined in matrix form (Fig. 2.12), where entry (i, j) denotes $use(q_i, \mathtt{A}_j)$. ♦

Attribute usage values are not sufficiently general to form the basis of attribute splitting and fragmentation, because they do not represent the weight of application frequencies. The frequency measure can be included in the definition of the attribute affinity measure $aff(\mathtt{A}_i, \mathtt{A}_j)$, which measures the bond between two attributes of a relation according to how they are accessed by queries.

	PNO	PNAME	BUDGET	LOC
q_1	0	1	1	0
q_2	1	1	1	0
q_3	1	0	0	1
q_4	0	0	1	0

Fig. 2.12 Example attribute usage matrix

The attribute affinity measure between two attributes $\mathtt{A}_i$ and $\mathtt{A}_j$ of a relation $\mathtt{R}(\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n)$ with respect to the set of queries $Q = \{q_1, q_2, \ldots, q_q\}$ is defined as

$$aff(\mathtt{A}_i, \mathtt{A}_j) = \sum_{k|use(q_k,\mathtt{A}_i)=1 \wedge use(q_k,\mathtt{A}_j)=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k)$$

where $ref_l(q_k)$ is the number of accesses to attributes $(\mathtt{A}_i, \mathtt{A}_j)$ for each execution of application q_k at site S_l and $acc_l(q_k)$ is the application access frequency measure previously defined and modified to include frequencies at different sites.

The result of this computation is an $n \times n$ matrix, each element of which is one of the measures defined above. This matrix is called the *attribute affinity matrix* (AA).

Example 2.15 Let us continue with the case that we examined in Example 2.14. For simplicity, let us assume that $ref_l(q_k) = 1$ for all q_k and S_l. If the application frequencies are

$$\begin{array}{ll}
acc_1(q_1) = 15 & acc_1(q_2) = 5 \\
acc_1(q_3) = 25 & acc_1(q_4) = 3 \\
acc_2(q_1) = 20 & acc_2(q_2) = 0 \\
acc_2(q_3) = 25 & acc_3(q_4) = 0 \\
acc_3(q_1) = 10 & acc_3(q_2) = 0 \\
acc_3(q_3) = 25 & acc_2(q_4) = 0
\end{array}$$

then the affinity measure between attributes PNO and BUDGET can be measured as

$$aff(\mathtt{PNO}, \mathtt{BUDGET}) = \sum_{k=1}^{1} \sum_{l=1}^{3} acc_l(q_k) = acc_1(q_1) + acc_2(q_1) + acc_3(q_1) = 45$$

since the only application that accesses both of the attributes is q_1. The complete attribute affinity matrix is shown in Fig. 2.13. Note that the diagonal values are not computed since they are meaningless. ♦

	PNO	PNAME	BUDGET	LOC
PNO	–	0	45	0
PNAME	0	–	5	75
BUDGET	45	5	–	3
LOC	0	75	3	–

Fig. 2.13 Attribute affinity matrix

The attribute affinity matrix will be used in the rest of this chapter to guide the fragmentation effort. The process first clusters together the attributes with high affinity for each other, and then splits the relation accordingly.

2.1.2.2 Clustering Algorithm

The fundamental task in designing a vertical fragmentation algorithm is to find some means of grouping the attributes of a relation based on the attribute affinity values in AA. We will discuss the bond energy algorithm (BEA) that has been proposed for this purpose. Other clustering algorithms can also be used.

BEA takes as input the attribute affinity matrix for relation $\mathtt{R}(\mathtt{A}_1, \ldots, \mathtt{A}_n)$, permutes its rows and columns, and generates a *clustered affinity matrix* (CA). The permutation is done in such a way as to *maximize* the following *global affinity measure* (AM):

$$AM = \sum_{i=1}^{n} \sum_{j=1}^{n} aff(\mathtt{A}_i, \mathtt{A}_j)[aff(\mathtt{A}_i, \mathtt{A}_{j-1}) + aff(\mathtt{A}_i, \mathtt{A}_{j+1}) + aff(\mathtt{A}_{i-1}, \mathtt{A}_j) + aff(\mathtt{A}_{i+1}, \mathtt{A}_j)]$$

where

$$aff(\mathtt{A}_0, \mathtt{A}_j) = aff(\mathtt{A}_i, \mathtt{A}_0) = aff(\mathtt{A}_{n+1}, \mathtt{A}_j) = aff(\mathtt{A}_i, \mathtt{A}_{n+1}) = 0$$

The last set of conditions takes care of the cases where an attribute is being placed in CA to the left of the leftmost attribute or to the right of the rightmost attribute during column permutations, and prior to the topmost row and following the last row during row permutations. We denote with $\mathtt{A}_0$ the attribute to the left of the leftmost attribute and the row prior to the topmost row, and with $\mathtt{A}_{n+1}$ the attribute to the right of the rightmost attribute or the row following the last row. In these cases, we set to 0 *aff* values between the attribute being considered for placement and its left or right (top or bottom) neighbors, since they do not exist in CA.

The maximization function considers the nearest neighbors only, thereby resulting in the grouping of large values with large ones, and small values with small ones. Also, the attribute affinity matrix (AA) is symmetric, which reduces the objective function to

$$AM = \sum_{i=1}^{n} \sum_{j=1}^{n} aff(\mathtt{A}_i, \mathtt{A}_j)[aff(\mathtt{A}_i, \mathtt{A}_{j-1}) + aff(\mathtt{A}_i, \mathtt{A}_{j+1})]$$

Algorithm 2.3: BEA

Input: AA: attribute affinity matrix
Output: CA: clustered affinity matrix
begin
  {initialize; remember that AA is an $n \times n$ matrix}
  $CA(\bullet, 1) \leftarrow AA(\bullet, 1)$
  $CA(\bullet, 2) \leftarrow AA(\bullet, 2)$
  $index \leftarrow 3$
  while $index \leq n$ **do** {choose the "best" location for attribute AA_{index}}
    for *i from 1 to* $index - 1$ *by 1* **do** calculate $cont(\mathtt{A}_{i-1}, \mathtt{A}_{index}, \mathtt{A}_i)$
    calculate $cont(\mathtt{A}_{index-1}, \mathtt{A}_{index}, \mathtt{A}_{index+1})$ {boundary condition}
    $loc \leftarrow$ placement given by maximum $cont$ value
    for *j from index to loc by* -1 **do**
      $CA(\bullet, j) \leftarrow CA(\bullet, j-1)$ {shuffle the two matrices}
    end for
    $CA(\bullet, loc) \leftarrow AA(\bullet, index)$
    $index \leftarrow index + 1$
  end while
  order the rows according to the relative ordering of columns
end

The details of BEA are given in Algorithm 2.3. Generation of the clustered affinity matrix (CA) is done in three steps:

1. *Initialization.* Place and fix one of the columns of AA arbitrarily into CA. Column 1 was chosen in the algorithm.
2. *Iteration.* Pick each of the remaining $n - i$ columns (where i is the number of columns already placed in CA) and try to place them in the remaining $i + 1$ positions in the CA matrix. Choose the placement that makes the greatest contribution to the global affinity measure described above. Continue this step until no more columns remain to be placed.
3. *Row ordering.* Once the column ordering is determined, the placement of the rows should also be changed so that their relative positions match the relative positions of the columns.[6]

For the second step of the algorithm to work, we need to define what is meant by the contribution of an attribute to the affinity measure. This contribution can be derived as follows. Recall that the global affinity measure AM was previously defined as

[6]From now on, we may refer to elements of the AA and CA matrices as $AA(i, j)$ and $CA(i, j)$, respectively. The mapping to the affinity measures is $AA(i, j) = aff(\mathtt{A}_i, \mathtt{A}_j)$ and $CA(i, j) = aff$(attribute placed at column i in CA, attribute placed at column j in CA). Even though AA and CA matrices are identical except for the ordering of attributes, since the algorithm orders all the CA columns before it orders the rows, the affinity measure of CA is specified with respect to columns. Note that the endpoint condition for the calculation of the affinity measure (AM) can be specified, using this notation, as $CA(0, j) = CA(i, 0) = CA(n+1, j) = CA(i, n+1) = 0$.

$$AM = \sum_{i=1}^{n} \sum_{j=1}^{n} aff(\mathtt{A}_i, \mathtt{A}_j)[aff(\mathtt{A}_i, \mathtt{A}_{j-1}) + aff(\mathtt{A}_i, \mathtt{A}_{j+1})]$$

which can be rewritten as

$$\begin{aligned} AM &= \sum_{i=1}^{n} \sum_{j=1}^{n} [aff(\mathtt{A}_i, \mathtt{A}_j) aff(\mathtt{A}_i, \mathtt{A}_{j-1}) + aff(\mathtt{A}_i, \mathtt{A}_j) aff(\mathtt{A}_i, \mathtt{A}_{j+1})] \\ &= \sum_{j=1}^{n} \left[\sum_{i=1}^{n} aff(\mathtt{A}_i, \mathtt{A}_j) aff(\mathtt{A}_i, \mathtt{A}_{j-1}) + \sum_{i=1}^{n} aff(\mathtt{A}_i, \mathtt{A}_j) aff(\mathtt{A}_i, \mathtt{A}_{j+1}) \right] \end{aligned}$$

Let us define the *bond* between two attributes $\mathtt{A}_x$ and $\mathtt{A}_y$ as

$$bond(\mathtt{A}_x, \mathtt{A}_y) = \sum_{z=1}^{n} aff(\mathtt{A}_z, \mathtt{A}_x) aff(\mathtt{A}_z, \mathtt{A}_y)$$

Then AM can be written as

$$AM = \sum_{j=1}^{n} [bond(\mathtt{A}_j, \mathtt{A}_{j-1}) + bond(\mathtt{A}_j, \mathtt{A}_{j+1})]$$

Now consider the following n attributes:

$$\underbrace{\mathtt{A}_1\ \mathtt{A}_2\ \ldots\ \mathtt{A}_{i-1}}_{AM'}\ \mathtt{A}_i\ \mathtt{A}_j\ \underbrace{\mathtt{A}_{j+1}\ \ldots\ \mathtt{A}_n}_{AM^1}$$

The global affinity measure for these attributes can be written as

$$\begin{aligned} AM_{old} &= AM' + AM^1 \\ &\quad + bond(\mathtt{A}_{i-1}, \mathtt{A}_i) + bond(\mathtt{A}_i, \mathtt{A}_j) + bond(\mathtt{A}_j, \mathtt{A}_i) + bond(\mathtt{A}_j, \mathtt{A}_{j+1}) \\ &= \sum_{l=1}^{i} [bond(\mathtt{A}_l, \mathtt{A}_{l-1}) + bond(\mathtt{A}_l, \mathtt{A}_{l+1})] \\ &\quad + \sum_{l=i+2}^{n} [bond(\mathtt{A}_l, \mathtt{A}_{l-1}) + bond(\mathtt{A}_l, \mathtt{A}_{l+1})] \\ &\quad + 2bond(\mathtt{A}_i, \mathtt{A}_j) \end{aligned}$$

Now consider placing a new attribute $\mathtt{A}_k$ between attributes $\mathtt{A}_i$ and $\mathtt{A}_j$ in the clustered affinity matrix. The new global affinity measure can be similarly written as

$$AM_{new} = AM' + AM^1 + bond(\mathtt{A}_i, \mathtt{A}_k) + bond(\mathtt{A}_k, \mathtt{A}_i)$$
$$+ bond(\mathtt{A}_k, \mathtt{A}_j) + bond(\mathtt{A}_j, \mathtt{A}_k)$$
$$= AM' + AM^1 + 2bond(\mathtt{A}_i, \mathtt{A}_k) + 2bond(\mathtt{A}_k, \mathtt{A}_j)$$

Thus, the net *contribution* to the global affinity measure of placing attribute $\mathtt{A}_k$ between $\mathtt{A}_i$ and $\mathtt{A}_j$ is

$$cont(\mathtt{A}_i, \mathtt{A}_k, \mathtt{A}_j) = AM_{new} - AM_{old}$$
$$= 2bond(\mathtt{A}_i, \mathtt{A}_k) + 2bond(\mathtt{A}_k, \mathtt{A}_j) - 2bond(\mathtt{A}_i, \mathtt{A}_j)$$

Example 2.16 Let us consider the AA matrix given in Fig. 2.13 and study the contribution of moving attribute LOC between attributes PNO and PNAME, given by the formula

$$cont(\mathtt{PNO}, \mathtt{LOC}, \mathtt{PNAME}) = 2bond(\mathtt{PNO}, \mathtt{LOC}) + 2bond(\mathtt{LOC}, \mathtt{PNAME})$$
$$- 2bond(\mathtt{PNO}, \mathtt{PNAME})$$

Computing each term, we get

$$bond(\mathtt{PNO}, \mathtt{LOC}) = 45 * 0 + 0 * 75 + 45 * 3 + 0 * 78 = 135$$
$$bond(\mathtt{LOC}, \mathtt{PNAME}) = 11865$$
$$bond(\mathtt{PNO}, \mathtt{PNAME}) = 225$$

Therefore,

$$cont(\mathtt{PNO}, \mathtt{LOC}, \mathtt{PNAME}) = 2 * 135 + 2 * 11865 - 2 * 225 = 23550$$

The algorithm and our discussion so far have both concentrated on the columns of the attribute affinity matrix. It is possible to redesign the algorithm to operate on the rows. Since the AA matrix is symmetric, both of these approaches will generate the same result.

Note that Algorithm 2.3 places the second column next to the first one during the initialization step. This obviously works since the bond between the two, however, is independent of their positions relative to one another.

Computing *cont* at the endpoints requires care. If an attribute $\mathtt{A}_i$ is being considered for placement to the left of the leftmost attribute, one of the bond equations to be calculated is between a nonexistent left element and $\mathtt{A}_k$ [i.e., $bond(\mathtt{A}_0, \mathtt{A}_k)$]. Thus we need to refer to the conditions imposed on the definition

of the global affinity measure AM, where $CA(0,k) = 0$. Similar arguments hold for the placement to the right of the rightmost attribute.

Example 2.17 We consider the clustering of the PROJ relation attributes and use the attribute affinity matrix AA of Fig. 2.13.

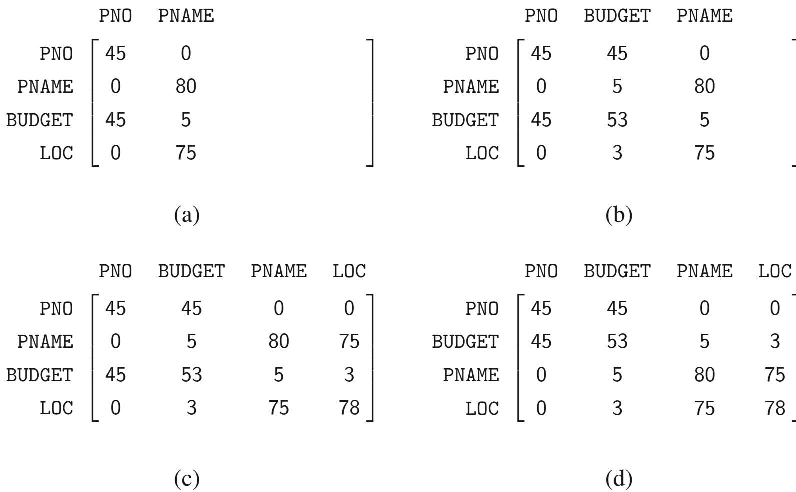
According to the initialization step, we copy columns 1 and 2 of the AA matrix to the CA matrix (Fig. 2.14a) and start with column 3 (i.e., attribute BUDGET). There are three alternative places where column 3 can be placed: to the left of column 1, resulting in the ordering (3-1-2), in between columns 1 and 2, giving (1-3-2), and to the right of 2, resulting in (1-2-3). Note that to compute the contribution of the last ordering we have to compute *cont*(PNAME, BUDGET, LOC) rather than *cont*(PNO, PNAME, BUDGET). However, note that attribute LOC has not yet been placed into the CA matrix (Fig. 2.14b), thus requiring special computation as outlined above. Let us calculate the contribution to the global affinity measure of each alternative.

Ordering (0-3-1):

$$cont(\mathtt{A_0}, \mathtt{BUDGET}, \mathtt{PNO}) = 2bond(\mathtt{A_0}, \mathtt{BUDGET}) + 2bond(\mathtt{BUDGET}, \mathtt{PNO}) - 2bond(\mathtt{A_0}, \mathtt{PNO})$$

We know that

$$\begin{aligned} bond(\mathtt{A_0}, \mathtt{PNO}) &= bond(\mathtt{A_0}, \mathtt{BUDGET}) = 0 \\ bond(\mathtt{BUDGET}, \mathtt{PNO}) &= 45 * 45 + 5 * 0 + 53 * 45 + 3 * 0 = 4410 \end{aligned}$$



(a)

	PNO	PNAME
PNO	45	0
PNAME	0	80
BUDGET	45	5
LOC	0	75

(b)

	PNO	BUDGET	PNAME
PNO	45	45	0
PNAME	0	5	80
BUDGET	45	53	5
LOC	0	3	75

(c)

	PNO	BUDGET	PNAME	LOC
PNO	45	45	0	0
PNAME	0	5	80	75
BUDGET	45	53	5	3
LOC	0	3	75	78

(d)

	PNO	BUDGET	PNAME	LOC
PNO	45	45	0	0
BUDGET	45	53	5	3
PNAME	0	5	80	75
LOC	0	3	75	78

Fig. 2.14 Calculation of the clustered affinity (CA) matrix

Thus

$$cont(\mathtt{A_0}, \mathtt{BUDGET}, \mathtt{PNO}) = 8820$$

Ordering (1-3-2):

$$\begin{aligned}
cont(\mathtt{PNO}, \mathtt{BUDGET}, \mathtt{PNAME}) = & \ 2bond\,\mathtt{PNO}, \mathtt{BUDGET}) + 2bond(\mathtt{BUDGET}, \mathtt{PNAME}) \\
& - 2bond(\mathtt{PNO}, \mathtt{PNAME}) \\
bond(\mathtt{PNO}, \mathtt{BUDGET}) &= bond(\mathtt{BUDGET}, \mathtt{PNO}) = 4410 \\
bond(\mathtt{BUDGET}, \mathtt{PNAME}) &= 890 \\
bond(\mathtt{PNO}, \mathtt{PNAME}) &= 225
\end{aligned}$$

Thus

$$cont(\mathtt{PNO}, \mathtt{BUDGET}, \mathtt{PNAME}) = 10150$$

Ordering (2-3-4):

$$\begin{aligned}
cont(\mathtt{PNAME}, \mathtt{BUDGET}, \mathtt{LOC}) = & \ 2bond(\mathtt{PNAME}, \mathtt{BUDGET}) + 2bond(\mathtt{BUDGET}, \mathtt{LOC}) \\
& - 2bond(\mathtt{PNAME}, \mathtt{LOC}) \\
bond(\mathtt{PNAME}, \mathtt{BUDGET}) &= 890 \\
bond(\mathtt{BUDGET}, \mathtt{LOC}) &= 0 \\
bond(\mathtt{PNAME}, \mathtt{LOC}) &= 0
\end{aligned}$$

Thus

$$cont(\mathtt{PNAME}, \mathtt{BUDGET}, \mathtt{LOC}) = 1780$$

Since the contribution of the ordering (1-3-2) is the largest, we select to place BUDGET to the right of PNO (Fig. 2.14b). Similar calculations for LOC indicate that it should be placed to the right of PNAME (Fig. 2.14c).

Finally, the rows are organized in the same order as the columns and the result is shown in Fig. 2.14d. ♦

In Fig. 2.14d we see the creation of two clusters: one is in the upper left corner and contains the smaller affinity values and the other is in the lower right corner and contains the larger affinity values. This clustering indicates how the attributes of relation PROJ should be split. However, in general the border for this split may not be this clear-cut. When the CA matrix is big, usually more than two clusters are formed and there are more than one candidate partitionings. Thus, there is a need to approach this problem more systematically.

2.1.2.3 Splitting Algorithm

The objective of splitting is to find sets of attributes that are accessed solely, or for the most part, by distinct sets of queries. For example, if it is possible to identify two attributes A_1 and A_2 that are accessed only by query q_1, and attributes A_3 and A_4 that are accessed by, say, two queries q_2 and q_3, it would be quite straightforward to decide on the fragments. The task lies in finding an algorithmic method of identifying these groups.

Consider the clustered attribute matrix of Fig. 2.15. If a point along the diagonal is fixed, two sets of attributes are identified. One set $\{A_1, A_2, \ldots, A_i\}$ is at the upper left-hand corner (denoted TA) and the second set $\{A_{i+1}, \ldots, A_n\}$ is at the lower right corner (denoted TB) relative to this point.

We now partition the set of queries $Q = \{q_1, q_2, \ldots, q_q\}$ that access only TA, only BA, or both. These sets are defined as follows:

$$AQ(q_i) = \{A_j | use(q_i, A_j) = 1\}$$

$$TQ = \{q_i | AQ(q_i) \subseteq TA\}$$

$$BQ = \{q_i | AQ(q_i) \subseteq BA\}$$

$$OQ = Q - \{TQ \cup BQ\}$$

The first of these equations defines the set of attributes accessed by query q_i; TQ and BQ are the sets of queries that only access TA or BA, respectively, and OQ is the set of queries that access both.

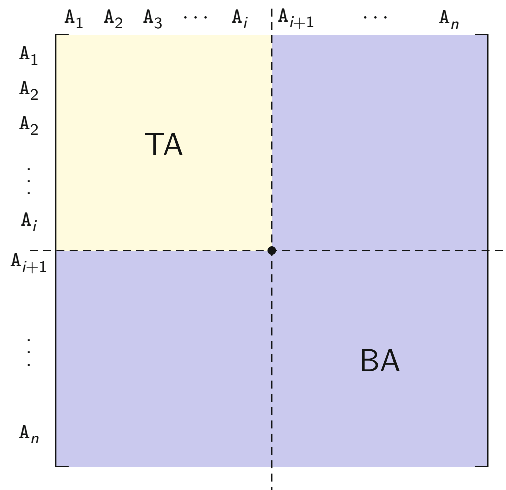


Fig. 2.15 Locating a splitting point

There is an optimization problem here. If there are n attributes of a relation, there are $n-1$ possible positions where the dividing point can be placed along the diagonal of the clustered attribute matrix for that relation. The best position for division is one which produces the sets TQ and BQ such that the total accesses to *only one* fragment are maximized, while the total accesses to *both* fragments are minimized. We therefore define the following cost equations:

$$CQ = \sum_{q_i \in Q} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$CTQ = \sum_{q_i \in TQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$CBQ = \sum_{q_i \in BQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$COQ = \sum_{q_i \in OQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

Each of the equations above counts the total number of accesses to attributes by queries in their respective classes. Based on these measures, the optimization problem is defined as finding the point x $(1 \leq x \leq n)$ such that the expression

$$z = CTQ * CBQ - COQ^2$$

is maximized. The important feature of this expression is that it defines two fragments such that the values of CTQ and CBQ are as nearly equal as possible. This enables the balancing of processing loads when the fragments are distributed to various sites. It is clear that the partitioning algorithm has linear complexity in terms of the number of attributes of the relation, that is, $O(n)$.

This procedure splits the set of attributes two-way. For larger sets of attributes, it is quite likely that m-way partitioning may be necessary. Designing an m-way partitioning is possible but computationally expensive. Along the diagonal of the CA matrix, it is necessary to try $1, 2, \ldots, m-1$ split points, and for each of these, it is necessary to check which point maximizes z. Thus, the complexity of such an algorithm is $O(2^m)$. Of course, the definition of z has to be modified for those cases where there are multiple split points. The alternative solution is to recursively apply the binary partitioning algorithm to each of the fragments obtained during the previous iteration. One would compute TQ, BQ, and OQ, as well as the associated access measures for each of the fragments, and partition them further.

Our discussion so far assumed that the split point is unique and single and divides the CA matrix into an upper left-hand partition and a second partition formed by the rest of the attributes. The partition, however, may also be formed in the middle of the matrix. In this case, we need to modify the algorithm slightly. The leftmost column of the CA matrix is shifted to become the rightmost column and the topmost row is

Algorithm 2.4: SPLIT

```
Input: CA: clustered affinity matrix; R: relation; ref: attribute usage matrix; acc: access
       frequency matrix
Output: F: set of fragments
begin
    {determine the z value for the first column}
    {the subscripts in the cost equations indicate the split point}
    calculate CTQ_{n-1}
    calculate CBQ_{n-1}
    calculate COQ_{n-1}
    best ← CTQ_{n-1} * CBQ_{n-1} − (COQ_{n-1})^2
    repeat
        {determine the best partitioning}
        for i from n − 2 to 1 by −1 do
            calculate CTQ_i
            calculate CBQ_i
            calculate COQ_i
            z ← CTQ * CBQ_i − COQ_i^2
            if z > best then best ← z          {record the split point within shift}
        end for
        call SHIFT(CA)
    until no more SHIFT is possible
    reconstruct the matrix according to the shift position
    R1 ← Π_TA(R) ∪ K                 {K is the set of primary key attributes of R}
    R2 ← Π_BA(R) ∪ K
    F ← {R1, R2}
end
```

shifted to the bottom. The shift operation is followed by checking the $n-1$ diagonal positions to find the maximum z. The idea behind shifting is to move the block of attributes that should form a cluster to the topmost left corner of the matrix, where it can easily be identified. With the addition of the shift operation, the complexity of the partitioning algorithm increases by a factor of n and becomes $O(n^2)$.

Assuming that a shift procedure, called SHIFT, has already been implemented, the splitting algorithm is given in Algorithm 2.4. The input of the algorithm is the clustered affinity matrix CA, the relation R to be fragmented, and the attribute usage and access frequency matrices. The output is a set of fragments $F_{\text{R}} = \{\text{R}_1, \text{R}_2\}$, where $\text{R}_i \subseteq \{\text{A}_1, \text{A}_2 \ldots, \text{A}_n\}$ and $\text{R}_1 \cap \text{R}_2 =$ the key attributes of relation RṄote that for n-way partitioning, this routine should be either invoked iteratively or implemented as a recursive procedure.

Example 2.18 When the SPLIT algorithm is applied to the CA matrix obtained for relation PROJ (Example 2.17), the result is the definition of fragments $F_{\text{PROJ}} = \{\text{PROJ}_1, \text{PROJ}_2\}$, where

$$\text{PROJ}_1 = \{\text{PNO, BUDGET}\}$$

$$\text{PROJ}_2 = \{\text{PNO, PNAME, LOC}\}$$

Note that in this exercise we performed the fragmentation over the entire set of attributes rather than only on the nonkey ones. The reason for this is the simplicity of the example. For that reason, we included PNO, which is the key of PROJ in $PROJ_2$ as well as in $PROJ_1$. ♦

2.1.2.4 Checking for Correctness

We follow arguments similar to those of horizontal partitioning to prove that the SPLIT algorithm yields a correct vertical fragmentation.

Completeness

Completeness is guaranteed by the SPLIT algorithm since each attribute of the global relation is assigned to one of the fragments. As long as the set of attributes A over which the relation R is defined consists of $A = \bigcup R_i$, completeness of vertical fragmentation is ensured.

Reconstruction

We have already mentioned that the reconstruction of the original global relation is made possible by the join operation. Thus, for a relation R with vertical fragmentation $F_R = \{R_1, R_2, \ldots, R_r\}$ and key attribute(s) K, $R = \bowtie_K R_i, \forall R_i \in F_R$. Therefore, as long as each R_i is complete, the join operation will properly reconstruct RȦnother important point is that either each R_i should contain the key attribute(s) of R or it should contain the system assigned tuple IDs (TIDs).

Disjointness

As noted earlier, the primary key attributes are replicated in each fragment. Excluding these, the SPLIT algorithm finds mutually exclusive clusters of attributes, leading to disjoint fragments with respect to the attributes.

2.1.3 Hybrid Fragmentation

In some cases a simple horizontal or vertical fragmentation of a database schema may not be sufficient to satisfy the requirements of user applications. In this case a vertical fragmentation may be followed by a horizontal one, or vice versa, producing a tree-structured partitioning (Fig. 2.16). Since the two types of

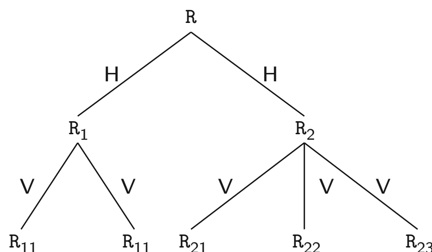


Fig. 2.16 Hybrid fragmentation

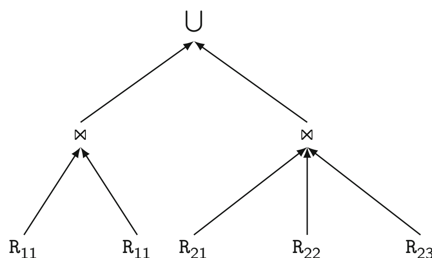


Fig. 2.17 Reconstruction of hybrid fragmentation

partitioning strategies are applied one after the other, this alternative is called *hybrid* fragmentation. It has also been named *mixed* fragmentation or *nested* fragmentation.

A good example for the necessity of hybrid fragmentation is relation PROJ. In Example 2.10 we partitioned it into six horizontal fragments based on two applications. In Example 2.18 we partitioned the same relation vertically into two. What we have, therefore, is a set of horizontal fragments, each of which is further partitioned into two vertical fragments.

The correctness rules and conditions for hybrid fragmentation follow naturally from those for vertical and horizontal fragmentations. For example, to reconstruct the original global relation in case of hybrid fragmentation, one starts at the leaves of the partitioning tree and moves upward by performing joins and unions (Fig. 2.17). The fragmentation is complete if the intermediate and leaf fragments are complete. Similarly, disjointness is guaranteed if intermediate and leaf fragments are disjoint.

2.2 Allocation

Following fragmentation, the next decision problem is to allocate fragments to the sites of the distributed DBMS. This can be done by either placing each fragment at a single site or replicating it on a number of sites. The reasons for replication are reliability and efficiency of read-only queries. If there are multiple copies of

a fragment, there is a good chance that some copy of the data will be accessible somewhere even when system failures occur. Furthermore, read-only queries that access the same data items can be executed in parallel since copies exist on multiple sites. On the other hand, the execution of update queries causes trouble since the system has to ensure that all the copies of the data are updated properly. Hence the decision regarding replication is a trade-off that depends on the ratio of the read-only queries to the update queries. This decision affects almost all of the distributed DBMS algorithms and control functions.

A nonreplicated database (commonly called a *partitioned* database) contains fragments that are allocated to sites such that each fragment is placed at one site. In case of replication, either the database exists in its entirety at each site (*fully replicated* database), or fragments are distributed to the sites in such a way that copies of a fragment may reside in multiple sites (*partially replicated* database). In the latter the number of copies of a fragment may be an input to the allocation algorithm or a decision variable whose value is determined by the algorithm. Figure 2.18 compares these three replication alternatives with respect to various distributed DBMS functions. We will discuss replication at length in Chap. 6.

The file allocation problem has long been studied within the context of distributed computing systems where the unit of allocation is a file. This is commonly referred as the *file allocation problem* (FAP) and the formulations are usually quite simple, reflecting the simplicity of file APIs. Even this simple version has been shown to be NP-complete, resulting in a search for reasonable heuristics.

FAP formulations are not suitable for distributed database design, due fundamentally to the characteristics of DBMSs: fragments are not independent of each other so they cannot simply be mapped to individual files; the access to data in a database is more complex than simple access to files; and DBMSs enforce integrity and transactional properties whose costs need to be considered.

There are no general heuristic models that take as input a set of fragments and produce a near-optimal allocation subject to the types of constraints discussed here. The models developed to date make a number of simplifying assumptions and are applicable to certain specific formulations. Therefore, instead of presenting one or

	Full replication	Partial replication	Partitioning
QUERY PROCESSING	Easy	Same difficulty	
DIRECTORY MANAGEMENT	Easy or nonexistent	Same difficulty	
CONCURRENCY CONTROL	Moderate	Difficult	Easy
RELIABILITY	Very high	High	Low
REALITY	Possible application	Realistic	Possible application

Fig. 2.18 Comparison of replication alternatives

more of these allocation algorithms, we present a relatively general model and then discuss a number of possible heuristics that might be employed to solve it.

2.2.1 Auxiliary Information

We need the quantitative data about the database, the workload, the communication network, the processing capabilities, and storage limitations of each site on the network.

To perform horizontal fragmentation, we defined the selectivity of minterms. We now need to extend that definition to fragments, and define the selectivity of a fragment F_j with respect to query q_i. This is the number of tuples of F_j that need to be accessed in order to process q_i. This value will be denoted as $sel_i(F_j)$.

Another piece of necessary information on the database fragments is their size. The size of a fragment F_j is given by

$$size(F_j) = card(F_j) * length(F_j)$$

where $length(F_j)$ is the length (in bytes) of a tuple of fragment F_j.

Most of the workload-related information is already compiled during fragmentation, but a few more are required by the allocation model. The two important measures are the number of read accesses that a query q_i makes to a fragment F_j during its execution (denoted as RR_{ij}), and its counterpart for the update accesses (UR_{ij}). These may, for example, count the number of block accesses required by the query.

We also need to define two matrices UM and RM, with elements u_{ij} and r_{ij}, respectively, which are specified as follows:

$$u_{ij} = \begin{cases} 1 & \text{if query } q_i \text{ updates fragment } F_j \\ 0 & \text{otherwise} \end{cases}$$

$$r_{ij} = \begin{cases} 1 & \text{if query } q_i \text{ retrieves from fragment } F_j \\ 0 & \text{otherwise} \end{cases}$$

A vector O of values $o(i)$ is also defined, where $o(i)$ specifies the originating site of query q_i. Finally, to define the response-time constraint, the maximum allowable response time of each application should be specified.

For each computer site, we need to know its storage and processing capacity. Obviously, these values can be computed by means of elaborate functions or by simple estimates. The unit cost of storing data at site S_k will be denoted as USC_k. There is also a need to specify a cost measure LPC_k as the cost of processing one unit of work at site S_k. The work unit should be identical to that of the RR and UR measures.

In our model we assume the existence of a simple network where the cost of communication is defined in terms of one message that contains a specific amount of data. Thus g_{ij} denotes the communication cost per message between sites S_i and S_j. To enable the calculation of the number of messages, we use $msize$ as the size (in bytes) of one message. There are more elaborate network models that take into consideration the channel capacities, distances between sites, protocol overhead, and so on, but this simple model is sufficient for our purposes.

2.2.2 Allocation Model

We discuss an allocation model that attempts to minimize the total cost of processing and storage while trying to meet certain response time restrictions. The model we use has the following form:

min(Total Cost)

subject to

response-time constraint
storage constraint
processing constraint

In the remainder of this section, we expand the components of this model based on the information requirements discussed in Sect. 2.2.1. The decision variable is x_{ij}, which is defined as

$$x_{ij} = \begin{cases} 1 \text{ if the fragment } F_i \text{ is stored at site } S_j \\ 0 \text{ otherwise} \end{cases}$$

2.2.2.1 Total Cost

The total cost function has two components: query processing and storage. Thus it can be expressed as

$$TOC = \sum_{\forall q_i \in Q} QPC_i + \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} STC_{jk}$$

where QPC_i is the query processing cost of query q_i, and STC_{jk} is the cost of storing fragment F_j at site S_k.

Let us consider the storage cost first. It is simply given by

$$STC_{jk} = USC_k * size(F_j) * x_{jk}$$

and the two summations find the total storage costs at all the sites for all the fragments.

The query processing cost is more difficult to specify. We specify it as consisting of the processing cost (PC) and the transmission cost (TC). Thus the query processing cost (QPC) for application q_i is

$$QPC_i = PC_i + TC_i$$

The processing component, PC, consists of three cost factors, the access cost (AC), the integrity enforcement cost (IE), and the concurrency control cost (CC):

$$PC_i = AC_i + IE_i + CC_i$$

The detailed specification of each of these cost factors depends on the algorithms used to accomplish these tasks. However, to demonstrate the point, we specify AC in some detail:

$$AC_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} (u_{ij} * UR_{ij} + r_{ij} * RR_{ij}) * x_{jk} * LPC_k$$

The first two terms in the above formula calculate the number of accesses of user query q_i to fragment F_j. Note that $(UR_{ij} + RR_{ij})$ gives the total number of update and retrieval accesses. We assume that the local costs of processing them are identical. The summation gives the total number of accesses for all the fragments referenced by q_i. Multiplication by LPC_k gives the cost of this access at site S_k. We again use x_{jk} to select only those cost values for the sites where fragments are stored.

The access cost function assumes that processing a query involves decomposing it into a set of subqueries, each of which works on a fragment stored at the site, followed by transmitting the results back to the site where the query has originated. Reality is more complex; for example, the cost function does not take into account the cost of performing joins (if necessary), which may be executed in a number of ways (see Chap. 4).

The integrity enforcement cost factor can be specified much like the processing component, except that the unit local processing cost would likely change to reflect the true cost of integrity enforcement. Since the integrity checking and concurrency control methods are discussed later in the book, we do not study these cost components further here. The reader should refer back to this section after reading Chaps. 3 and 5 to be convinced that the cost functions can indeed be derived.

The transmission cost function can be formulated along the lines of the access cost function. However, the data transmission overhead for update and that for retrieval requests may be quite different. In update queries it is necessary to inform all the sites where replicas exist, while in retrieval queries, it is sufficient to access only one of the copies. In addition, at the end of an update request, there is no data transmission back to the originating site other than a confirmation message, whereas the retrieval-only queries may result in significant data transmission.

The update component of the transmission function is

$$TCU_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{o(i),k} + \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{k,o(i)}$$

The first term is for sending the update message from the originating site $o(i)$ of q_i to all the fragment replicas that need to be updated. The second term is for the confirmation.

The retrieval cost can be specified as

$$TCR_i = \sum_{\forall F_j \in F} \min_{S_k \in S}(r_{ij} * x_{jk} * g_{o(i),k} + r_{ij} * x_{jk} * \frac{sel_i(F_j) * length(F_j)}{msize} * g_{k,o(i)})$$

The first term in TCR represents the cost of transmitting the retrieval request to those sites which have copies of fragments that need to be accessed. The second term accounts for the transmission of the results from these sites to the originating site. The equation states that among all the sites with copies of the same fragment, only the site that yields the minimum total transmission cost should be selected for the execution of the operation.

Now the transmission cost function for query q_i can be specified as

$$TC_i = TCU_i + TCR_i$$

which fully specifies the total cost function.

2.2.2.2 Constraints

The constraint functions can be specified in similar detail. However, instead of describing these functions in depth, we will simply indicate what they should look like. The response-time constraint should be specified as

$$\text{execution time of } q_i \leq \text{ maximum response time of } q_i, \forall q_i \in Q$$

Preferably, the cost measure in the objective function should be specified in terms of time, as it makes the specification of the execution time constraint relatively straightforward.

The storage constraint is

$$\sum_{\forall F_j \in F} STC_{jk} \leq \text{ storage capacity at site } S_k, \forall S_k \in S$$

whereas the processing constraint is

$$\sum_{\forall q_i \in Q} \text{processing load of } q_i \text{ at site } S_k \leq \text{ processing capacity of } S_k, \forall S_k \in S$$

This completes our development of the allocation model. Even though we have not developed it entirely, the precision in some of the terms indicates how one goes about formulating such a problem. In addition to this aspect, we have indicated the important issues that need to be addressed in allocation models.

2.2.3 *Solution Methods*

As noted earlier, simple file allocation problem is NP-complete. Since the model we developed in the previous section is more complex, it is likely to be NP-complete as well. Thus one has to look for heuristic methods that yield suboptimal solutions. The test of "goodness" in this case is, obviously, how close the results of the heuristic algorithm are to the optimal allocation.

It was observed early on that there is a correspondence between the file allocation and the facility location problems. In fact, the isomorphism of the simple file allocation problem and the single commodity warehouse location problem has been shown. Thus, heuristics developed for the latter have been used for the former. Examples are the knapsack problem solution, branch-and-bound techniques, and network flow algorithms.

There have been other attempts to reduce the complexity of the problem. One strategy has been to assume that all the candidate partitionings have been determined together with their associated costs and benefits in terms of query processing. The problem, then, is modeled as choosing the optimal partitioning and placement for each relation. Another simplification frequently employed is to ignore replication at first and find an optimal nonreplicated solution. Replication is handled at the second step by applying a greedy algorithm which starts with the nonreplicated solution as the initial feasible solution, and tries to improve upon it. For these heuristics, however, there is not enough data to determine how close the results are to the optimal.

2.3 Combined Approaches

The design process depicted in Fig. 2.1 on which we based our discussion separates the fragmentation and allocation steps. The methodology is linear where the output of fragmentation is input to allocation; we call this the *fragment-then-allocate approach*. This simplifies the formulation of the problem by reducing the decision space, but the isolation of the two steps may in fact contribute to the complexity of the allocation models. Both steps have similar inputs, differing only in that fragmentation works on global relations, whereas allocation considers fragments. They both require workload information, but ignore how each other makes use of these inputs. The end result is that the fragmentation algorithms decide how to partition a relation based partially on how queries access it, but the allocation models

ignore the part that this input plays in fragmentation. Therefore, the allocation models have to include all over again detailed specification of the relationship among the fragment relations and how user applications access them. There are approaches that combine the fragmentation and allocation steps in such a way that the data partitioning algorithm also dictates allocation, or the allocation algorithm dictates how the data is partitioned; we call these the *combined approaches*. These mostly consider horizontal partitioning, since that is the common method for obtaining significant parallelism. In this section we present these approaches, classified as either workload-agnostic or workload-aware.

2.3.1 Workload-Agnostic Partitioning Techniques

This class of techniques ignores the workload that will run on the data and simply focus on the database, often not even paying attention to the schema definition. These approaches are mostly used in parallel DBMSs where data dynamism is higher than distributed DBMSs, so simpler techniques that can be quickly applied are preferred.

The simplest form of these algorithms is *round-robin partitioning* (Fig. 2.19). With n partitions, the ith tuple in insertion order is assigned to partition (i mod n). This strategy enables the sequential access to a relation to be done in parallel. However, the direct access to individual tuples, based on a predicate, requires accessing the entire relation. Thus, round-robin partitioning is appropriate for full scan queries, as in data mining.

An alternative is *hash partitioning*, which applies a hash function to some attribute that yields the partition number (Fig. 2.20). This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all

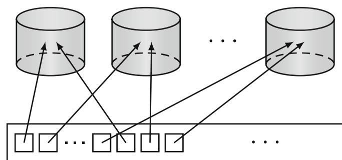


Fig. 2.19 Round-robin partitioning

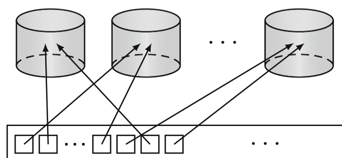


Fig. 2.20 Hash partitioning

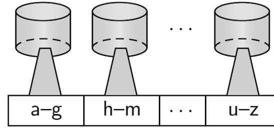


Fig. 2.21 Range partitioning

other queries to be processed by all the nodes in parallel. However, if the attribute used for partitioning has nonuniform data distribution, e.g., as with people's names, the resulting placement may be unbalanced, with some partitions much bigger than some others. This is called *data skew* and it is an important issue that can cause unbalanced load.

Finally, there is *range partitioning* (Fig. 2.21) that distributes tuples based on the value intervals (ranges) of some attribute and thus can deal with nonuniform data distributions. Unlike hashing, which relies on hash functions, ranges must be maintained in an index structure, e.g., a B-tree. In addition to supporting exact-match queries (as in hashing), it is well-suited for range queries. For instance, a query with a predicate "A between A_1 and A_2" may be processed by the only node(s) containing tuples whose A value is in range $[A_1, A_2]$.

These techniques are simple, can be computed quickly and, as we discuss in Chap. 8, nicely fit the dynamicity of data in parallel DBMSs. However, they have indirect ways of handling the semantic relationships among relations in the database. For example, consider two relations that have a foreign key–primary key join relationship such as $\texttt{R} \bowtie_{\texttt{R.A=S.B}} \texttt{S}$, hash partitioning would use the same function over attribute `R.A` and `S.B` to ensure that they are located at the same node, thereby localizing the joins and parallelizing the join execution. A similar approach can be used in range partitioning, but round-robin would not take this relationship into account.

2.3.2 Workload-Aware Partitioning Techniques

This class of techniques considers the workload as input and performs partitioning to localize as much of the workload on one site as possible. As noted at the beginning of this chapter, their objective is to minimize the amount of distributed queries.

One approach that has been proposed in a system called Schism uses the database and workload information to build a graph $G = V, E$ where each vertex v in V represents a tuple in the database, and each edge $e = (v_i, v_j)$ in E represents a query that accesses both tuples v_i and v_j. Each edge is assigned a weight that is the count of the number of transactions that access both tuples.

In this model, it is also easy to take into account replicas, by representing each copy by a separate vertex. The number of replica vertices is determined by the number of transactions accessing the tuple; i.e., each transaction accesses one

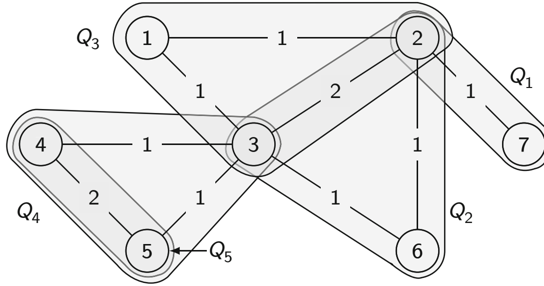


Fig. 2.22 Graph representation for partitioning in schism

copy. A replicated tuple is represented in the graph by a star-shaped configuration consisting of $n+1$ vertices where the "central" vertex represents the logical tuple and the other n vertices represent the physical copies. The weight of an edge between the physical copy vertex and the central vertex is the number of transactions that update the tuple; the weights of other edges remain as the number of queries that access the tuple. This arrangement makes sense since the objective is to localize transactions as much as possible and this technique uses replication to achieve localization.

Example 2.19 Let us consider a database with one relation consisting of seven tuples that are accessed by five transactions. In Fig. 2.22 we depict the graph that is constructed: there are seven vertices corresponding to the tuples, and the queries that access them together are shown as cliques. For example, query Q_1 accesses tuples 2 and 7, query Q_2 accesses tuples 2, 3, and 6, query Q_3 accesses tuples 1, 2, and 3, query Q_4 accesses tuples 3, 4, and 5, and query Q_5 accesses tuples 4 and 5. Edge weights capture the number of transaction accesses.

Replication can be incorporated into this graph but replicating the tuples that are accessed by multiple transactions; this is shown in Fig. 2.23. Note that tuples 1, 6, and 7 are not replicated since they are only accessed by one transaction each, tuples 4 and 5 are replicated twice, and tuples 2 and 3 are replicated three times. We represent the "replication edges" between the central vertex and each physical copy by dashed lines and omit the weights for these edges in this example. ♦

Once the database and the workload are captured by this graph representation, the next step is to perform a vertex-disjoint graph partitioning. Since we discuss these techniques in detail in Sect. 10.4.1, we do not get into the details here, but simply state that vertex-disjoint partitioning allocates each vertex of the graph to a separate partition such that partitions are mutually exclusive. These algorithms have, as their objective function, a balanced (or nearly balanced) set of partitions while minimizing the cost of edge cuts. The cost of an edge cut takes into account the weights of each edge so as to minimize the number of distributed queries.

The advantage of the Schism approach is its fine-grained allocation—it treats each tuple as an allocation unit and the partitioning "emerges" as the allocation decision is made for each tuple. Thus, the mapping of sets of tuples to queries can

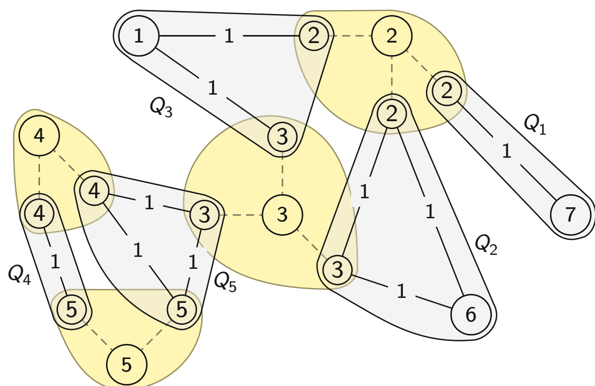


Fig. 2.23 Schism graph incorporating replication

be controlled and many of them can execute at one site. However, the downside of the approach is that the graph becomes very large as the database size increases, in particular when replicas are added to the graph. This makes the management of the graph difficult and its partitioning expensive. Another issue to consider is that the mapping tables that record where each tuple is stored (i.e., the directory) become very large and may pose a management problem of their own.

One approach to overcome these issues has been proposed as part of the SWORD system that employs a hypergraph model[7] where each clique in Fig. 2.22 is represented as a hyperedge. Each hyperedge represents one query and the set of vertices spanned by the hyperedge represents the tuples accessed by it. Each hyperedge has a weight that represents the frequency of that query in the workload. Therefore what we have is a weighted hypergraph. This hypergraph is then partitioned using a k-way balanced min-cut partitioning algorithm that produces k balanced partitions, each of which is allocated to a site. This minimizes the number of distributed queries since the algorithm is minimizing the cuts in hyperedges and each of these cuts indicates a distributed query.

Of course, this change in the model is not sufficient to address the issues discussed above. In order to reduce the size of the graph, and the overhead of maintaining the associated mapping table, SWORD compresses this hypergraph as follows. The set of vertices V in the original hypergraph G is mapped to a set of virtual vertices V' using a hash or other function that operates on the primary keys of the tuples. Once the set of virtual vertices are determined, the edges in the original hypergraph are now mapped to hyperedges in the compressed graph (E') such that if the vertices spanned by a hyperedge $e \in E$ are mapped to different virtual vertices in the compressed graph, then there will be a hyperedge $e' \in E'$.

[7] A hypergraph allows each edge (called a hyperedge) to connect more than two vertices as is the case with regular graphs. The details of the hypergraph model are beyond our scope.

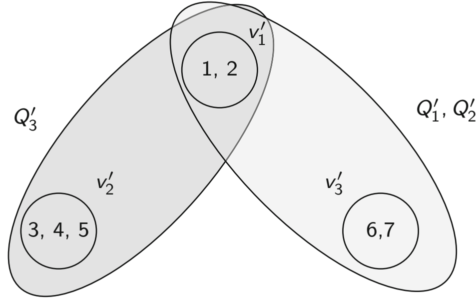


Fig. 2.24 Sword compressed hypergraph

Of course for this compression to make sense, $|V'| < |V|$, so a critical issue is to determine how much compression is desired—too much compression will reduce the number of virtual vertices, but will increase the number of hyperedges, and therefore the possibility of distributed queries. The resulting compressed hypergraph $G' = (V', E')$ is going to be smaller than the original hypergraph so easier to manage and partition, and the mapping tables will also be smaller since they will only consider the mapping of sets of virtual vertices.

Example 2.20 Let us revisit the case in Example 2.19 and consider that we are compressing the hypergraph into three virtual vertices: $v_1' = 1, 2$, $v_2' = 3, 4, 5$, $v_3' = 6, 7$. Then there would be two hyperedges: $e_1' = (v_1', v_3')$ with frequency 2 (corresponding to Q_1 and Q_2 in the original hypergraph) and $e_2' = (v_1', v_2')$ with frequency 1 (corresponding to Q_3). The hyperedges representing queries Q_4 and Q_5 would be local (i.e., not spanning virtual vertices) so no hyperedges are required in the compressed hypergraph. This is shown in Fig. 2.24. ♦

Performing the k-way balanced min-cut partitioning on the compressed hypergraph can be performed much faster and the resulting mapping table will be smaller due to the reduced size of the graph.

SWORD incorporates replication in the compressed hypergraph. It first determines, for each virtual vertex, how many replicas are required. It does this by using the tuple-level access pattern statistics for each tuple t_j in each virtual vertex v_i', namely its read frequency f_{ij}^r and its write frequency f_{ij}^w. Using these, it computes the average read and write frequencies (ARF and AWF, respectively) of virtual vertex v_i' as follows:

$$ARF(v_i') = \frac{\sum_j f_{ij}^r}{\log S(v_i')} \quad \text{and} \quad AWF(v_i') = \frac{\sum_j f_{ij}^w}{\log S(v_i')}$$

$S(v_i')$ is the size of each virtual vertex (in terms of the number of actual vertices mapped to it) and its log is taken to compensate for the skew in the sizes of virtual vertices (so, these are size-compensated averages). From these, SWORD defines a

replication factor, $\mathcal{R} = \frac{AWF(v_i')}{ARWF(v_i')}$ and a user-specified threshold δ $(0 < \delta < 1)$ is defined. The number of replicas (#_*rep*) for virtual vertex v_i' is then given as

$$\#_rep(v_i') = \begin{cases} 1 & \text{if } \mathcal{R} \geq \delta \\ ARF(v_i') & \text{otherwise} \end{cases}$$

Once the number of replicas for each virtual vertex is determined, these are added to the compressed hypergraph and assigned to hyperedges in a way that minimizes the min-cut in the partitioning algorithm. We ignore the details of this assignment.

2.4 Adaptive Approaches

The work described in this chapter generally assumes a static environment where design is conducted only once and this design can persist. Reality, of course, is quite different. Both physical (e.g., network characteristics, available storage at various sites) and logical (e.g., workload) changes occur necessitating redesign of the database. In a dynamic environment, the process becomes one of design-redesign-materialization of the redesign. When things change, the simplest approach is to redo the distribution design from scratch. For large or highly dynamic systems, this is not quite realistic as the overhead of redesign is likely to be very high. A preferred approach is to perform incremental redesign, focusing only on the parts of the database that are likely to be affected by the changes. The incremental redesign can either be done every time a change is detected or periodically where changes are batched and evaluated at regular intervals.

Most of the work in this area has focused on changes in the workload (queries and transactions) over time and those are what we focus in this section. While some work in this area has focused on the fragment-then-allocate approach, most follow the combined approach. In the former case one alternative that has been proposed involves a split phase where fragments are further subdivided based on the changed application requirements until no further subdivision is profitable based on a cost function. At this point, the merging phase starts where fragments that are accessed together by a set of applications are merged into one fragment. We will focus more on the dynamic combined approaches that perform incremental redesign as the workload changes.

The objective in the adaptive approaches is the same as the workload-aware partitioning strategies discussed in Sect. 2.3.2: to minimize the number of distributed queries and ensure that the data for each query is local. Within this context, there are three interrelated issues that need to be addressed in adaptive distribution design:

1. How to detect workload changes that require changes in the distribution design?
2. How to determine which data items are going to be affected in the design?
3. How to perform the changes in an efficient manner?

In the remainder we discuss each of these issues.

2.4.1 Detecting Workload Changes

This is a difficult issue on which there is not much work. Most of the adaptive techniques that have been proposed assume that the change in the workload is detected, and simply focus on the migration problem. To be able to detect workload changes, the incoming queries need to be monitored. One way to do this is to periodically examine the system logs, but this may have high overhead, especially in highly dynamic systems. An alternative is to continuously monitor the workload within the DBMS. In the SWORD system we discussed above, the system monitors the percentage increase in the number of distributed transactions and considers that the system has changed sufficiently to require a reconfiguration if this percentage increase is above a defined threshold. As another example, the E-Store system monitors both system-level metrics and tuple-level access. It starts by collecting system-level metrics at each computing node using OS facilities. E-Store currently focuses primarily on detecting workload imbalances across the computing nodes, and therefore only collects CPU utilization data. If the CPU utilization imbalance exceeds a threshold, then it invokes more fine-grained tuple-level monitoring to detect the affected items (see next section). Although imbalance in CPU utilization may be a good indicator of possible performance problems, it is too simple to capture more significant workload changes. It is possible, of course, to do more sophisticated monitoring, e.g., one can create a profile that looks at the frequency of each query in a given time period, the percentage of queries that meet (or exceed) their agreed-upon latencies (as captured in a service level agreement, perhaps), and others. Then it can be decided whether the changes in the profile require redesign, which can be done either continuously (i.e., every time the monitor registers information) or periodically. The challenge here is to do this efficiently without intruding on the system performance. This is an open research area that has not been properly studied.

2.4.2 Detecting Affected Items

Once a change is detected in the workload the next step is to determine what data items are affected and need to be migrated to address this change. How this is done is very much dependent on the detection method. For example, if the system is monitoring the frequency of queries and detects changes, then the queries will identify the data items. It is possible to generalize from individual queries to query templates in order to capture "similar" queries that might also be affected by the changes. This is done in the Apollo system where each constant is replaced by a wildcard. For example, the query

```
SELECT PNAME FROM PROJ WHERE BUDGET>200000 AND LOC = "London"
```

would be generalized to

```
SELECT PNAME FROM PROJ WHERE BUDGET>? AND LOC = "?"
```

While this reduces the granularity of determining the exact set of data items that are affected, it may allow the detection of additional data items that might be affected by similar queries and reduce the frequency of changes that are necessary.

The E-Store system starts tuple-level monitoring once it detects a system load imbalance. For a short period, it collects access data to the tuples in each computing node (i.e., each partition) and determines the "hot" tuples, which are the top-k most frequently accessed tuples within a time period. To do this, it uses a histogram for each tuple that is initialized when the tuple-level monitoring is enabled and updated as access happens within the monitoring window. At the end of this time period, the top-k list is assembled. The monitoring software gathers these lists and generates a global top-k list of hot tuples—these are the data items that need to be migrated. A side-effect is the determination of cold tuples; of particular importance are tuples that were previously hot and have since become cold. The determination of the time window for tuple-level monitoring and the value of k are parameters set by the database administrator.

2.4.3 Incremental Reconfiguration

As noted earlier, the naive approach to perform redesign is to redo the entire data partitioning and distribution. While this may be of interest in environments where workload change occurs infrequently, in most cases, the overhead of redesign is too high to do it from scratch. The preferred approach is to apply the changes incrementally by migrating data; in other words, we only look at the changed workload and the data items that are affected, and move them around.[8] So, in this section, we focus on incremental approaches.

Following from the previous section, one obvious approach is to use an incremental graph partitioning algorithm that reacts to changes in the graph representation we discussed. This has been followed in the SWORD system discussed above and in AdaptCache, both of which represent usage as hypergraphs and perform incremental partitioning on these graphs. The incremental graph partitioning initiates data migration for reconfiguration.

The E-Store system we have been discussing takes a more sophisticated approach. Once the set of hot tuples are identified, a migration plan is prepared that identifies where the hot tuples should be moved and what reallocation of cold tuples is necessary. This can be posed as an optimization problem that creates a balanced load across the computing nodes (balance is defined as average-load-across-nodes

[8]The research in this area has exclusively focused on horizontal partitioning, which will be our focus here as well, meaning that our units of migration are individual tuples.

± a threshold value), but solving this optimization problem in real time for online reconfiguration is not easy, so it uses approximate placement approaches (e.g., greedy, first-fit) to generate the reconfiguration plan. Basically, it first determines the appropriate computing nodes at which each hot tuple should be located, then addresses cold tuples, if necessary due to remaining imbalance, by moving them in blocks. So, the generated reconfiguration plan addresses the migration of hot tuples individually, but the migration of cold tuples as blocks. As part of the plan a coordinating node is determined to manage the migration, and this plan is an input to the Squall reconfiguration system.

Squall performs reconfiguration and data migration in three steps. In the first step, the coordinator identified in the reconfiguration plan initializes the system for migration. This step includes the coordinating obtaining exclusive access control to all of these partitions through a transaction as we will discuss in Chap. 5. Then the coordinator asks each site to identify the tuples that will be moving out of the local partition and the tuples that will be coming in. This analysis is done on the metadata so can be done quickly after which each site notifies the coordinator and the initialization transaction terminates. In the second step, the coordinator instructs each site to do the data migration. This is critical as there are queries accessing the data as it is being moved. If a query is executing at a given computing node where the data is supposed to be according to the reconfiguration plan but the required tuples are not locally available, Squall pulls the missing tuples to process the query. This is done in addition to the normal migration of the data according to the reconfiguration plan. In other words, in order to execute the queries in a timely fashion, Squall performs on-demand movement in addition to its normal migration. Once this step is completed, each node informs the coordinator, which then starts the final termination step and informs each node that reconfiguration is completed. These three steps are necessary for Squall to be able to perform migration while executing user queries at the same time rather than stopping all query execution, performing the migration and then restarting the query execution.

Another approach is *database cracking*, which is an adaptive indexing technique that targets dynamic, hard to predict workloads and scenarios where there is little or no idle time to devote to workload analysis and index building. Database cracking works by continuously reorganizing data to match the query workload. Every query is used as an advice on how the data should be stored. Cracking does this by building and refining indices partially and incrementally as part of query processing. By reacting to every single query with lightweight actions, database cracking manages to adapt to a changing workload instantly. As more queries arrive, the indices are refined, and the performance improves, eventually reaching the optimal performance, i.e., the performance we would get from a manually tuned system.

The main idea in the original database cracking approach is that the data system reorganizes one column of the data at a time and only when touched by a query. In other words, the reorganization utilizes the fact that the data is already read and decides how to refine it in the best way. Effectively the original cracking approach overloads the select operator of a database system and uses the predicates of each query to determine how to reorganize the relevant column. The first time an attribute

A is required by a query, a copy of the base column A is created, called the cracker column of A. Each select operator on A triggers the physical reorganization of the cracker column based on the requested range of the query. Entries with a key that is smaller than the lower bound are moved before the lower bound, while entries with a key that is greater than the upper bound are moved after the upper bound in the respective column. The partitioning information for each cracker column is maintained in an AVL-tree, the cracker index. Future queries on column A search the cracker index for the partition where the requested range falls. If the requested key already exists in the index, i.e., if past queries have cracked on exactly those ranges, then the select operator can return the result immediately. Otherwise, the select operator refines on the fly the column further, i.e., only the partitions/pieces of the column where the predicates fall will be reorganized (at most two partitions at the boundaries of the range). Progressively the column gets more "ordered" with more but smaller pieces.

The primary concept in database cracking and its basic techniques can be extended to partition data in a distributed setting, i.e., to store data across a set of nodes using incoming queries as an advice. Each time a node needs a specific part of the data for a local query but the data does not exist in this node, this information can be used as a hint that the data could be moved to this node. However, contrary to the in-memory database cracking methods where the system reacts immediately to every query, in a distributed setting we need to consider that moving the data is more expensive. At the same time, for the same reason, the benefit that future queries may have is going to be more significant. In fact, the same trade-off has already been studied in variations of the original database cracking approach to optimize for disk-based data. The net effect is twofold: (1) instead of reacting with every query, we should wait for more workload evidence before we embark on expensive data reorganization actions, and (2) we should apply "heavier" reorganizations to utilize the fact that reading and writing data is more expensive out of memory. We expect future approaches to explore and develop such adaptive indexing methods to benefit from effective partitioning in scenarios where the workload is not easy to predict, and there is not enough time to fully sort/partition all data before the first query arrives.

2.5 Data Directory

The final distribution design issue we discuss is related to data directory. The distributed database schema needs to be stored and maintained by the system. This information is necessary during distributed query optimization, as we will discuss later. The schema information is stored in a *catalog/data dictionary/directory* (simply directory). A directory is a metadatabase that stores a number of information such as schema and mapping definitions, usage statistics, access control information, and the like.

In the case of a distributed DBMS, schema definition is done at the global level (i.e., the global conceptual schema—GCS) as well as at the local sites (i.e., local conceptual schemas—LCSs). GCS defines the overall database while each LCS describes data at that particular site. Consequently, there are two types of directories: a *global directory/dictionary* (GD/D)[9] that describes the database schema as the end users see it, and the *local directory/dictionary* (LD/D) that describes the local mappings and describes the schema at each site. Thus, the local database management components are integrated by means of global DBMS functions.

As stated above, the directory is itself a database that contains *metadata* about the actual data stored in the database. Therefore, the techniques we discussed in this chapter, with respect to distributed database design also apply to directory management, but in much simpler manner. Briefly, a directory may be either *global* to the entire database or *local* to each site. In other words, there might be a single directory containing information about all the data in the database (the GD/D), or a number of directories, each containing the information stored at one site (the LD/D). In the latter case, we might either build hierarchies of directories to facilitate searches or implement a distributed search strategy that involves considerable communication among the sites holding the directories.

A second issue is replication. There may be a *single* copy of the directory or *multiple* copies. Multiple copies would provide more reliability, since the probability of reaching one copy of the directory would be higher. Furthermore, the delays in accessing the directory would be lower, due to less contention and the relative proximity of the directory copies. On the other hand, keeping the directory up-to-date would be considerably more difficult, since multiple copies would need to be updated. Therefore, the choice should depend on the environment in which the system operates and should be made by balancing such factors as the response-time requirements, the size of the directory, the machine capacities at the sites, the reliability requirements, and the volatility of the directory (i.e., the amount of change experienced by the database, which would cause a change to the directory).

2.6 Conclusion

In this chapter, we presented the techniques that can be used for distributed database design with special emphasis on the partitioning and allocation issues. We have discussed, in detail, the algorithms that one can use to fragment a relational schema in various ways. These algorithms have been developed quite independently and there is no underlying design methodology that combines the horizontal and vertical partitioning techniques. If one starts with a global relation, there are algorithms to decompose it horizontally as well as algorithms to decompose it vertically into a set of fragment relations. However, there are no algorithms that fragment a

[9] In the remainder, we will simply refer to this as the *global directory*.

global relation into a set of fragment relations some of which are decomposed horizontally and others vertically. It is commonly pointed out that most real-life fragmentations would be mixed, i.e., would involve both horizontal and vertical partitioning of a relation, but the methodology research to accomplish this is lacking. If this design methodology is to be followed, what is needed is a distribution design methodology which encompasses the horizontal and vertical fragmentation algorithms and uses them as part of a more general strategy. Such a methodology should take a global relation together with a set of design criteria and come up with a set of fragments some of which are obtained via horizontal and others obtained via vertical fragmentation.

We also discussed techniques that do not separate fragmentation and allocation steps—the way data is partitioned dictates how it is allocated or vice versa. These techniques typically have two characteristics. The first is that they exclusively focus on horizontal partitioning. The second is that they are more fine-grained and the unit of allocation is a tuple; fragments at each site "emerge" as the union of tuples from the same relation assigned to that site.

We finally discussed adaptive techniques that take into account changes in workload. These techniques again typically involve horizontal partitioning, but monitor the workload changes (both in terms of the query set and in terms of the access patterns) and adjust the data partitioning accordingly. The naïve way achieving this is by to do new batch run of the partitioning algorithm, but this is obviously not desired. Therefore, the better algorithms in this class adjust data distribution incrementally.

2.7 Bibliographic Notes

Distributed database design has been studied systematically since the early years of the technology. An early paper that characterizes the design space is [Levin and Morgan 1975]. Davenport [1981], Ceri et al. [1983], and Ceri et al. [1987] provide nice overviews of the design methodology. Ceri and Pernici [1985] discuss a particular methodology, called DATAID-D, which is similar to what we presented in Fig. 2.1. Other attempts to develop a methodology are due to Fisher et al. [1980], Dawson [1980], Hevner and Schneider [1980], and Mohan [1979].

Most of the known results about fragmentation have been covered in this chapter. Work on fragmentation in distributed databases initially concentrated on horizontal fragmentation. The discussion on that topic is mainly based on [Ceri et al. 1982b] and [Ceri et al. 1983]. Data partitioning in parallel DBMS is treated in [DeWitt and Gray 1992]. The topic of vertical fragmentation for distribution design has been addressed in several papers (e.g., Navathe et al. [1984] and Sacca and Wiederhold [1985]). The original work on vertical fragmentation goes back to Hoffer's dissertation [Hoffer 1975, Hoffer and Severance 1975] and to Niamir [1978] and Hammer and Niamir [1979]. McCormick et al. [1972] present the bond

energy algorithm that has been adopted to vertical fragmentation by Hoffer and Severance [1975] and Navathe et al. [1984].

The investigation of file allocation problem on wide area networks goes back to Chu's work [Chu 1969, 1973]. Most of the early work on this has been covered in the excellent survey by Dowdy and Foster [1982]. Some theoretical results are reported by Grapa and Belford [1977] and Kollias and Hatzopoulos [1981]. The distributed data allocation work dates back to the mid-1970s to the works of Eswaran [1974] and others. In their earlier work, Levin and Morgan [1975] concentrated on data allocation, but later they considered program and data allocation together [Morgan and Levin 1977]. The distributed data allocation problem has been studied in many specialized settings as well. Work has been done to determine the placement of computers and data in a wide area network design [Gavish and Pirkul 1986]. Channel capacities have been examined along with data placement [Mahmoud and Riordon 1976] and data allocation on supercomputer systems [Irani and Khabbaz 1982] as well as on a cluster of processors [Sacca and Wiederhold 1985]. An interesting work is the one by Apers [1981], where the relations are optimally placed on the nodes of a virtual network, and then the best matching between the virtual network nodes and the physical network is found. The isomorphism of data allocation problem to single commodity warehouse location problem is due to Ramamoorthy and Wah [1983]. For other solution approaches, the sources are as follows: knapsack problem solution [Ceri et al. 1982a], branch-and-bound techniques [Fisher and Hochbaum 1980], and network flow algorithms [Chang and Liu 1982].

The Schism approach to combined partitioning (Sect. 2.3.2) is due to Curino et al. [2010] and SWORD is due to Quamar et al. [2013]. Other works along these lines are [Zilio 1998], [Rao et al. 2002], and [Agrawal et al. 2004], which mostly focus on partitioning for parallel DBMSs.

An early adaptive technique is discussed by Wilson and Navathe [1986]. Limited redesign, in particular, the materialization issue, is studied in [Rivera-Vega et al. 1990, Varadarajan et al. 1989]. Complete redesign and materialization issues have been studied in [Karlapalem et al. 1996, Karlapalem and Navathe 1994, Kazerouni and Karlapalem 1997]. Kazerouni and Karlapalem [1997] describe the stepwise redesign methodology that we referred to in Sect. 2.4. AdaptCache is described in [Asad and Kemme 2016].

The impact of workload changes on distributed/parallel DBMSs and the desirability of localizing data for each transaction have been studied by Pavlo et al. [2012] and Lin et al. [2016]. There are a number of works that address adaptive partitioning in the face of these changes. Our discussion focused on E-Store [Taft et al. 2014] as an exemplar. E-Store implements the E-Monitor and E-Planner systems, respectively, for monitoring and detecting workload changes, and for detecting affected items to create a migration plan. For actual migration it uses an optimized version of Squall [Elmore et al. 2015]. There are other works along the same vein; for example, P-Store [Taft et al. 2018] predicts load demands (as opposed to E-Store reacting to them).

The log-inspection-based determination of workload changes is due to Levandoski et al. [2013].

One work that focuses on detecting workload shifts for autonomic computing is described in Holze and Ritter [2008]. The Apollo system, which we referred to in discussion how to detect data items that are affected, and that abstracts queries to query templates in order to do predictive computation is described in Glasbergen et al. [2018].

Database cracking as a concept has been studied in the context of main-memory column-stores [Idreos et al. 2007b, Schuhknecht et al. 2013]. The cracking algorithms have been adapted to work for many core database architecture issues such as: updates to incrementally and adaptively absorb data changes [Idreos et al. 2007a], multiattribute queries to reorganize whole relations as opposed to only columns [Idreos et al. 2009], to use also the join operator as a trigger for adaptation [Idreos 2010], concurrency control to deal with the problem that cracking effectively turns reads into writes [Graefe et al. 2014, 2012], and partition-merge-like logic to provide cracking algorithms that can balance index convergence versus initialization costs [Idreos et al. 2011]. In addition, tailored benchmarks have been developed to stress-test critical features such as how quickly an algorithm adapts [Graefe et al. 2010]. Stochastic database cracking [Halim et al. 2012] shows how to be robust on various workloads, and Graefe and Kuno [2010b] show how adaptive indexing can apply to key columns. Finally, recent work on parallel adaptive indexing studies CPU-efficient implementations and proposes cracking algorithms to utilize multicores [Pirk et al. 2014, Alvarez et al. 2014] or even idle CPU time [Petraki et al. 2015].

The database cracking concept has also been extended to broader storage layout decisions, i.e., reorganizing base data (columns/rows) according to incoming query requests [Alagiannis et al. 2014], or even about which data should be loaded [Idreos et al. 2011, Alagiannis et al. 2012]. Cracking has also been studied in the context of Hadoop [Richter et al. 2013] for local indexing in each node as well as for improving more traditional disk-based indexing which forces reading data at the granularity of pages and where writing back the reorganized data needs to be considered as a major overhead [Graefe and Kuno 2010a].

Exercises

Problem 2.1 (*) Given relation EMP as in Fig. 2.2, let p_1: TITLE < "Programmer" and p_2: TITLE > "Programmer" be two simple predicates. Assume that character strings have an order among them, based on the alphabetical order.

(a) Perform a horizontal fragmentation of relation EMP with respect to $\{p_1, p_2\}$.

(b) Explain why the resulting fragmentation (EMP_1, EMP_2) does not fulfill the correctness rules of fragmentation.

(c) Modify the predicates p_1 and p_2 so that they partition EMP obeying the correctness rules of fragmentation. To do this, modify the predicates, compose

all minterm predicates and deduce the corresponding implications, and then perform a horizontal fragmentation of EMP based on these minterm predicates. Finally, show that the result has completeness, reconstruction, and disjointness properties.

Problem 2.2 (*) Consider relation ASG in Fig. 2.2. Suppose there are two applications that access ASG. The first is issued at five sites and attempts to find the duration of assignment of employees given their numbers. Assume that managers, consultants, engineers, and programmers are located at four different sites. The second application is issued at two sites where the employees with an assignment duration of less than 20 months are managed at one site, whereas those with longer duration are managed at a second site. Derive the primary horizontal fragmentation of ASG using the foregoing information.

Problem 2.3 Consider relations EMP and PAY in Fig. 2.2. EMP and PAY are horizontally fragmented as follows:

$$\mathrm{EMP}_1 = \sigma_{\mathrm{TITLE}=\text{“Elect. Eng.”}}(\mathrm{EMP})$$

$$\mathrm{EMP}_2 = \sigma_{\mathrm{TITLE}=\text{“Syst. Anal.”}}(\mathrm{EMP})$$

$$\mathrm{EMP}_3 = \sigma_{\mathrm{TITLE}=\text{“Mech. Eng.”}}(\mathrm{EMP})$$

$$\mathrm{EMP}_4 = \sigma_{\mathrm{TITLE}=\text{“Programmer”}}(\mathrm{EMP})$$

$$\mathrm{PAY}_1 = \sigma_{\mathrm{SAL}\geq 30000}(\mathrm{PAY})$$

$$\mathrm{PAY}_2 = \sigma_{\mathrm{SAL}<30000}(\mathrm{PAY})$$

Draw the join graph of EMP $\ltimes_{\mathrm{TITLE}}$ PAY. Is the graph simple or partitioned? If it is partitioned, modify the fragmentation of either EMP or PAY so that the join graph of EMP $\ltimes_{\mathrm{TITLE}}$ PAY is simple.

Problem 2.4 Give an example of a CA matrix where the split point is not unique and the partition is in the middle of the matrix. Show the number of shift operations required to obtain a single, unique split point.

Problem 2.5 (**) Given relation PAY as in Fig. 2.2, let p_1 : SAL $<$ 30000 and p_2 : SAL $\geq$ 30000 be two simple predicates. Perform a horizontal fragmentation of PAY with respect to these predicates to obtain PAY_1 and PAY_2. Using the fragmentation of PAY, perform further derived horizontal fragmentation for EMP. Show completeness, reconstruction, and disjointness of the fragmentation of EMP.

Problem 2.6 (**) Let $Q = \{q_1, \ldots, q_5\}$ be a set of queries, $\mathrm{A} = \{\mathrm{A}_1, \ldots, \mathrm{A}_5\}$ be a set of attributes, and $S = \{S_1, S_2, S_3\}$ be a set of sites. The matrix of Fig. 2.25a describes the attribute usage values and the matrix of Fig. 2.25b gives the application access frequencies. Assume that $ref_i(q_k) = 1$ for all q_k and S_i and that A_1 is the key attribute. Use the bond energy and vertical partitioning algorithms to obtain a vertical fragmentation of the set of attributes in A.

Problem 2.7 (**) Write an algorithm for derived horizontal fragmentation.

(a)

	A_1	A_2	A_3	A_4	A_5
q_1	0	1	1	0	1
q_2	1	1	1	0	1
q_3	1	0	0	1	1
q_4	0	0	1	0	0
q_5	1	1	1	0	0

(b)

	S_1	S_2	S_3
q_1	10	20	0
q_2	5	0	10
q_3	0	35	5
q_4	0	10	0
q_5	0	15	0

Fig. 2.25 Attribute usage values and application access frequencies in Exercise 3.6

Problem 2.8 (**) Assume the following view definition:

```
CREATE VIEW   EMPVIEW(ENO, ENAME, PNO, RESP)
AS     SELECT EMP.ENO, EMP.ENAME, ASG.PNO, ASG.RESP
       FROM   EMP JOIN ASG
       WHERE  DUR=24
```

is accessed by application q_1, located at sites 1 and 2, with frequencies 10 and 20, respectively. Let us further assume that there is another query q_2 defined as

```
SELECT ENO, DUR
FROM   ASG
```

which is run at sites 2 and 3 with frequencies 20 and 10, respectively. Based on the above information, construct the $use(q_i, A_j)$ matrix for the attributes of both relations EMP and ASG. Also construct the affinity matrix containing all attributes of EMP and ASG. Finally, transform the affinity matrix so that it could be used to split the relation into two vertical fragments using heuristics or BEA.

Problem 2.9 (**) Formally define the three correctness criteria for derived horizontal fragmentation.

Problem 2.10 (*) Given a relation $R(K, A, B, C)$ (where K is the key) and the following query:

```
SELECT *
FROM   R
WHERE  R.A=10 AND R.B=15
```

(a) What will be the outcome of running PHF on this query?
(b) Does the COM_MIN algorithm produce in this case a complete and minimal predicate set? Justify your answer.

Problem 2.11 (*) Show that the bond energy algorithm generates the same results using either row or column operation.

Problem 2.12 (**) Modify algorithm SPLIT to allow n-way partitioning, and compute the complexity of the resulting algorithm.

Problem 2.13 (**) Formally define the three correctness criteria for hybrid fragmentation.

Problem 2.14 Discuss how the order in which the two basic fragmentation schemas are applied in hybrid fragmentation affects the final fragmentation.

Problem 2.15 (**) Describe how the following can be properly modeled in the database allocation problem.

(a) Relationships among fragments
(b) Query processing
(c) Integrity enforcement
(d) Concurrency control mechanisms

Problem 2.16 (**) Consider the various heuristic algorithms for the database allocation problem.

(a) What are some of the reasonable criteria for comparing these heuristics? Discuss.
(b) Compare the heuristic algorithms with respect to these criteria.

Problem 2.17 (*) Pick one of the heuristic algorithms used to solve the DAP, and write a program for it.

Problem 2.18 (**) Assume the environment of Exercise 3.8. Also assume that 60% of the accesses of query q_1 are updates to `PNO` and `RESP` of view `EMPVIEW` and that `ASG.DUR` is not updated through `EMPVIEW`. In addition, assume that the data transfer rate between site 1 and site 2 is half of that between site 2 and site 3. Based on the above information, find a reasonable fragmentation of `ASG` and `EMP` and an optimal replication and placement for the fragments, assuming that storage costs do not matter here, but copies are kept consistent.

Hint: Consider horizontal fragmentation for `ASG` based on `DUR` = 24 predicate and the corresponding derived horizontal fragmentation for `EMP`. Also look at the affinity matrix obtained in Example 2.7 for `EMP` and `ASG` together, and consider whether it would make sense to perform a vertical fragmentation for `ASG`.