M. Tamer Özsu · Patrick Valduriez

Principles of Distributed Database Systems

Springer

Principles of Distributed Database Systems

M. Tamer Özsu • Patrick Valduriez

Principles of Distributed Database Systems

Fourth Edition

Springer

M. Tamer Özsu
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada

Patrick Valduriez
Inria and LIRMM
University of Montpellier
Montpellier, France

The first two editions of this book were published by: Pearson Education, Inc.

ISBN 978-3-030-26252-5 ISBN 978-3-030-26253-2 (eBook)
https://doi.org/10.1007/978-3-030-26253-2

To our families
and our parents
M.T.Ö. and P.V.

Preface

The first edition of this book appeared in 1991 when the technology was new and there were not too many products. In the Preface to the first edition, we had quoted Michael Stonebraker who claimed in 1988 that in the following 10 years, centralized DBMSs would be an "antique curiosity" and most organizations would move towards distributed DBMSs. That prediction has certainly proved to be correct, and a large proportion of the systems in use today are either distributed or parallel—commonly referred to as scale-out systems. When we were putting together the first edition, undergraduate and graduate database courses were not as prevalent as they are now; so the initial version of the book contained lengthy discussions of centralized solutions before introducing their distributed/parallel counterparts. Times have certainly changed on that front as well, and now, it is hard to find a graduate student who does not have at least some rudimentary knowledge of database technology. Therefore, a graduate-level textbook on distributed/parallel database technology needs to be positioned differently today. That was our objective in this edition while maintaining the many new topics we introduced in the third edition. The main revisions introduced in this fourth edition are the following:

1. Over the years, the motivations and the environment for this technology have somewhat shifted (Web, cloud, etc.). In light of this, the introductory chapter needed a serious refresh. We revised the introduction with the aim of a more contemporary look at the technology.
2. We have added a new chapter on big data processing to cover distributed storage systems, data stream processing, MapReduce and Spark platforms, graph analytics, and data lakes. With the proliferation of these systems, systematic treatment of these topics is essential.
3. Similarly, we addressed the growing influence of NoSQL systems by devoting a new chapter to it. This chapter covers the four types of NoSQL (key-value stores, document stores, wide column systems, and graph DBMSs), as well as NewSQL systems and polystores.
4. We have combined the database integration and multidatabase query processing chapters from the third edition into a uniform chapter on database integration.

5. We undertook a major revision of the web data management discussion that previously focused mostly on XML to refocus on RDF technology, which is more prevalent at this time. We now discuss, in this chapter, web data integration approaches, including the important issue of data quality.
6. We have revised and updated the peer-to-peer data management chapter and included a lengthy discussion of blockchain.
7. As part of our cleaning the previous chapters, we condensed the query processing and transaction management chapters by removing the fundamental centralized techniques and focused these chapters on distributed/parallel techniques. In the process, we included some topics that have since gained importance, such as dynamic query processing (eddies) and Paxos consensus algorithm and its use in commit protocols.
8. We updated the parallel DBMS chapter by clarifying the objectives, in particular, scale-up versus scale-out, and discussing parallel architectures that include UMA or NUMA. We also added a new section of parallel sorting algorithms and variants of parallel join algorithms to exploit large main memories and multicore processors that are prevalent today.
9. We updated the distribution design chapter by including a lengthy discussion of modern approaches that combine fragmentation and allocation. By rearranging material, this chapter is now central to data partitioning for both the distributed and parallel data management discussions in the remainder of the book.
10 Although object technology continues to play a role in information systems, its importance in distributed/parallel data management has declined. Therefore, we removed the chapter on object databases from this edition.

As is evident, the entire book and every chapter have seen revisions and updates for a more contemporary treatment. The material we removed in the process is not lost—they are included as online appendices and appear on the book's web page: https://cs.uwaterloo.ca/ddbs. We elected to make these available online rather than in the print version to keep the size of the book reasonable (which also keeps the price reasonable). The web site also includes presentation slides that can be used to teach from the book as well as solutions to most of the exercises (available only to instructors who have adopted the book for teaching).

As in previous editions, many colleagues helped with this edition of the book whom we would like to thank (in no specific order). Dan Olteanu provided a nice discussion of two optimizations that can significantly reduce the maintenance time of materialized views in Chap. 3. Phil Bernstein provided leads for new papers on the multiversion transaction management that resulted in updates to that discussion in Chap. 5. Khuzaima Daudjee was also helpful in providing a list of more contemporary publications on distributed transaction processing that we include in the bibliographic notes section of that chapter. Ricardo Jimenez-Peris contributed text on high-performance transaction systems that is included in the same chapter. He also contributed a section on LeanXcale in the NoSQL, NewSQL, and polystores chapter. Dennis Shasha reviewed the new blockchain section in the P2P chapter. Michael Carey read the big data, NoSQL, NewSQL and

polystores, and parallel DBMS chapters and provided extremely detailed comments that improved those chapters considerably. Tamer's students Anil Pacaci, Khaled Ammar and postdoc Xiaofei Zhang provided extensive reviews of the big data chapter, and texts from their publications are included in this chapter. The NoSQL, NewSQL, and polystores chapter includes text from publications of Boyan Kolev and Patrick's student Carlyna Bondiombouy. Jim Webber reviewed the section on Neo4j in that chapter. The characterization of graph analytics systems in that chapter is partially based on Minyang Han's master's thesis where he also proposes GiraphUC approach that is discussed in that chapter. Semih Salihoglu and Lukasz Golab also reviewed and provided very helpful comments on parts of this chapter. Alon Halevy provided comments on the WebTables discussion in Chap. 12. The data quality discussion in web data integration is contributed by Ihab Ilyas and Xu Chu. Stratos Idreos was very helpful in clarifying how database cracking can be used as a partitioning approach and provided text that is included in Chap. 2. Renan Souza and Fabian Stöter reviewed the entire book.

The third edition of the book introduced a number of new topics that carried over to this edition, and a number of colleagues were very influential in writing those chapters. We would like to, once again, acknowledge their assistance since their impact is reflected in the current edition as well. Renée Miller, Erhard Rahm, and Alon Halevy were critical in putting together the discussion on database integration, which was reviewed thoroughly by Avigdor Gal. Matthias Jarke, Xiang Li, Gottfried Vossen, Erhard Rahm, and Andreas Thor contributed exercises to this chapter. Hubert Naacke contributed to the section on heterogeneous cost modeling and Fabio Porto to the section on adaptive query processing. Data replication (Chap. 6) could not have been written without the assistance of Gustavo Alonso and Bettina Kemme. Esther Pacitti also contributed to the data replication chapter, both by reviewing it and by providing background material; she also contributed to the section on replication in database clusters in the parallel DBMS chapter. Peer-to-peer data management owes a lot to the discussions with Beng Chin Ooi. The section of this chapter on query processing in P2P systems uses material from the PhD work of Reza Akbarinia and Wenceslao Palma, while the section on replication uses material from the PhD work of Vidal Martins.

We thank our editor at Springer Susan Lagerstrom-Fife for pushing this project within Springer and also pushing us to finish it in a timely manner. We missed almost all of her deadlines, but we hope the end result is satisfactory.

Finally, we would be very interested to hear your comments and suggestions regarding the material. We welcome any feedback, but we would particularly like to receive feedback on the following aspects:

1. Any errors that may have remained despite our best efforts (although we hope there are not many);

2. Any topics that should no longer be included and any topics that should be added or expanded;
3. Any exercises that you may have designed that you would like to be included in the book.

Waterloo, Canada
Montpellier, France
June 2019

M. Tamer Özsu (tamer.ozsu@uwaterloo.ca)
Patrick Valduriez (patrick.valduriez@inria.fr)

Contents

Chapter 1
Introduction

The current computing environment is largely distributed—computers are connected to Internet to form a worldwide distributed system. Organizations have geographically distributed and interconnected data centers, each with hundreds or thousands of computers connected with high-speed networks, forming mixture of distributed and parallel systems (Fig. 1.1). Within this environment, the amount of data that is captured has increased dramatically. Not all of this data is stored in database systems (in fact a small portion is) but there is a desire to provide some sort of data management capability on these widely distributed data. This is the scope of distributed and parallel database systems, which have moved from a small part of the worldwide computing environment a few decades ago to mainstream. In this chapter, we provide an overview of this technology, before we examine the details in subsequent chapters.

1.1 What Is a Distributed Database System?

We define a *distributed database* as *a collection of multiple, logically interrelated databases located at the nodes of a distributed system*. A *distributed database management system* (distributed DBMS) is then defined as *the software system that permits the management of the distributed database and makes the distribution transparent to the users*. Sometimes "distributed database system" (distributed DBMS) is used to refer jointly to the distributed database and the distributed DBMS. The two important characteristics are that data is logically interrelated and that it resides on a distributed system.

The existence of a distributed system is an important characteristic. In this context, we define a *distributed computing system* as a number of interconnected autonomous processing elements (PEs). The capabilities of these processing elements may differ, they may be heterogeneous, and the interconnections might be

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_1

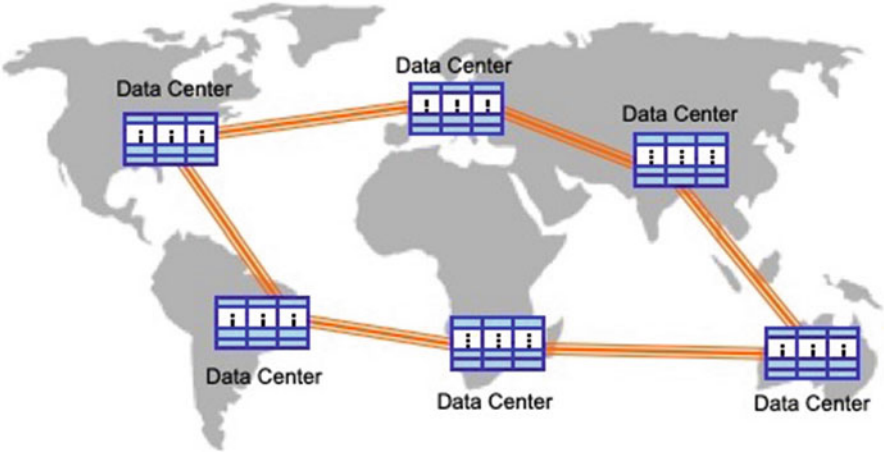

Fig. 1.1 Geographically distributed data centers

different, but the important aspect is that PEs do not have access to each other's state, which they can only learn by exchanging messages that incur a communication cost. Therefore, when data is distributed, its management and access in a logically integrated manner requires special care from the distributed DBMS software.

A distributed DBMS is not a "collection of files" that can be individually stored at each PE of a distributed system (usually called "site" of a distributed DBMS); data in a distributed DBMS is interrelated. We will not try to be very specific with what we mean by interrelated, because the requirements differ depending on the type of data. For example, in the case of relational data, different relations or their partitions might be stored at different sites (more on this in Chap. 2), requiring join or union operations to answer queries that are typically expressed in SQL. One can usually define a *schema* of this distributed data. At the other extreme, data in NoSQL systems (discussed further in Chap. 11) may have a much looser definition of interrelatedness; for example, it may be vertices of a graph that might be stored at different sites.

The upshot of this discussion is that a distributed DBMS is *logically integrated* but *physically distributed*. What this means is that a distributed DBMS gives the users the view of a unified database, while the underlying data is physically distributed.

As noted above, we typically consider two types of distributed DBMSs: geographically distributed (commonly referred to as *geo-distributed*) and single location (or single site) . In the former, the sites are interconnected by wide area networks that are characterized by long message latencies and higher error rates. The latter consist of systems where the PEs are located in close proximity allowing much faster exchanges leading to shorter (even negligible with new technologies) message latencies and very low error rates. Single location distributed DBMSs are typically characterized by computer clusters in one data center, and are commonly

known as parallel DBMSs (and the PEs are referred to as "nodes" to distinguish from "sites"). As noted above, it is now quite common to find distributed DBMSs that have multiple single site clusters interconnected by wide area networks, leading to hybrid, multisite systems. For most of this book, we will focus on the problems of data management among the sites of a geo-distributed DBMS; we will focus on the problems of single site systems in Chaps. 8, 10, and 11 where we discuss parallel DBMSs, big data systems, and NoSQL/NewSQL systems.

1.2 History of Distributed DBMS

Before the advent of database systems in the 1960s, the prevalent mode of computation was one where each application defined and maintained its own data (Fig. 1.2). In this mode, each application defined the data that it used, its structure and access methods, and managed the file in the storage system. The end result was significant uncontrolled redundancy in the data, and high overhead for the programmers to manage this data within their applications.

Database systems allow data to be defined and administered centrally (Fig. 1.3). This new orientation results in *data independence*, whereby the application programs are immune to changes in the logical or physical organization of the data and vice versa. Consequently, programmers are freed from the task of managing and maintaining the data that they need, and the redundancy of the data can be eliminated (or reduced).

One of the original motivations behind the use of database systems was the desire to integrate the operational data of an enterprise and to provide integrated, thus controlled access to that data. We carefully use the term "integrated" rather

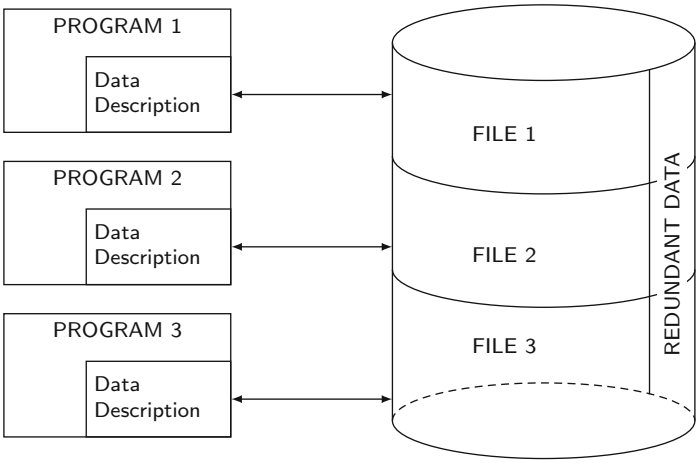

Fig. 1.2 Traditional file processing

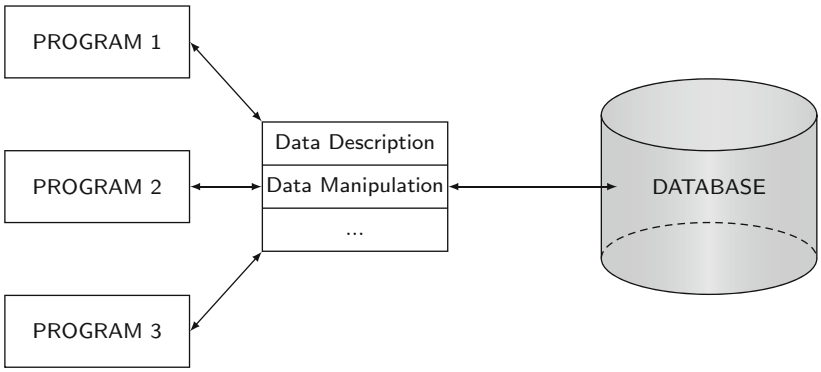

Fig. 1.3 Database processing

than "centralized" because, as discussed earlier, the data can physically be located on different machines that might be geographically distributed. This is what the distributed database technology provides. As noted earlier, this physical distribution can be concentrated at one geographic location or it can be at multiple locations. Therefore, each of the locations in Fig. 1.5 might be a data center that is connected to other data centers over a communication network. These are the types of distributed environments that are now common and that we study in this book.

Over the years, distributed database system architectures have undergone significant changes. The original distributed database systems such as Distributed INGRES and SDD-1 were designed as geographically distributed systems with very slow network connections; consequently they tried to optimize operations to reduce network communication. They were early *peer-to-peer systems* (P2P) in the sense that each site had similar functionality with respect to data management. With the development of personal computers and workstations, the prevailing distribution model shifted to *client/server* where data operations were moved to a back-end server, while user applications ran on the front-end workstations. These systems became dominant in particular for distribution at one particular location where the network speeds would be higher, enabling frequent communication between the clients and the server(s). There was a reemergence of P2P systems in the 2000s, where there is no distinction of client machines versus servers. These modern P2P systems have important differences from the earlier systems that we discuss later in this chapter. All of these architectures can still be found today and we discuss them in subsequent chapters.

The emergence of the World Wide Web (usually called the web) as a major collaboration and sharing platform had a profound impact on distributed data management research. Significantly more data was opened up for access, but this was not the well-structured, well-defined data DBMSs typically handle; instead, it is unstructured or semistructured (i.e., it has some structure but not at the level of a database schema), with uncertain provenance (so it might be "dirty" or unreliable), and conflicting. Furthermore, a lot of the data is stored in systems that are not easily

accessible (what is called the *dark web*). Consequently, distributed data management efforts focus on accessing this data in meaningful ways.

This development added particular impetus to one thread of research that existed since the beginning of distributed database efforts, namely *database integration*. Originally, these efforts focused on finding ways to access data in separate databases (thus the terms *federated database* and *multidatabase*), but with the emergence of web data, these efforts shifted to virtual integration of different data types (and the term *data integration* became more popular). The term that is in vogue right now is *data lake* which implies that all of the data is captured in a logically single store, from which relevant data is extracted for each application. We discuss the former in Chap. 7 and the latter in Chaps. 10 and 12.

A significant development in the last ten years has been the emergence of cloud computing. Cloud computing refers to a computing model where a number of service providers make available shared and geo-distributed computing resources such that users can rent some of these resources based on their needs. Clients can rent the basic computing infrastructure on which they could develop their own software, but then decide on the operating system they wish to use and create virtual machines (VMs) to create the environment in which they wish to work—the so-called *Infrastructure-as-a-Service* (IaaS) approach. A more sophisticated cloud environment involves renting, in addition to basic infrastructure, the full computing platform leading to *Platform-as-a-Service* (PaaS) on which clients can develop their own software. The most sophisticated version is where service providers make available specific software that the clients can then rent; this is called *Software-as-a-Service* (SaaS). There has been a trend in providing distributed database management services on the cloud as part of SaaS offering, and this has been one of the more recent developments.

In addition to the specific chapters where we discuss these architectures in depth, we provide an overview of all of them in Sect. 1.6.1.2.

1.3 Data Delivery Alternatives

In distributed databases, data delivery occurs between sites—either from server sites to client sites in answer to queries or between multiple servers. We characterize the data delivery alternatives along three orthogonal dimensions: *delivery modes*, *frequency*, and *communication methods*. The combinations of alternatives along each of these dimensions provide a rich design space.

The alternative delivery modes are pull-only, push-only, and hybrid. In the *pull-only* mode of data delivery, the transfer of data is initiated by a pull (i.e., request) from one site to a data provider—this may be a client requesting data from a server or a server requesting data from another server. In the following we use the terms "receiver" and "provider" to refer to the machine that received the data and the machine that sends the data, respectively. When the request is received by the provider, the data is located and transferred. The main characteristic of pull-

based delivery is that receivers become aware of new data items or updates at the provider only when they explicitly poll. Also, in pull-based mode, providers must be interrupted continuously to deal with requests. Furthermore, the data that receivers can obtain from a provider is limited to when and what clients know to ask for. Conventional DBMSs offer primarily pull-based data delivery.

In the *push-only* mode of data delivery, the transfer of data from providers is initiated by a push without a specific request. The main difficulty of the push-based approach is in deciding which data would be of common interest, and when to send it to potentially interested receivers—alternatives are periodic, irregular, or conditional. Thus, the usefulness of push depends heavily upon the accuracy of a provider to predict the needs of receivers. In push-based mode, providers disseminate information to either an unbounded set of receivers (random broadcast) who can listen to a medium or selective set of receivers (multicast), who belong to some categories of recipients.

The *hybrid* mode of data delivery combines the pull and push mechanisms. The persistent query approach (see Sect. 10.3) presents one possible way of combining the pull and push modes, namely: the transfer of data from providers to receivers is first initiated by a pull (by posing the query), and the subsequent transfer of updated data is initiated by a push by the provider.

There are three typical frequency measurements that can be used to classify the regularity of data delivery. They are periodic, conditional, and ad hoc (or irregular).

In *periodic delivery*, data is sent from the providers at regular intervals. The intervals can be defined by system default or by receivers in their profiles. Both pull and push can be performed in periodic fashion. Periodic delivery is carried out on a regular and prespecified repeating schedule. A request for a company's stock price every week is an example of a periodic pull. An example of periodic push is when an application can send out stock price listing on a regular basis, say every morning. Periodic push is particularly useful for situations in which receivers might not be available at all times, or might be unable to react to what has been sent, such as in the mobile setting where clients can become disconnected.

In *conditional delivery*, data is sent by providers whenever certain conditions specified by receivers in their profiles are satisfied. Such conditions can be as simple as a given time span or as complicated as event-condition-action rules. Conditional delivery is mostly used in the hybrid or push-only delivery systems. Using conditional push, data is sent out according to a prespecified condition, rather than any particular repeating schedule. An application that sends out stock prices only when they change is an example of conditional push. An application that sends out a balance statement only when the total balance is 5% below the predefined balance threshold is an example of hybrid conditional push. Conditional push assumes that changes are critical to the receivers who are always listening and need to respond to what is being sent. Hybrid conditional push further assumes that missing some update information is not crucial to the receivers.

Ad hoc delivery is irregular and is performed mostly in a pure pull-based system. Data is pulled from providers in an ad hoc fashion in response to requests. In

contrast, periodic pull arises when a requestor uses polling to obtain data from providers based on a regular period (schedule).

The third component of the design space of information delivery alternatives is the communication method. These methods determine the various ways in which providers and receivers communicate for delivering information to clients. The alternatives are unicast and one-to-many. In *unicast*, the communication from a provider to a receiver is one-to-one: the provider sends data to one receiver using a particular delivery mode with some frequency. In *one-to-many*, as the name implies, the provider sends data to a number of receivers. Note that we are not referring here to a specific protocol; one-to-many communication may use a multicast or broadcast protocol.

We should note that this characterization is subject to considerable debate. It is not clear that every point in the design space is meaningful. Furthermore, specification of alternatives such as conditional **and** periodic (which may make sense) is difficult. However, it serves as a first-order characterization of the complexity of emerging distributed data management systems. For the most part, in this book, we are concerned with pull-only, ad hoc data delivery systems, and discuss push-based and hybrid modes under streaming systems in Sect. 10.3.

1.4 Promises of Distributed DBMSs

Many advantages of distributed DBMSs can be cited; these can be distilled to four fundamentals that may also be viewed as promises of distributed DBMS technology: transparent management of distributed and replicated data, reliable access to data through distributed transactions, improved performance, and easier system expansion. In this section, we discuss these promises and, in the process, introduce many of the concepts that we will study in subsequent chapters.

1.4.1 Transparent Management of Distributed and Replicated Data

Transparency refers to separation of the higher-level semantics of a system from lower-level implementation issues. In other words, a transparent system "hides" the implementation details from users. The advantage of a fully transparent DBMS is the high level of support that it provides for the development of complex applications. Transparency in distributed DBMS can be viewed as an extension of the data independence concept in centralized DBMS (more on this below).

Let us start our discussion with an example. Consider an engineering firm that has offices in Boston, Waterloo, Paris, and San Francisco. They run projects at each of these sites and would like to maintain a database of their employees, the projects,

EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET, LOC)
ASG(ENO, PNO, RESP, DUR)
PAY(TITLE, SAL)

Fig. 1.4 Example engineering database

and other related data. Assuming that the database is relational, we can store this information in a number of relations (Fig. 1.4): EMP stores employee information with employee number, name, and title[1]; PROJ holds project information where LOC records where the project is located. The salary information is stored in PAY (assuming everyone with the same title gets the same salary) and the assignment of people to projects is recorded in ASG where DUR indicates the duration of the assignment and the person's responsibility on that project is maintained in RESP. If all of this data were stored in a centralized DBMS, and we wanted to find out the names and employees who worked on a project for more than 12 months, we would specify this using the following SQL query:

```
SELECT ENAME, AMT
FROM EMP NATURAL JOIN ASG, EMP NATURAL JOIN PAY
WHERE ASG.DUR > 12
```

However, given the distributed nature of this firm's business, it is preferable, under these circumstances, to localize data such that data about the employees in Waterloo office is stored in Waterloo, those in the Boston office is stored in Boston, and so forth. The same applies to the project and salary information. Thus, what we are engaged in is a process where we partition each of the relations and store each partition at a different site. This is known as *data partitioning* or *data fragmentation* and we discuss it further below and in detail in Chap. 2.

Furthermore, it may be preferable to duplicate some of this data at other sites for performance and reliability reasons. The result is a distributed database which is fragmented and replicated (Fig. 1.5). Fully transparent access means that the users can still pose the query as specified above, without paying any attention to the fragmentation, location, or replication of data, and let the system worry about resolving these issues. For a system to adequately deal with this type of query over a distributed, fragmented, and replicated database, it needs to be able to deal with a number of different types of transparencies as discussed below.

Data Independence. This notion carries over from centralized DBMSs and refers to the immunity of user applications to changes in the definition and organization of data, and vice versa.

Two types of data independence are usually cited: logical data independence and physical data independence. *Logical data independence* refers to the immunity of user applications to changes in the logical structure (i.e., schema) of the database.

[1] Primary key attributes are underlined.

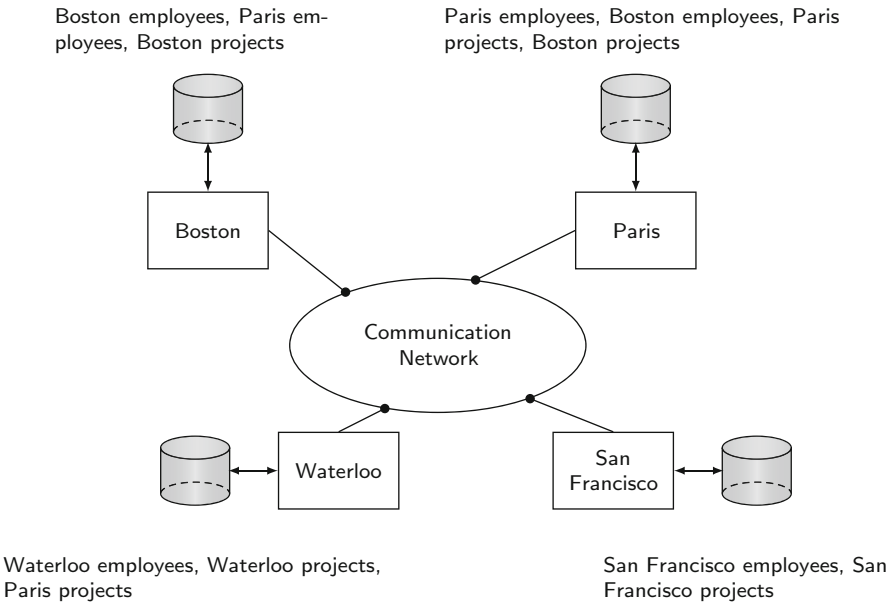

Fig. 1.5 Distributed database

Physical data independence, on the other hand, deals with hiding the details of the storage structure from user applications. When a user application is written, it should not be concerned with the details of physical data organization. Therefore, the user application should not need to be modified when data organization changes occur due to performance considerations.

Network Transparency. Preferably, users should be protected from the operational details of the communication network that connects the sites; possibly even hiding the existence of the network. Then there would be no difference between database applications that would run on a centralized database and those that would run on a distributed database. This type of transparency is referred to as *network transparency* or *distribution transparency*.

Sometimes two types of distribution transparency are identified: location transparency and naming transparency. *Location transparency* refers to the fact that the command used to perform a task is independent of both the location of the data and the system on which an operation is carried out. *Naming transparency* means that a unique name is provided for each object in the database. In the absence of naming transparency, users are required to embed the location name (or an identifier) as part of the object name.

Fragmentation Transparency. As discussed above, it is commonly desirable to divide each database relation into smaller fragments and treat each fragment as a separate database object (i.e., another relation). This is commonly done for reasons of performance, availability, and reliability—a more in-depth discussion

is in Chap. 2. It would be preferable for the users not to be aware of data fragmentation in specifying queries, and let the system deal with the problem of mapping a user query that is specified on full relations as specified in the schema to a set of queries executed on subrelations. In other words, the issue is one of finding a query processing strategy based on the fragments rather than the relations, even though the queries are specified on the latter.

Replication Transparency. For performance, reliability, and availability reasons, it is usually desirable to be able to distribute data in a replicated fashion across the machines on a network. Assuming that data is replicated, the transparency issue is whether the users should be aware of the existence of copies or whether the system should handle the management of copies and the user should act as if there is a single copy of the data (note that we are not referring to the placement of copies, only their existence). From a user's perspective it is preferable not to be involved with handling copies and having to specify the fact that a certain action can and/or should be taken on multiple copies. The issue of replicating data within a distributed database is introduced in Chap. 2 and discussed in detail in Chap. 6.

1.4.2 Reliability Through Distributed Transactions

Distributed DBMSs are intended to improve reliability since they have replicated components and thereby eliminate single points of failure. The failure of a single site, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system. In the case of a distributed database, this means that some of the data may be unreachable, but with proper care, users may be permitted to access other parts of the distributed database. The "proper care" comes mainly in the form of support for distributed transactions.

A DBMS that provides full transaction support guarantees that concurrent execution of user transactions will not violate database consistency, i.e., each user thinks their query is the only one executing on the database (called *concurrency transparency*) even in the face of system failures (called *failure transparency*) as long as each transaction is correct, i.e., obeys the integrity rules specified on the database.

Providing transaction support requires the implementation of distributed concurrency control and distributed reliability protocols—in particular, two-phase commit (2PC) and distributed recovery protocols—which are significantly more complicated than their centralized counterparts. These are discussed in Chap. 5. Supporting replicas requires the implementation of replica control protocols that enforce a specified semantics of accessing them. These are discussed in Chap. 6.

1.4.3 Improved Performance

The case for the improved performance of distributed DBMSs is typically made based on two points. First, a distributed DBMS fragments the database, enabling data to be stored in close proximity to its points of use (also called *data locality*). This has two potential advantages:

1. Since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases.
2. Locality reduces remote access delays that are usually involved in wide area networks.

This point relates to the overhead of distributed computing if the data resides at remote sites and one has to access it by remote communication. The argument is that it is better, in these circumstances, to distribute the data management functionality to where the data is located rather than moving large amounts of data. This is sometimes a topic of contention. Some argue that with the widespread use of high-speed, high-capacity networks, distributing data and data management functions no longer makes sense and that it may be much simpler to store data at a central site using a very large machine and access it over high-speed networks. This is commonly referred to as *scale-up* architecture. It is an appealing argument, but misses an important point of distributed databases. First, in most of today's applications, data is distributed; what may be open for debate is how and where we process it. Second, and more important, point is that this argument does not distinguish between bandwidth (the capacity of the computer links) and latency (how long it takes for data to be transmitted). Latency is inherent in distributed environments and there are physical limits to how fast we can send data over computer networks. Remotely accessing data may incur latencies that might not be acceptable for many applications.

The second point is that the inherent parallelism of distributed systems may be exploited for interquery and intraquery parallelism. *Interquery parallelism* enables the parallel execution of multiple queries generated by concurrent transactions, in order to increase the transactional throughput. The definition of intraquery parallelism is different in distributed versus parallel DBMSs. In the former, intraquery parallelism is achieved by breaking up a single query into a number of subqueries, each of which is executed at a different site, accessing a different part of the distributed database. In parallel DBMSs, it is achieved by *interoperator* and *intraoperator* parallelism. Interoperator parallelism is obtained by executing in parallel different operators of the query trie on different processors, while with intraoperator parallelism, the same operator is executed by many processors, each one working on a subset of the data. Note that these two forms of parallelism also exist in distributed query processing.

Intraoperator parallelism is based on the decomposition of one operator in a set of independent suboperators, called *operator instances*. This decomposition is done using partitioning of relations. Each operator instance will then process

one relation partition. The operator decomposition frequently benefits from the initial partitioning of the data (e.g., the data is partitioned on the join attribute). To illustrate intraoperator parallelism, let us consider a simple select-join query. The select operator can be directly decomposed into several select operators, each on a different partition, and no redistribution is required (Fig. 1.6). Note that if the relation is partitioned on the select attribute, partitioning properties can be used to eliminate some select instances. For example, in an exact-match select, only one select instance will be executed if the relation was partitioned by hashing (or range) on the select attribute. It is more complex to decompose the join operator. In order to have independent joins, each partition of one relation `R` may be joined to the entire other relation `S`. Such a join will be very inefficient (unless `S` is very small) because it will imply a broadcast of `S` on each participating processor. A more efficient way is to use partitioning properties. For example, if `R` and `S` are partitioned by hashing on the join attribute and if the join is an equijoin, then we can partition the join into independent joins. This is the ideal case that cannot be always used, because it depends on the initial partitioning of `R` and `S`. In the other cases, one or two operands may be repartitioned. Finally, we may notice that the partitioning function (hash, range, round robin—discussed in Sect. 2.3.1) is independent of the local algorithm (e.g., nested loop, hash, sort merge) used to process the join operator (i.e., on each processor). For instance, a hash join using a hash partitioning needs two hash functions. The first one, h_1, is used to partition the two base relations on the join attribute. The second one, h_2, which can be different for each processor, is used to process the join on each processor.

Two forms of interoperator parallelism can be exploited. With *pipeline parallelism*, several operators with a producer–consumer link are executed in parallel. For instance, the two select operators in Fig. 1.7 will be executed in parallel with the join operator. The advantage of such execution is that the intermediate result does not need to be entirely materialized, thus saving memory and disk accesses. *Independent*

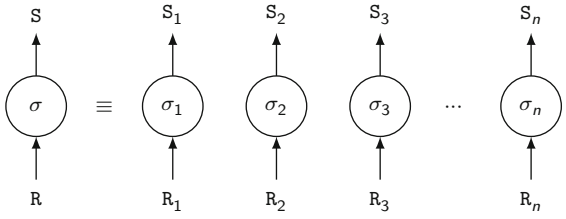

Fig. 1.6 Intraoperator parallelism. σ_i is instance i of the operator; n is the degree of parallelism

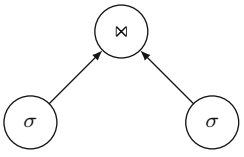

Fig. 1.7 Interoperator parallelism

parallelism is achieved when there is no dependency between the operators that are executed in parallel. For instance, the two select operators of Fig. 1.7 can be executed in parallel. This form of parallelism is very attractive because there is no interference between the processors.

1.4.4 Scalability

In a distributed environment, it is much easier to accommodate increasing database sizes and bigger workloads. System expansion can usually be handled by adding processing and storage power to the network. Obviously, it may not be possible to obtain a linear increase in "power," since this also depends on the overhead of distribution. However, significant improvements are still possible. That is why distributed DBMSs have gained much interest in *scale-out* architectures in the context of cluster and cloud computing. Scale-out (also called *horizontal scaling*) refers to adding more servers, called "scale-out servers" in a loosely coupled fashion, to scale almost infinitely. By making it easy to add new component database servers, a distributed DBMS can provide scale-out.

1.5 Design Issues

In the previous section, we discussed the promises of distributed DBMS technology, highlighting the challenges that need to be overcome in order to realize them. In this section, we build on this discussion by presenting the design issues that arise in building a distributed DBMS. These issues will occupy much of the remainder of this book.

1.5.1 Distributed Database Design

The question that is being addressed is how the data is placed across the sites. The starting point is one global database and the end result is a distribution of the data across the sites. This is referred to as *top-down design*. There are two basic alternatives to placing data: *partitioned* (or *nonreplicated*) and *replicated*. In the partitioned scheme the database is divided into a number of disjoint partitions each of which is placed at a different site. Replicated designs can be either *fully replicated* (also called *fully duplicated*) where the entire database is stored at each site, or *partially replicated* (or *partially duplicated*) where each partition of the database is stored at more than one site, but not at all the sites. The two fundamental design issues are *fragmentation*, the separation of the database into partitions called *fragments*, and *distribution*, the optimum distribution of fragments.

A related problem is the design and management of system directory. In centralized DBMSs, the *catalog* contains metainformation (i.e., description) about the data.

In a distributed system, we have a directory that contains additional information such as where data is located. Problems related to directory management are similar in nature to the database placement problem discussed in the preceding section. A directory may be global to the entire distributed DBMS or local to each site; it can be centralized at one site or distributed over several sites; there can be a single copy or multiple copies. Distributed database design and directory management are topics of Chap. 2.

1.5.2 Distributed Data Control

An important requirement of a DBMS is to maintain data consistency by controlling how data is accessed. This is called *data control* and involves view management, access control, and integrity enforcement. Distribution imposes additional challenges since data that is required to check rules is distributed to different sites requiring distributed rule checking and enforcement. The topic is covered in Chap. 3.

1.5.3 Distributed Query Processing

Query processing deals with designing algorithms that analyze queries and convert them into a series of data manipulation operations. The problem is how to decide on a strategy for executing each query over the network in the most cost-effective way, however, cost is defined. The factors to be considered are the distribution of data, communication costs, and lack of sufficient locally available information. The objective is to optimize where the inherent parallelism is used to improve the performance of executing the transaction, subject to the above-mentioned constraints. The problem is NP-hard in nature, and the approaches are usually heuristic. Distributed query processing is discussed in detail in Chap. 4.

1.5.4 Distributed Concurrency Control

Concurrency control involves the synchronization of accesses to the distributed database, such that the integrity of the database is maintained. The concurrency control problem in a distributed context is somewhat different than in a centralized framework. One not only has to worry about the integrity of a single database, but also about the consistency of multiple copies of the database. The condition that requires all the values of multiple copies of every data item to converge to the same value is called *mutual consistency*.

The two general classes of solutions are *pessimistic*, synchronizing the execution of user requests before the execution starts, and *optimistic*, executing the requests and then checking if the execution has compromised the consistency of the database. Two fundamental primitives that can be used with both approaches are *locking*, which is based on the mutual exclusion of accesses to data items, and *timestamping*, where the transaction executions are ordered based on timestamps. There are variations of these schemes as well as hybrid algorithms that attempt to combine the two basic mechanisms.

In locking-based approaches deadlocks are possible since there is mutually exclusive access to data by different transactions. The well-known alternatives of prevention, avoidance, and detection/recovery also apply to distributed DBMSs. Distributed concurrency control is covered in Chap. 5.

1.5.5 Reliability of Distributed DBMS

We mentioned earlier that one of the potential advantages of distributed systems is improved reliability and availability. This, however, is not a feature that comes automatically. It is important that mechanisms be provided to ensure the consistency of the database as well as to detect failures and recover from them. The implication for distributed DBMSs is that when a failure occurs and various sites become either inoperable or inaccessible, the databases at the operational sites remain consistent and up-to-date. Furthermore, when the computer system or network recovers from the failure, the distributed DBMSs should be able to recover and bring the databases at the failed sites up-to-date. This may be especially difficult in the case of network partitioning, where the sites are divided into two or more groups with no communication among them. Distributed reliability protocols are the topic of Chap. 5.

1.5.6 Replication

If the distributed database is (partially or fully) replicated, it is necessary to implement protocols that ensure the consistency of the replicas, i.e., copies of the same data item have the same value. These protocols can be *eager* in that they force the updates to be applied to all the replicas before the transaction completes, or they may be *lazy* so that the transaction updates one copy (called the *master*) from which updates are propagated to the others after the transaction completes. We discuss replication protocols in Chap. 6.

1.5.7 Parallel DBMSs

As earlier noted, there is a strong relationship between distributed databases and parallel databases. Although the former assumes each site to be a single logical computer, most of these installations are, in fact, parallel clusters. This is the distinction that we highlighted earlier between single site distribution as in data center clusters and geo-distribution. Parallel DBMS objectives are somewhat different from distributed DBMSs in that the main objectives are high scalability and performance. While most of the book focuses on issues that arise in managing data in geo-distributed databases, interesting data management issues exist within a single site distribution as a parallel system. We discuss these issues in Chap. 8.

1.5.8 Database Integration

One of the important developments has been the move towards "looser" federation among data sources, which may also be heterogeneous. As we discuss in the next section, this has given rise to the development of multidatabase systems (also called *federated database systems*) that require reinvestigation of some of the fundamental database techniques. The input here is a set of already distributed databases and the objective is to provide easy access by (physically or logically) integrating them. This involves *bottom-up design*. These systems constitute an important part of today's distributed environment. We discuss multidatabase systems, or as more commonly termed now *database integration*, including design issues and query processing challenges in Chap. 7.

1.5.9 Alternative Distribution Approaches

The growth of the Internet as a fundamental networking platform has raised important questions about the assumptions underlying distributed database systems. Two issues are of particular concern to us. One is the re-emergence of peer-to-peer computing, and the other is the development and growth of the World Wide Web. Both of these aim at improving data sharing, but take different approaches and pose different data management challenges. We discuss peer-to-peer data management in Chap. 9 and web data management in Chap. 12.

1.5.10 Big Data Processing and NoSQL

The last decade has seen the explosion of "big data" processing. The exact definition of big data is elusive, but they are typically accepted to have four characteristics dubbed the "four V's": data is very high *volume*, is multimodal (*variety*), usually

comes at very high speed as data streams (*velocity*), and may have quality concerns due to uncertain sources and conflicts (*veracity*). There have been significant efforts to develop systems to deal with "big data," all spurred by the perceived unsuitability of relational DBMSs for a number of new applications. These efforts typically take two forms: one thread has developed general purpose computing platforms (almost always scale-out) for processing, and the other special DBMSs that do not have the full relational functionality, with more flexible data management capabilities (the so-called NoSQL systems). We discuss the big data platforms in Chap. 10 and NoSQL systems in Chap. 11.

1.6 Distributed DBMS Architectures

The architecture of a system defines its structure. This means that the components of the system are identified, the function of each component is specified, and the interrelationships and interactions among these components are defined. The specification of the architecture of a system requires identification of the various modules, with their interfaces and interrelationships, in terms of the data and control flow through the system.

In this section, we develop four "reference" architectures[2] for a distributed DBMS: client/server, peer-to-peer, multidatabase, and cloud. These are "idealized" views of a DBMS in that many of the commercially available systems may deviate from them; however, the architectures will serve as a reasonable framework within which the issues related to distributed DBMS can be discussed.

We start with a discussion of the design space to better position the architectures that will be presented.

1.6.1 Architectural Models for Distributed DBMSs

We use a classification (Fig. 1.8) that recognizes three dimensions according to which distributed DBMSs may be architected: (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity. These dimensions are orthogonal as we discuss shortly and in each dimension we identify a number of alternatives. Consequently, there are 18 possible architectures in the design space; not all of these architectural alternatives are meaningful, and most are not relevant from the perspective of this book. The three on which we focus are identified in Fig. 1.8.

[2] A reference architecture is commonly created by standards developers to clearly define the interfaces that need to be standardized.

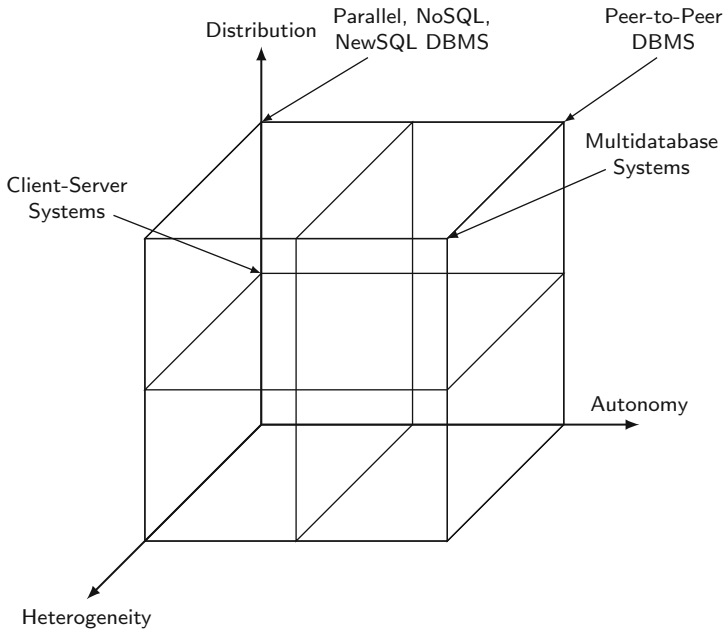

Fig. 1.8 DBMS implementation alternatives

1.6.1.1 Autonomy

Autonomy, in this context, refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems (i.e., individual DBMSs) exchange information, whether they can independently execute transactions, and whether one is allowed to modify them.

We will use a classification that covers the important aspects of these features. This classification highlights three alternatives. One alternative is *tight integration*, where a single-image of the entire database is available to any user who wants to share the data that may reside in multiple databases. From the users' perspective, the data is logically integrated in one database. In these tightly integrated systems, the data managers are implemented so that one of them is in control of the processing of each user request even if that request is serviced by more than one data manager. The data managers do not typically operate as independent DBMSs even though they usually have the functionality to do so.

Next, we identify *semiautonomous* systems that consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determines what parts of their own database they will make accessible to users of other DBMSs. They are not fully

autonomous systems because they need to be modified to enable them to exchange information with one another.

The last alternative that we consider is *total isolation*, where the individual systems are stand-alone DBMSs that know neither of the existence of other DBMSs nor how to communicate with them. In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

1.6.1.2 Distribution

Whereas autonomy refers to the distribution (or decentralization) of control, the distribution dimension of the taxonomy deals with data. Of course, we are considering the physical distribution of data over multiple sites; as we discussed earlier, the user sees the data as one logical pool. There are a number of ways DBMSs have been distributed. We abstract these alternatives into two classes: *client/server* distribution and *peer-to-peer* distribution (or *full* distribution). Together with the nondistributed option, the taxonomy identifies three alternative architectures.

The client/server distribution concentrates data management duties at servers, while the clients focus on providing the application environment including the user interface. The communication duties are shared between the client machines and servers. Client/server DBMSs represent a practical compromise to distributing functionality. There are a variety of ways of structuring them, each providing a different level of distribution. We leave detailed discussion to Sect. 1.6.2.

In *peer-to-peer systems*, there is no distinction of client machines versus servers. Each machine has full DBMS functionality and can communicate with other machines to execute queries and transactions. Most of the very early work on distributed database systems have assumed peer-to-peer architecture. Therefore, our main focus in this book is on peer-to-peer systems (also called *fully distributed*), even though many of the techniques carry over to client/server systems as well.

1.6.1.3 Heterogeneity

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of this book relate to data models, query languages, and transaction management protocols. Representing data with different modeling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models. Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in some object-oriented systems), but also covers differences in languages even when the individual systems use the same data model. Although SQL is now the standard relational query language, there are many

different implementations and every vendor's language has a slightly different flavor (sometimes even different semantics, producing different results). Furthermore, big data platforms and NoSQL systems have significantly variable access languages and mechanisms.

1.6.2 Client/Server Systems

Client/server entered the computing scene at the beginning of 1990s and has made a significant impact on the DBMS technology. The general idea is very simple and elegant: distinguish the functionality that needs to be provided on a server machine from those that need to be provided on a client. This provides a *two-level architecture* which makes it easier to manage the complexity of modern DBMSs and the complexity of distribution.

In relational client/server DBMSs, the server does most of the data management work. This means that all of query processing and optimization, transaction management, and storage management are done at the server. The client, in addition to the application and the user interface, has a *DBMS client* module that is responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well. It is also possible to place consistency checking of user queries at the client side, but this is not common since it requires the replication of the system catalog at the client machines. This architecture, depicted in Fig. 1.9, is quite common in relational systems where the communication between the clients and the server(s) is at the level of SQL

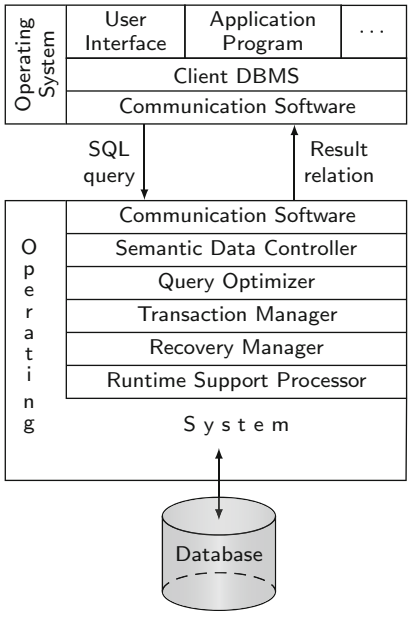

Fig. 1.9 Client/server reference architecture

statements. In other words, the client passes SQL queries to the server without trying to understand or optimize them. The server does most of the work and returns the result relation to the client.

There are a number of different realizations of the client/server architecture. The simplest is the case where there is only one server which is accessed by multiple clients. We call this *multiple client/single server*. From a data management perspective, this is not much different from centralized databases since the database is stored on only one machine (the server) that also hosts the software to manage it. However, there are important differences from centralized systems in the way transactions are executed and caches are managed—since data is cached at the client, it is necessary to deploy cache coherence protocols.

A more sophisticated client/server architecture is one where there are multiple servers in the system (the so-called *multiple client/multiple server* approach). In this case, two alternative management strategies are possible: either each client manages its own connection to the appropriate server or each client knows of only its "home server" which then communicates with other servers as required. The former approach simplifies server code, but loads the client machines with additional responsibilities. This leads to what has been called "heavy client" systems. The latter approach, on the other hand, concentrates the data management functionality at the servers. Thus, the transparency of data access is provided at the server interface, leading to "light clients."

In the multiple server systems, data is partitioned and may be replicated across the servers. This is transparent to the clients in the case of light client approach, and servers may communicate among themselves to answer a user query. This approach is implemented in parallel DBMS to improve performance through parallel processing.

Client/server can be naturally extended to provide for a more efficient function distribution on different kinds of servers: *clients* run the user interface (e.g., web servers), *application servers* run application programs, and *database servers* run database management functions. This leads to the three-tier distributed system architecture.

The application server approach (indeed, an n-tier distributed approach) can be extended by the introduction of multiple database servers and multiple application servers (Fig. 1.10), as can be done in classical client/server architectures. In this case, it is typically the case that each application server is dedicated to one or a few applications, while database servers operate in the multiple server fashion discussed above. Furthermore, the interface to the application is typically through a load balancer that routes the client requests to the appropriate servers.

The database server approach, as an extension of the classical client/server architecture, has several potential advantages. First, the single focus on data management makes possible the development of specific techniques for increasing data reliability and availability, e.g., using parallelism. Second, the overall performance of database management can be significantly enhanced by the tight integration of the database system and a dedicated database operating system. Finally, database servers can also exploit advanced hardware assists such as GPUs and FPGAs to enhance both performance and data availability.

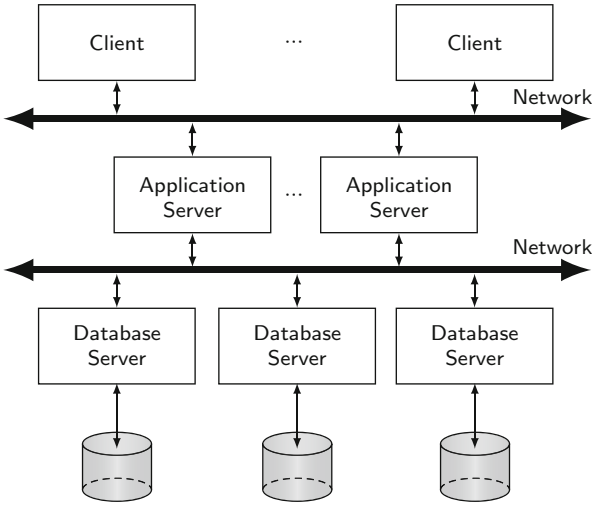

Fig. 1.10 Distributed database servers

Although these advantages are significant, there is the additional overhead introduced by another layer of communication between the application and the data servers. The communication cost can be amortized if the server interface is sufficiently high level to allow the expression of complex queries involving intensive data processing.

1.6.3 Peer-to-Peer Systems

The early works on distributed DBMSs all focused on peer-to-peer architectures where there was no differentiation between the functionality of each site in the system. Modern peer-to-peer systems have two important differences from their earlier relatives. The first is the massive distribution in more recent systems. While in the early days the focus was on a few (perhaps at most tens of) sites, current systems consider thousands of sites. The second is the inherent heterogeneity of every aspect of the sites and their autonomy. While this has always been a concern of distributed databases, as discussed earlier, coupled with massive distribution, site heterogeneity and autonomy take on an added significance, disallowing some of the approaches from consideration. In this book we initially focus on the classical meaning of peer-to-peer (the same functionality at each site), since the principles and fundamental techniques of these systems are very similar to those of client/server systems, and discuss the modern peer-to-peer database issues in a separate chapter (Chap. 9).

In these systems, the database design follows a top-down design as discussed earlier. So, the input is a (centralized) database with its own schema definition (*global conceptual schema*—GCS). This database is partitioned and allocated to sites of the distributed DBMS. Thus, at each site, there is a local database with its own schema (called the *local conceptual schema*—LCS). The user formulates queries according to the GCS, irrespective of its location. The distributed DBMS translates global queries into a group of local queries, which are executed by distributed DBMS components at different sites that communicate with one another. From a querying perspective, peer-to-peer systems and client/server DBMSs provide the same view of data. That is, they give the user the appearance of a logically single database, while at the physical level data is distributed.

The detailed components of a distributed DBMS are shown in Fig. 1.11. One component handles the interaction with users, and another deals with the storage.

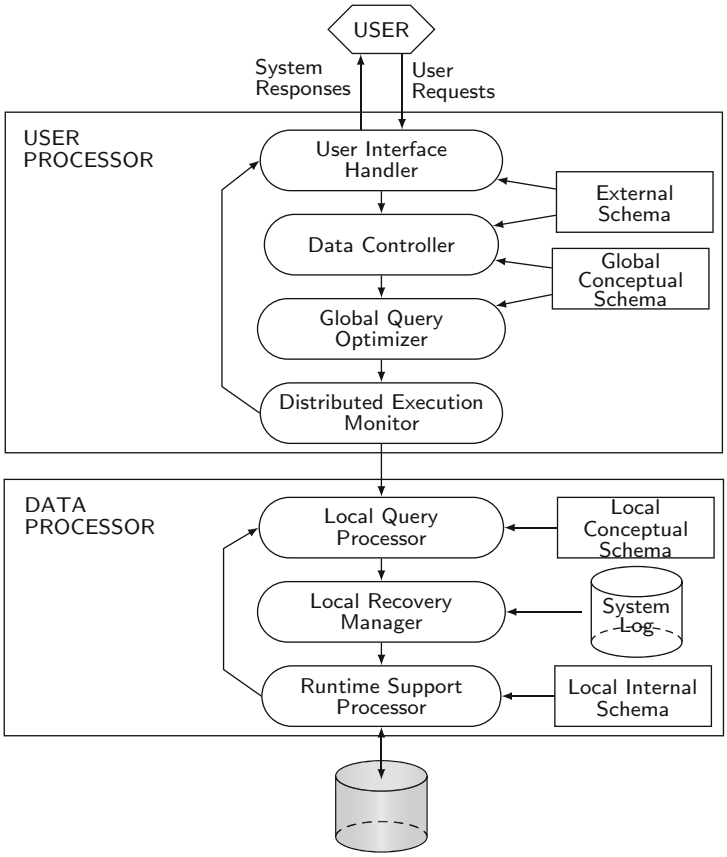

Fig. 1.11 Components of a distributed DBMS

The first major component, which we call the *user processor*, consists of four elements:

1. The *user interface handler* is responsible for interpreting user commands as they come in, and formatting the result data as it is sent to the user.
2. The *data controller* uses the integrity constraints and authorizations that are defined as part of the global conceptual schema to check if the user query can be processed. This component, which is studied in detail in Chap. 3, is also responsible for authorization and other functions.
3. The *global query optimizer and decomposer* determines an execution strategy to minimize a cost function, and translates the global queries into local ones using the global and local conceptual schemas as well as the global directory. The global query optimizer is responsible, among other things, for generating the best strategy to execute distributed join operations. These issues are discussed in Chap. 4.
4. The *distributed execution monitor* coordinates the distributed execution of the user request. The execution monitor is also called the *distributed transaction manager*. In executing queries in a distributed fashion, the execution monitors at various sites may, and usually do, communicate with one another. Distributed transaction manager functionality is covered in Chap. 5.

The second major component of a distributed DBMS is the *data processor* and consists of the following three elements. These are all issues that centralized DBMSs deal with, so we do not focus on them in this book.

1. The *local query optimizer*, which actually acts as the *access path selector*, is responsible for choosing the best access path[3] to access any data item.
2. The *local recovery manager* is responsible for making sure that the local database remains consistent even when failures occur.
3. The *runtime support processor* physically accesses the database according to the physical commands in the schedule generated by the query optimizer. The runtime support processor is the interface to the operating system and contains the *database buffer* (or *cache*) *manager*, which is responsible for maintaining the main memory buffers and managing the data accesses.

It is important to note that our use of the terms "user processor" and "data processor" does not imply a functional division similar to client/server systems. These divisions are merely organizational and there is no suggestion that they should be placed on different machines. In peer-to-peer systems, one expects to find both the user processor modules and the data processor modules on each machine. However, there can be "query-only sites" that only have the user processor.

[3]The term *access path* refers to the data structures and the algorithms that are used to access the data. A typical access path, for example, is an index on one or more attributes of a relation.

1.6.4 Multidatabase Systems

Multidatabase systems (MDBSs) represent the case where individual DBMSs are fully autonomous and have no concept of cooperation; they may not even "know" of each other's existence or how to talk to each other. Our focus is, naturally, on distributed MDBSs, which refers to the MDBS where participating DBMSs are located on different sites. Many of the issues that we discussed are common to both single-node and distributed MDBSs; in those cases we will simply use the term MDBS without qualifying it as single node or distributed. In most current literature, one finds the term *database integration* used instead. We discuss these systems further in Chap. 7. We note, however, that there is considerable variability in the use of the term "multidatabase" in literature. In this book, we use it consistently as defined above, which may deviate from its use in some of the existing literature.

The differences in the level of autonomy between the MDBSs and distributed DBMSs are also reflected in their architectural models. The fundamental difference relates to the definition of the global conceptual schema. In the case of logically integrated distributed DBMSs, the global conceptual schema defines the conceptual view of the *entire* database, while in the case of MDBSs, it represents only the collection of *some* of the local databases that each local DBMS wants to share. The individual DBMSs may choose to make some of their data available for access by others. Thus the definition of a *global database* is different in MDBSs than in distributed DBMSs. In the latter, the global database is equal to the union of local databases, whereas in the former it is only a (possibly proper) subset of the same union. In an MDBS, the GCS (which is also called a *mediated schema*) is defined by integrating (possibly parts of) local conceptual schemas.

The component-based architectural model of a distributed MDBS is significantly different from a distributed DBMS, because each site is a full-fledged DBMS that manages a different database. The MDBS provides a layer of software that runs on top of these individual DBMSs and provides users with the facilities of accessing various databases (Fig. 1.12). Note that in a distributed MDBS, the MDBS layer may run on multiple sites or there may be central site where those services are offered. Also note that as far as the individual DBMSs are concerned, the MDBS layer is simply another application that submits requests and receives answers.

A popular implementation architecture for MDBSs is the mediator/wrapper approach (Fig. 1.13). A *mediator* "is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications" [Wiederhold 1992]. Thus, each mediator performs a particular function with clearly defined interfaces. Using this architecture to implement an MDBS, each module in the MDBS layer of Fig. 1.12 is realized as a mediator. Since mediators can be built on top of other mediators, it is possible to construct a layered implementation. The mediator level implements the GCS. It is this level that handles user queries over the GCS and performs the MDBS functionality.

The mediators typically operate using a common data model and interface language. To deal with potential heterogeneities of the source DBMSs, *wrappers*

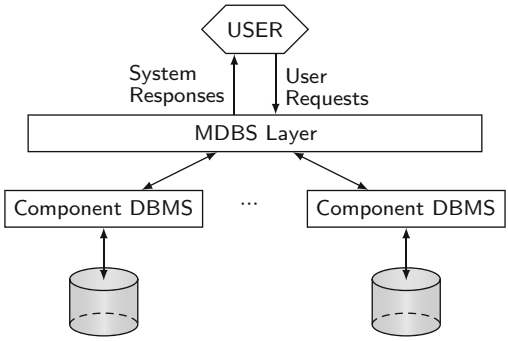

Fig. 1.12 Components of an MDBS

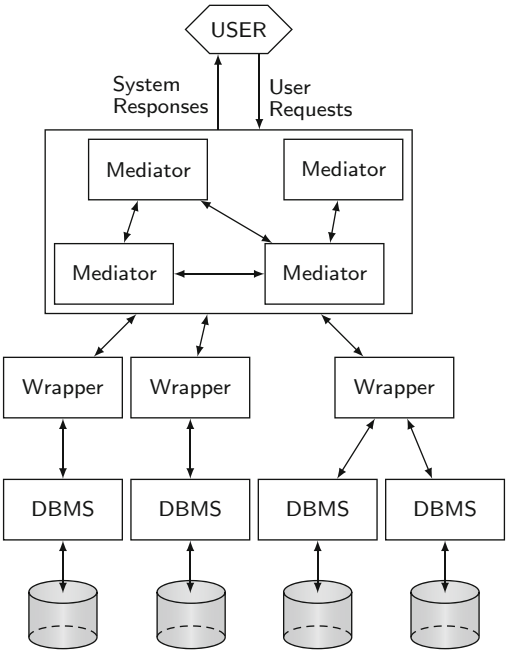

Fig. 1.13 Mediator/wrapper architecture

are implemented whose task is to provide a mapping between a source DBMSs view and the mediators' view. For example, if the source DBMS is a relational one, but the mediator implementations are object-oriented, the required mappings are established by the wrappers. The exact role and function of mediators differ from one implementation to another. In some cases, mediators do nothing more than translation; these are called "thin" mediators. In other cases, wrappers take over the execution of some of the query functionality.

One can view the collection of mediators as a middleware layer that provides services above the source systems. Middleware is a topic that has been the subject of significant study in the past decade and very sophisticated middleware systems have been developed that provide advanced services for development of distributed applications. The mediators that we discuss only represent a subset of the functionality provided by these systems.

1.6.5 Cloud Computing

Cloud computing has caused a significant shift in how users and organizations deploy scalable applications, in particular, data management applications. The vision encompasses on demand, reliable services provided over the Internet (typically represented as a cloud) with easy access to virtually infinite computing, storage, and networking resources. Through very simple web interfaces and at small incremental cost, users can outsource complex tasks, such as data storage, database management, system administration, or application deployment, to very large data centers operated by cloud providers. Thus, the complexity of managing the software/hardware infrastructure gets shifted from the users' organization to the cloud provider.

Cloud computing is a natural evolution, and combination, of different computing models proposed for supporting applications over the web: service-oriented architectures (SOA) for high-level communication of applications through web services, utility computing for packaging computing and storage resources as services, cluster and virtualization technologies to manage lots of computing and storage resources, and autonomous computing to enable self-management of complex infrastructure. The cloud provides various levels of functionality such as:

- Infrastructure-as-a-Service (IaaS): the delivery of a computing infrastructure (i.e., computing, networking, and storage resources) as a service;
- Platform-as-a-Service (PaaS): the delivery of a computing platform with development tools and APIs as a service;
- Software-as-a-Service (SaaS): the delivery of application software as a service; or
- Database-as-a-Service (DaaS): the delivery of database as a service.

What makes cloud computing unique is its ability to provide and combine all kinds of services to best fit the users' requirements. From a technical point of view, the grand challenge is to support in a cost-effective way, the very large scale of the infrastructure that has to manage lots of users and resources with high quality of service.

Agreeing on a precise definition of cloud computing is difficult as there are many different perspectives (business, market, technical, research, etc.). However, a good working definition is that a "cloud provides on demand resources and services over the Internet, usually at the scale and with the reliability of a data center" [Grossman and Gu 2009]. This definition captures well the main objective (providing on-

demand resources and services over the Internet) and the main requirements for supporting them (at the scale and with the reliability of a data center). Since the resources are accessed through services, everything gets delivered as a service. Thus, as in the services industry, this enables cloud providers to propose a pay-as-you-go pricing model, whereby users only pay for the resources they consume.

The main functions provided by clouds are: security, directory management, resource management (provisioning, allocation, monitoring), and data management (storage, file management, database management, data replication). In addition, clouds provide support for pricing, accounting, and service level agreement management.

The typical advantages of cloud computing are the following:

- **Cost.** The cost for the customer can be greatly reduced since the infrastructure does not need to be owned and managed; billing is only based on resource consumption. As for the cloud provider, using a consolidated infrastructure and sharing costs for multiple customers reduces the cost of ownership and operation.
- **Ease of access and use.** The cloud hides the complexity of the IT infrastructure and makes location and distribution transparent. Thus, customers can have access to IT services anytime, and from anywhere with an Internet connection.
- **Quality of service.** The operation of the IT infrastructure by a specialized provider that has extensive experience in running very large infrastructures (including its own infrastructure) increases quality of service and operational efficiency.
- **Innovation.** Using state-of-the-art tools and applications provided by the cloud encourages modern practice, thus increasing the innovation capabilities of the customers.
- **Elasticity.** The ability to scale resources out, up and down dynamically to accommodate changing conditions is a major advantage. This is typically achieved through server virtualization, a technology that enables multiple applications to run on the same physical computer as virtual machines (VMs), i.e., as if they would run on distinct physical computers. Customers can then require computing instances as VMs and attach storage resources as needed.

However, there are also disadvantages that must be well-understood before moving to the cloud. These disadvantages are similar to when outsourcing applications and data to an external company.

- **Provider dependency.** Cloud providers tend to lock in customers, through proprietary software, proprietary format, or high outbound data transfer costs, thus making cloud service migration difficult.
- **Loss of control.** Customers may lose managerial control over critical operations such as system downtime, e.g., to perform a software upgrade.
- **Security.** Since a customer's cloud data is accessible from anywhere on the Internet, security attacks can compromise business's data. Cloud security can be improved using advanced capabilities, e.g., virtual private cloud, but may be complex to integrate with a company's security policy.

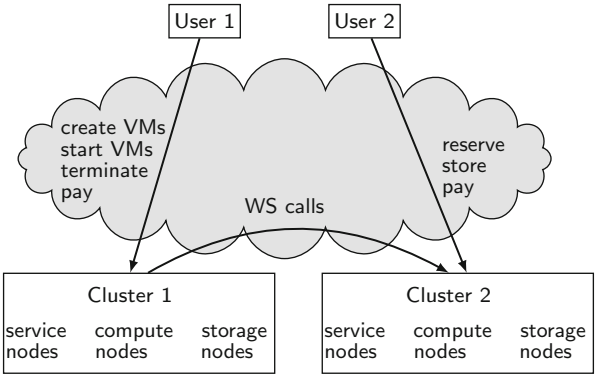

Fig. 1.14 Simplified cloud architecture

- **Hidden costs.** Customizing applications to make them cloud-ready using SaaS/-PaaS may incur significant development costs.

There is no standard cloud architecture and there will probably never be one, since different cloud providers provide different cloud services (IaaS, PaaS, SaaS, etc.) in different ways (public, private, virtual private, etc.) depending on their business models. Thus, in this section, we discuss a simplified cloud architecture with emphasis on database management.

A cloud is typically multisite (Fig. 1.14), i.e., made of several geographically distributed sites (or data centers), each with its own resources and data. Major cloud providers divide the world in several regions, each with several sites. There are three major reasons for this. First, there is low latency access in a user's region since user requests can be directed to the closest site. Second, using data replication across sites in different regions provides high availability, in particular, resistance from catastrophic (site) failures. Third, some national regulations that protect citizen's data privacy force cloud providers to locate data centers in their region (e.g., Europe). Multisite transparency is generally a default option, so the cloud appears "centralized" and the cloud provider can optimize resource allocation to users. However, some cloud providers (e.g., Amazon and Microsoft) make their sites visible to users (or application developers). This allows choosing a particular data center to install an application with its database, or to deploy a very large application across multiple sites communicating through web services (WS). For instance, in Fig. 1.14, we could imagine that Client 1 first connects to an application at Data Center 1, which would call an application at Data Center 2 using WS.

The architecture of a cloud site (data center) is typically 3-tier. The first tier consists of web clients that access cloud web servers, typically via a router or load balancer at the cloud site. The second tier consists of web/application servers that support the clients and provide business logic. The third tier consists of database servers. There can be other kinds of servers, e.g., cache servers between the application servers and database servers. Thus, the cloud architecture provides two

levels of distribution: geographical distribution across sites using a WAN and within a site, distribution across the servers, typically in a computer cluster. The techniques used at the first level are those of geographically distributed DBMS, while the techniques used at the second level are those of parallel DBMS.

Cloud computing has been originally designed by web giants to run their very large scale applications on data centers with thousands of servers. Big data systems (Chap. 10) and NoSQL/NewSQL systems (Chap. 11) specifically address the requirements of such applications in the cloud, using distributed data management techniques. With the advent of SaaS and PaaS solutions, cloud providers also need to serve small applications for very high numbers of customers, called *tenants*, each with its own (small) database accessed by its users. Dedicating a server for each tenant is wasteful in terms of hardware resources. To reduce resource wasting and operation cost, cloud providers typically share resources among tenants using a "multitenancy" architecture in which a single server can accommodate multiple tenants. Different multitenant models yield different trade-offs between performance, isolation (both security and performance isolation), and design complexity. A straightforward model used in IaaS is hardware sharing, which is typically achieved through server virtualization, with a VM for each tenant database and operating system. This model provides strong security isolation. However, resource utilization is limited because of redundant DBMS instances (one per VM) that do not cooperate and perform independent resource management. In the context of SaaS, PaaS, or DaaS, we can distinguish three main multitenant database models with increasing resource sharing and performance at the expense of less isolation and increased complexity.

- **Shared DBMS server.** In this model, tenants share a server with one DBMS instance, but each tenant has a different database. Most DBMSs provide support for multiple databases in a single DBMS instance. Thus, this model can be easily supported using a DBMS. It provides strong isolation at the database level and is more efficient than shared hardware as the DBMS instance has full control over hardware resources. However, managing each of these databases separately may still lead to inefficient resource management.
- **Shared database.** In this model, tenants share a database, but each tenant has its own schema and tables. Database consolidation is typically provided by an additional abstraction layer in the DBMS. This model is implemented by some DBMS (e.g., Oracle) using a single container database hosting multiple databases. It provides good resource usage and isolation at schema level. However, with lots (thousands) of tenants per server, there is a high number of small tables, which induces much overhead.
- **Shared tables.** In this model, tenants share a database, schema, and tables. To distinguish the rows of different tenants in a table, there is usually an additional column tenant_id. Although there is better resource sharing (e.g., cache memory), there is less isolation, both in security and performance. For instance, bigger customers will have more rows in shared tables, thus hurting the performance for smaller customers.

1.7 Bibliographic Notes

There are not many books on distributed DBMSs. The two early ones by Ceri and Pelagatti [1983] and Bell and Grimson [1992] are now out of print. A more recent book by Rahimi and Haug [2010] covers some of the classical topics that are also covered in this book. In addition, almost every database book now has a chapter on distributed DBMSs.

The pioneering systems Distributed INGRES and SDD-1 are discussed in [Stonebraker and Neuhold 1977] and [Wong 1977], respectively.

Database design is discussed in an introductory manner in [Levin and Morgan 1975] and more comprehensively in [Ceri et al. 1987]. A survey of the file distribution algorithms is given in [Dowdy and Foster 1982]. Directory management has not been considered in detail in the research community, but general techniques can be found in [Chu and Nahouraii 1975] and [Chu 1976]. A survey of query processing techniques can be found in [Sacco and Yao 1982]. Concurrency control algorithms are reviewed in [Bernstein and Goodman 1981] and [Bernstein et al. 1987]. Deadlock management has also been the subject of extensive research; an introductory paper is [Isloor and Marsland 1980] and a widely quoted paper is [Obermack 1982]. For deadlock detection, good surveys are [Knapp 1987] and [Elmagarmid 1986]. Reliability is one of the issues discussed in [Gray 1979], which is one of the landmark papers in the field. Other important papers on this topic are [Verhofstadt 1978] and [Härder and Reuter 1983]. [Gray 1979] is also the first paper discussing the issues of operating system support for distributed databases; the same topic is addressed in [Stonebraker 1981]. Unfortunately, both papers emphasize centralized database systems. A very good early survey of multidatabase systems is by Sheth and Larson [1990]; Wiederhold [1992] proposes the mediator/wrapper approach to MDBSs. Cloud computing has been the topic of quite a number of recent books; perhaps [Agrawal et al. 2012] is a good starting point and [Cusumano 2010] is a good short overview. The architecture we used in Sect. 1.6.5 is from [Agrawal et al. 2012]. Different multitenant models in cloud environments are discussed in [Curino et al. 2011] and [Agrawal et al. 2012].

There have been a number of architectural framework proposals. Some of the interesting ones include Schreiber's quite detailed extension of the ANSI/SPARC framework which attempts to accommodate heterogeneity of the data models [Schreiber 1977], and the proposal by Mohan and Yeh [1978]. As expected, these date back to the early days of the introduction of distributed DBMS technology. The detailed component-wise system architecture given in Fig. 1.11 derives from [Rahimi 1987]. An alternative to the classification that we provide in Fig. 1.8 can be found in [Sheth and Larson 1990].

The book by Agrawal et al. [2012] gives a very good presentation of the challenges and concepts of data management in the cloud, including distributed transactions, big data systems, and multitenant databases.

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

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

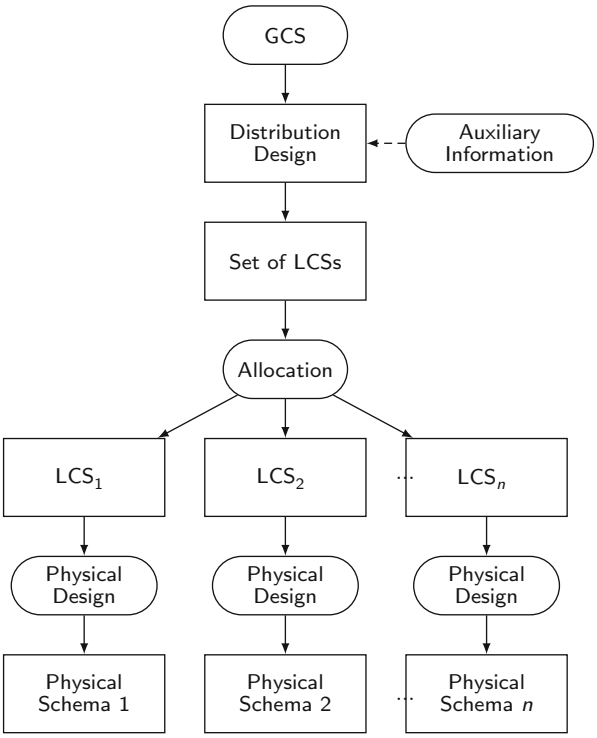

Fig. 2.1 Distribution design process

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: PROJ_1 contains information about projects whose budgets are less than $200,000, whereas PROJ_2 stores information about projects with larger budgets. ♦

$\texttt{PROJ}_1$

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

$\texttt{PROJ}_2$

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

Fig. 2.3 Example of horizontal partitioning

$\texttt{PROJ}_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

$\texttt{PROJ}_2$

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: $\texttt{PROJ}_1$ and $\texttt{PROJ}_2$. $\texttt{PROJ}_1$ contains only the information about project budgets, whereas $\texttt{PROJ}_2$ 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 $F_{\texttt{R}} = \{\texttt{R}_1, \texttt{R}_2, \ldots, \texttt{R}_n\}$, each data item that is in R can also be found in one or more of $\texttt{R}_i$'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.

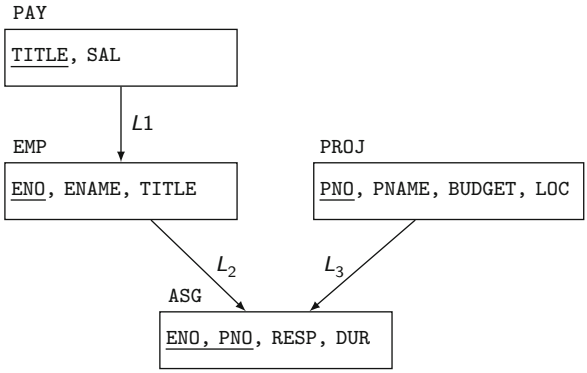

Fig. 2.5 Join graph representing relationships among relations

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,

$$\texttt{PNAME} = \text{“Maintenance”} \text{ and } \texttt{BUDGET} \leq 200000$$

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 $\texttt{R}_i$, 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.

$$p_1 : \texttt{TITLE} = \text{“Elect. Eng.”}$$
$$p_2 : \texttt{TITLE} = \text{“Syst. Anal.”}$$
$$p_3 : \texttt{TITLE} = \text{“Mech. Eng.”}$$
$$p_4 : \texttt{TITLE} = \text{“Programmer”}$$
$$p_5 : \texttt{SAL} \leq 30000$$

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

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

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]:

$$\mathtt{PROJ}_1 = \sigma_{\mathtt{BUDGET} \leq 200000}(\mathtt{PROJ})$$
$$\mathtt{PROJ}_2 = \sigma_{\mathtt{BUDGET} > 200000}(\mathtt{PROJ})$$

♦

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

$$\mathtt{PROJ}_2 = \sigma_{200000 < \mathtt{BUDGET} \wedge \mathtt{BUDGET} \leq 400000}(\mathtt{PROJ})$$
$$\mathtt{PROJ}_3 = \sigma_{\mathtt{BUDGET} > 400000}(\mathtt{PROJ})$$

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 $0 \leq$ BUDGET also needs to be included in Pr. We assume this to be true in all our examples and discussions in this chapter.

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

$PROJ_2$

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

$PROJ_3$

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 (BUDGET $\leq$ 200000, 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:

$$p_1 : \text{LOC} = \text{“Montreal”}$$
$$p_2 : \text{LOC} = \text{“New York”}$$
$$p_3 : \text{LOC} = \text{“Paris”}$$

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

$$p_4 : \text{BUDGET} \leq 200000$$
$$p_5 : \text{BUDGET} > 200000$$

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:

$$m_1 : (\text{LOC} = \text{“Montreal”}) \wedge (\text{BUDGET} \leq 200000)$$
$$m_2 : (\text{LOC} = \text{“Montreal”}) \wedge (\text{BUDGET} > 200000)$$
$$m_3 : (\text{LOC} = \text{“New York”}) \wedge (\text{BUDGET} \leq 200000)$$
$$m_4 : (\text{LOC} = \text{“New York”}) \wedge (\text{BUDGET} > 200000)$$
$$m_5 : (\text{LOC} = \text{“Paris”}) \wedge (\text{BUDGET} \leq 200000)$$
$$m_6 : (\text{LOC} = \text{“Paris”}) \wedge (\text{BUDGET} > 200000)$$

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 $\text{PAY}_1 \rightarrow \text{EMP}_1$ and $\text{PAY}_2 \rightarrow \text{EMP}_2$; 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 → EMP → 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 $F_{\mathrm{R}} = \{\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_w\}$, $\mathrm{R} = \bigcup \mathrm{R}_i, \quad \forall \mathrm{R}_i \in F_{\mathrm{R}}$.

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

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

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

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 $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_r$, each of which contains a subset of R's attributes as well as the primary key of RÀs 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

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

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

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

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

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

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

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

Computing each term, we get

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

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):

$$cont(\mathrm{A}_0, \mathrm{BUDGET}, \mathrm{PNO}) = 2bond(\mathrm{A}_0, \mathrm{BUDGET}) + 2bond(\mathrm{BUDGET}, \mathrm{PNO}) - 2bond(\mathrm{A}_0, \mathrm{PNO})$$

We know that

$$bond(\mathrm{A}_0, \mathrm{PNO}) = bond(\mathrm{A}_0, \mathrm{BUDGET}) = 0$$

$$bond(\mathrm{BUDGET}, \mathrm{PNO}) = 45 * 45 + 5 * 0 + 53 * 45 + 3 * 0 = 4410$$

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

(a)

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

(b)

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

(c)

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

(d)

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

Thus

$$cont(\mathrm{A}_0, \texttt{BUDGET}, \texttt{PNO}) = 8820$$

Ordering (1-3-2):

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

Thus

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

Ordering (2-3-4):

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

Thus

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

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 $\mathtt{A}_1$ and $\mathtt{A}_2$ that are accessed only by query q_1, and attributes $\mathtt{A}_3$ and $\mathtt{A}_4$ 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 $\{\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_i\}$ is at the upper left-hand corner (denoted TA) and the second set $\{\mathtt{A}_{i+1}, \ldots, \mathtt{A}_n\}$ 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:

$$
\begin{aligned}
AQ(q_i) &= \{\mathtt{A}_j | use(q_i, \mathtt{A}_j) = 1\} \\
TQ &= \{q_i | AQ(q_i) \subseteq TA\} \\
BQ &= \{q_i | AQ(q_i) \subseteq BA\} \\
OQ &= Q - \{TQ \cup BQ\}
\end{aligned}
$$

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.

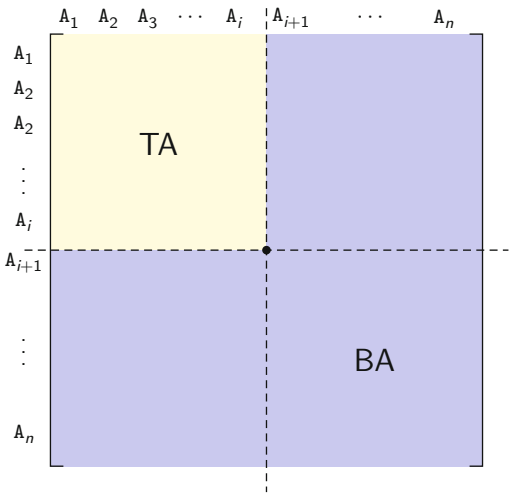

Fig. 2.15 Locating a splitting point

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}
  $best \leftarrow CTQ_{n-1} * CBQ_{n-1} - (COQ_{n-1})^2$
  repeat
    {determine the best partitioning}
    for i *from* $n-2$ *to* 1 *by* -1 **do**
      calculate CTQ_i
      calculate CBQ_i
      calculate COQ_i
      $z \leftarrow CTQ * CBQ_i - COQ_i^2$
      if $z > best$ **then** $best \leftarrow z$ {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
  $R_1 \leftarrow \Pi_{TA}(R) \cup K$ {K is the set of primary key attributes of R}
  $R_2 \leftarrow \Pi_{BA}(R) \cup K$
  $F \leftarrow \{R_1, R_2\}$
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 $F_R = \{R_1, R_2\}$, where $R_i \subseteq \{A_1, A_2 \ldots, A_n\}$ and $R_1 \cap R_2$ = the key attributes of relation R. Note 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 $F_{PROJ} = \{PROJ_1, PROJ_2\}$, where

$$PROJ_1 = \{PNO, BUDGET\}$$

$$PROJ_2 = \{PNO, PNAME, LOC\}$$

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 $\texttt{PROJ}_2$ as well as in $\texttt{PROJ}_1$. ♦

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 $\texttt{A} = \bigcup \texttt{R}_i$, 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 $F_\texttt{R} = \{\texttt{R}_1, \texttt{R}_2, \ldots, \texttt{R}_r\}$ and key attribute(s) K, $\texttt{R} = \bowtie_\texttt{K} \texttt{R}_i, \forall \texttt{R}_i \in F_\texttt{R}$. Therefore, as long as each $\texttt{R}_i$ is complete, the join operation will properly reconstruct RÅnother important point is that either each $\texttt{R}_i$ 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 trie-structured partitioning (Fig. 2.16). Since the two types of

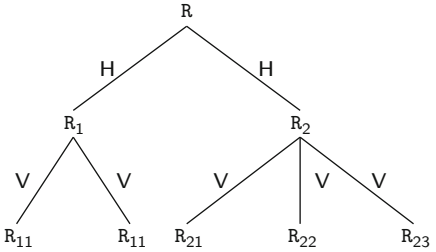

Fig. 2.16 Hybrid fragmentation

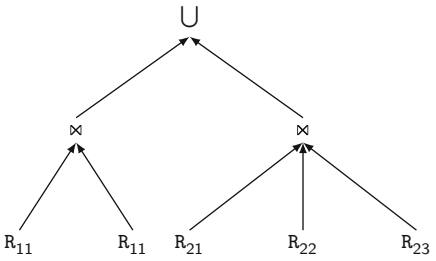

Fig. 2.17 Reconstruction of hybrid fragmentation

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 trie 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:

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

The first two terms in the above formula calculate the number of accesses of user query q_i to fragment F_j. Note that $(URij + RR_{ij})$ 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

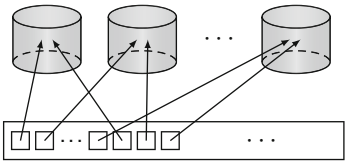

Fig. 2.19 Round-robin partitioning

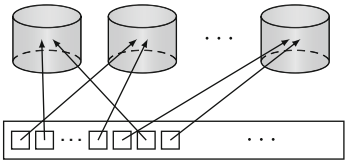

Fig. 2.20 Hash partitioning

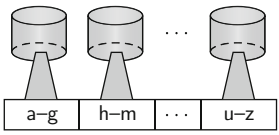

Fig. 2.21 Range partitioning

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

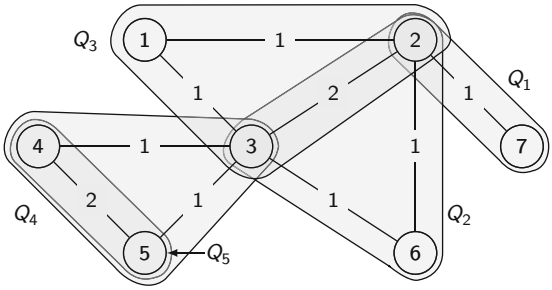

Fig. 2.22 Graph representation for partitioning in schism

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

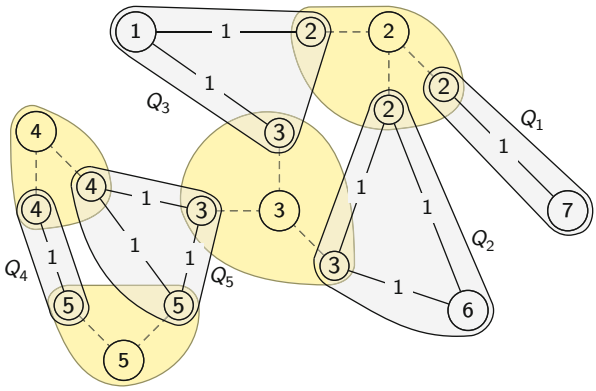

Fig. 2.23 Schism graph incorporating replication

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.

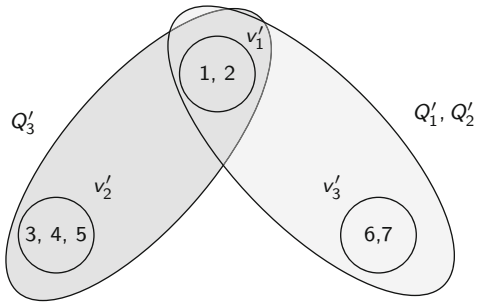

Fig. 2.24 Sword compressed hypergraph

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^r_{ij} and its write frequency f^w_{ij}. Using these, it computes the average read and write frequencies (ARF and AWF, respectively) of virtual vertex v'_i as follows:

$$ARF(v'_i) = \frac{\sum_j f^r_{ij}}{\log S(v'_i)} \quad \text{and} \quad AWF(v'_i) = \frac{\sum_j f^w_{ij}}{\log S(v'_i)}$$

$S(v'_i)$ 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:

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

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

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

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

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

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

Draw the join graph of $\text{EMP} \ltimes_{\text{TITLE}} \text{PAY}$. Is the graph simple or partitioned? If it is partitioned, modify the fragmentation of either EMP or PAY so that the join graph of $\text{EMP} \ltimes_{\text{TITLE}} \text{PAY}$ 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 $p_1 : \text{SAL} < 30000$ and $p_2 : \text{SAL} \geq 30000$ 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, $\text{A} = \{\text{A}_1, \ldots, \text{A}_5\}$ 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.

$$
\begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{array}
\begin{array}{c}
\begin{array}{ccccc} A_1 & A_2 & A_3 & A_4 & A_5 \end{array} \\
\left[\begin{array}{ccccc}
0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0
\end{array}\right]
\end{array}
$$

(a)

$$
\begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{array}
\begin{array}{c}
\begin{array}{ccc} S_1 & S_2 & S_3 \end{array} \\
\left[\begin{array}{ccc}
10 & 20 & 0 \\
5 & 0 & 10 \\
0 & 35 & 5 \\
0 & 10 & 0 \\
0 & 15 & 0
\end{array}\right]
\end{array}
$$

(b)

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 $use(q_i, \mathtt{A}_j)$ 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`.

Chapter 3
Distributed Data Control

An important requirement of a DBMS is the ability to support data control, i.e., controlling how data is accessed using a high-level language. Data control typically includes view management, access control, and semantic integrity control. Informally, these functions must ensure that *authorized* users perform *correct* operations on the database, thus contributing to the maintenance of database integrity. The functions necessary for maintaining the physical integrity of the database in the presence of concurrent accesses and failures are studied separately in Chap. 5 in the context of transaction management. In relational DBMSs, data control can be achieved in a uniform fashion. Views, authorizations, and semantic integrity constraints can be defined as rules that the system automatically enforces. The violation of some rules by database operations generally implies the rejection of the effects of some operations (e.g., undoing some updates) or propagating some effects (e.g., updating related data) to preserve the database integrity.

The definition of these rules is part of the administration of the database, a function generally performed by a database administrator (DBA). This person is also in charge of applying the organizational policies. Well-known solutions for data control have been proposed for centralized DBMSs. In this chapter, we discuss how these solutions can be extended to distributed DBMSs. The cost of enforcing data control, which is high in terms of resource utilization in a centralized DBMS, can be prohibitive in a distributed environment.

Since the rules for data control must be stored, the management of a distributed directory is also relevant in this chapter. The directory of a distributed DBMS can be viewed as a distributed database. There are several ways to store data control definitions, according to the way the directory is managed. Directory information can be stored differently according to its type; in other words, some information might be fully replicated, whereas other information might be distributed. For example, information that is useful at compile time, such as access control information, could be replicated. In this chapter, we emphasize the impact of directory management on the performance of data control mechanisms.

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_3

This chapter is organized as follows: View management is the subject of Sect. 3.1. Access control is presented in Sect. 3.2. Finally, semantic integrity control is treated in Sect. 3.3. For each section we first outline the solution in a centralized DBMS and then give the distributed solution, which is often an extension of the centralized one, although more difficult.

3.1 View Management

One of the main advantages of the relational model is that it provides full logical data independence. As introduced in Chap. 1, external schemas enable user groups to have their particular *view* of the database. In a relational system, a view is a *virtual relation*, defined as the result of a query on *base relations* (or real relations), but not materialized like a base relation in the database. A view is a dynamic window in the sense that it reflects all updates to the database. An external schema can be defined as a set of views and/or base relations. Besides their use in external schemas, views are useful for ensuring data security in a simple way. By selecting a subset of the database, views *hide* some data. If users may only access the database through views, they cannot see or manipulate the hidden data, which is therefore secure.

In the remainder of this section we look at view management in centralized and distributed systems as well as the problems of updating views. Note that in a distributed DBMS, a view can be derived from distributed relations, and the access to a view requires the execution of the distributed query corresponding to the view definition. An important issue in a distributed DBMS is to make view materialization efficient. We will see how the concept of materialized views helps in solving this problem, among others, but requires efficient techniques for materialized view maintenance.

3.1.1 Views in Centralized DBMSs

Most relational DBMSs use a view mechanism where a view is a relation derived from base relations as the result of a relational query (this was first proposed within the INGRES and System R projects). It is defined by associating the name of the view with the retrieval query that specifies it.

Example 3.1 The view of system analysts (`SYSAN`) derived from relation `EMP` can be defined by the following SQL query:

```
CREATE VIEW SYSAN(ENO, ENAME) AS
  SELECT ENO, ENAME
  FROM   EMP
  WHERE  TITLE = "Syst. Anal."
```

♦

The single effect of this statement is the storage of the view definition in the catalog. No other information needs to be recorded. Therefore, the result of the query defining the view (i.e., a relation having the attributes ENO and ENAME for the system analysts as shown in Fig. 3.1) is *not* produced. However, the view SYSAN can be manipulated as a base relation.

Example 3.2 The query

"Find the names of all the system analysts with their project number and responsibility(ies)"

involving the view SYSAN and relation ASG can be expressed as

```
SELECT ENAME, PNO, RESP
FROM   SYSAN NATURAL JOIN ASG
```

♦

Mapping a query expressed on views into a query expressed on base relations can be done by *query modification*. With this technique the variables are changed to range on base relations and the query qualification is merged (ANDed) with the view qualification.

Example 3.3 The preceding query can be modified to

```
SELECT ENAME, PNO, RESP
FROM   EMP NATURAL JOIN ASG
WHERE  TITLE = "Syst. Anal."
```

The result of this query is illustrated in Fig. 3.2. ♦

The modified query is expressed on base relations and can therefore be processed by the query processor. It is important to note that view processing can be done at compile time. The view mechanism can also be used for refining the access controls

SYSAN

ENO	ENAME
E2	M. Smith
E5	B. Casey
E8	J. Jones

Fig. 3.1 Relation corresponding to the view SYSAN

ENAME	PNO	RESP
M. Smith	P1	Analyst
M. Smith	P2	Analyst
B. Casey	P3	Manager
J. Jones	P4	Manager

Fig. 3.2 Result of query involving view SYSAN

to include subsets of objects. To specify any user from whom one wants to hide data, the keyword USER generally refers to the logged-on user identifier.

Example 3.4 The view ESAME restricts the access by any user to those employees having the same title:

```
CREATE VIEW ESAME AS
  SELECT *
  FROM   EMP E1, EMP E2
  WHERE  E1.ENO = E2.ENO
  AND    E1.ENO = USER
```

In the view definition above, * stands for "all attributes" and the two tuple variables (E1 and E2) ranging over relation EMP are required to express the join of one tuple of EMP (the one corresponding to the logged-on user) with all tuples of EMP based on the same title. For example, the following query issued by the user J. Doe

```
SELECT *
FROM   ESAME
```

returns the relation of Fig. 3.3. Note that the user J. Doe also appears in the result. If the user who creates ESAME is an electrical engineer, as in this case, the view represents the set of all electrical engineers. ♦

Views can be defined using arbitrarily complex relational queries involving selection, projection, join, aggregate functions, and so on. All views can be interrogated as base relations, but not all views can be manipulated as such. Updates through views can be handled automatically only if they can be propagated correctly to the base relations. We can classify views as being updatable and not updatable. A view is updatable only if the updates to the view can be propagated to the base relations without ambiguity. The view SYSAN above is updatable; the insertion, for example, of a new system analyst ⟨201, Smith⟩ will be mapped into the insertion of a new employee ⟨201, Smith, Syst. Anal.⟩. If attributes other than TITLE were hidden by the view, they would be assigned *null values*.

Example 3.5 However, the following view, which uses a natural join (i.e., the equijoin of two relations on a common attribute), is not updatable:

```
CREATE VIEW EG(ENAME, RESP) AS
  SELECT DISTINCT ENAME, RESP
  FROM   EMP NATURAL JOIN ASG
```

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	L. Chu	Elect. Eng.

Fig. 3.3 Result of query on view ESAME

The deletion, for example, of the tuple ⟨Smith, Analyst⟩ cannot be propagated, since it is ambiguous. Deletions of Smith in relation EMP or analyst in relation ASG are both meaningful, but the system does not know which is correct. ♦

Current systems are very restrictive about supporting updates through views. Views can be updated only if they are derived from a single relation by selection and projection. This precludes views defined by joins, aggregates, and so on. However, it is theoretically possible to automatically support updates of a larger class of views. It is interesting to note that views derived by join are updatable if they include the keys of the base relations.

3.1.2 Views in Distributed DBMSs

The definition of a view is similar in a distributed DBMS and in centralized systems. However, a view in a distributed system may be derived from fragmented relations stored at different sites. When a view is defined, its name and its retrieval query are stored in the catalog.

Since views may be used as base relations by application programs, their definition should be stored in the directory in the same way as the base relation descriptions. Depending on the degree of site autonomy offered by the system, view definitions can be centralized at one site, partially duplicated or fully duplicated. In any case, the information associating a view name to its definition site should be duplicated. If the view definition is not present at the site where the query is issued, remote access to the view definition site is necessary.

The mapping of a query expressed on views into a query expressed on base relations (which can potentially be fragmented) can also be done through query modification in the same way as in centralized DBMSs. With this technique, the qualification defining the view is found in the distributed database catalog and then merged with the query to provide a query on base relations. Such a modified query is a *distributed query*, which can be processed by the distributed query processor (see Chap. 4). The query processor maps the distributed query into a query on physical fragments.

In Chap. 2 we presented alternative ways of fragmenting base relations. The definition of fragmentation is, in fact, very similar to the definition of particular views. Thus, it is possible to manage views and fragments using a unified mechanism. Furthermore, replicated data can be handled in the same way. The value of such a unified mechanism is to facilitate distributed database administration. The objects manipulated by the database administrator can be seen as a hierarchy where the leaves are the fragments from which relations and views can be derived. Therefore, the DBA may increase locality of reference by making views in one-to-one correspondence with fragments. For example, it is possible to implement the view SYSAN illustrated in Example 3.1 by a fragment at a given site, provided that most users accessing the view SYSAN are at the same site.

Evaluating views derived from distributed relations may be costly. In a given organization it is likely that many users access the same view which must be recomputed for each user. We saw in Sect. 3.1.1 that view derivation is done by merging the view qualification with the query qualification. An alternative solution is to avoid view derivation by maintaining actual versions of the views, called *materialized views*. A *materialized view* stores the tuples of a view in a database relation, like the other database tuples, possibly with indices. Thus, access to a materialized view is much faster than deriving the view, in particular, in a distributed DBMS where base relations can be remote. Introduced in the early 1980s, materialized views have since gained much interest in the context of data warehousing to speed up Online Analytical Processing (OLAP) applications. Materialized views in data warehouses typically involve aggregate (such as **SUM** and **COUNT**) and grouping (**GROUP BY**) operators because they provide compact database summaries. Today, all major database products support materialized views.

Example 3.6 The following view over relation PROJ(PNO,PNAME,BUDGET,LOC) gives, for each location, the number of projects and the total budget.

```
CREATE VIEW PL(LOC, NBPROJ, TBUDGET) AS
  SELECT LOC, COUNT(*),SUM(BUDGET)
  FROM   PROJ
  GROUP BY LOC
```

♦

3.1.3 Maintenance of Materialized Views

A materialized view is a copy of some base data and thus must be kept consistent with that base data which may be updated. *View maintenance* is the process of updating (or refreshing) a materialized view to reflect the changes made to the base data. The issues related to view materialization are somewhat similar to those of database replication which we will address in Chap. 6. However, a major difference is that materialized view expressions, in particular, for data warehousing, are typically more complex than replica definitions and may include join, group by, and aggregate operators. Another major difference is that database replication is concerned with more general replication configurations, e.g., with multiple copies of the same base data at multiple sites.

A view maintenance policy allows a DBA to specify *when* and *how* a view should be refreshed. The first question (when to refresh) is related to consistency (between the view and the base data) and efficiency. A view can be refreshed in two modes: *immediate* or *deferred*. With the immediate mode, a view is refreshed immediately as part as the transaction that updates base data used by the view. If the view and the base data are managed by different DBMSs, possibly at different sites, this requires the use of a distributed transaction, for instance, using the two-phase commit (2PC) protocol (see Chap. 5). The main advantages of immediate refreshment are that

the view is always consistent with the base data and that read-only queries can be fast. However, this is at the expense of increased transaction time to update both the base data and the views within the same transactions. Furthermore, using distributed transactions may be difficult.

In practice, the deferred mode is preferred because the view is refreshed in separate (refresh) transactions, thus without performance penalty on the transactions that update the base data. The refresh transactions can be triggered at different times: *lazily*, just before a query is evaluated on the view; *periodically*, at predefined times, e.g., every day; or *forcedly*, after a predefined number of updates to the base data. Lazy refreshment enables queries to see the latest consistent state of the base data but at the expense of increased query time to include the refreshment of the view. Periodic and forced refreshment allow queries to see views whose state is not consistent with the latest state of the base data. The views managed with these strategies are also called *snapshots*.

The second question (how to refresh a view) is an important efficiency issue. The simplest way to refresh a view is to recompute it from scratch using the base data. In some cases, this may be the most efficient strategy, e.g., if a large subset of the base data has been changed. However, there are many cases where only a small subset of view needs to be changed. In these cases, a better strategy is to compute the view *incrementally*, by computing only the changes to the view. Incremental view maintenance relies on the concept of differential relation. Let u be an update of relation R. $\mathtt{R}^+$ and $\mathtt{R}^-$ are *differential relations* of R by u, where $\mathtt{R}^+$ contains the tuples inserted by u into R, and $\mathtt{R}^-$ contains the tuples of R deleted by u. If u is an insertion, $\mathtt{R}^-$ is empty. If u is a deletion, $\mathtt{R}^+$ is empty. Finally, if u is a modification, relation R can be obtained by computing $(\mathtt{V} - \mathtt{V}^-) \cup \mathtt{V}^+$. Computing the changes to the view, i.e., $\mathtt{V}^+$ and $\mathtt{V}^-$, may require using the base relations in addition to differential relations.

Example 3.7 Consider the view EG of Example 3.5 which uses relations EMP and ASG as base data and assume its state is derived from that of Example 3.1, so that EG has 9 tuples (see Fig. 3.4). Let $\mathtt{EMP}^+$ consist of one tuple ⟨E9, B. Martin, Programmer⟩ to be inserted in EMP, and $\mathtt{ASG}^+$ consist of two tuples ⟨E4, P3, Programmer, 12⟩ and ⟨E9, P3, Programmer, 12⟩ to be inserted in ASG. The changes to the view EG can be computed as:

```
EG+ = (SELECT ENAME, RESP
       FROM   EMP NATURAL JOIN ASG+)
             UNION
      (SELECT ENAME, RESP
       FROM   EMP+ NATURAL JOIN ASG)
             UNION
      (SELECT ENAME, RESP
       FROM   EMP+ NATURAL JOIN ASG+)
```

which yields tuples ⟨B. Martin, Programmer⟩ and ⟨J. Miller, Programmer⟩. Note that integrity constraints would be useful here to avoid useless work (see Sect. 3.3.2). Assuming that relations EMP and ASG are related by a referential constraint that

says that ENO in ASG must exist in EMP, the second **SELECT** statement is useless as it produces an empty relation. ♦

Efficient techniques have been devised to perform incremental view maintenance using both the materialized views and the base relations. The techniques essentially differ in their views' expressiveness, their use of integrity constraints, and the way they handle insertion and deletion. They can be classified along the view expressiveness dimension as nonrecursive views, views involving outerjoins, and recursive views. For nonrecursive views, i.e., select-project-join (SPJ) views that may have duplicate elimination, union, and aggregation, an elegant solution is the *counting algorithm*. One problem stems from the fact that individual tuples in the view may be derived from several tuples in the base relations, thus making deletion in the view difficult. The basic idea of the counting algorithm is to maintain a count of the number of derivations for each tuple in the view, and to increment (resp. decrement) tuple counts based on insertions (resp. deletions); a tuple in the view of which count is zero can then be deleted.

Example 3.8 Consider the view EG in Fig. 3.4. Each tuple in EG has one derivation (i.e., a count of 1) except tuple ⟨M. Smith, Analyst⟩ which has two (i.e., a count of 2). Assume now that tuples ⟨E2, P1, Analyst, 24⟩ and ⟨E3, P3, Consultant, 10⟩ are deleted from ASG. Then only tuple ⟨A. Lee, Consultant⟩ needs to be deleted from EG. ♦

We now present the basic counting algorithm for refreshing a view V defined over two relations R and S as a query q(R, S). Assuming that each tuple in V has an associated derivation count, the algorithm has three main steps (see Algorithm 3.1). First, it applies the view differentiation technique to formulate the differential views V^+ and V^- as queries over the view, the base relations, and the differential relations. Second, it computes V^+ and V^- and their tuple counts. Third, it applies the changes V^+ and V^- in V by adding positive counts and subtracting negative counts, and deleting tuples with a count of zero.

The counting algorithm is optimal since it computes exactly the view tuples that are inserted or deleted. However, it requires access to the base relations. This implies that the base relations be maintained (possibly as replicas) at the sites of the

ENAME	RESP
J. Doe	Manager
M. Smith	Analyst
A. Lee	Consultant
A. Lee	Engineer
J. Miller	Programmer
B. Casey	Manager
L. Chu	Manager
R. Davis	Engineer
J. Jones	Manager

Fig. 3.4 State of view EG

Algorithm 3.1: COUNTING

Input: V: view defined as $q(\mathtt{R}, \mathtt{S})$; R, S: relations; $\mathtt{R}^+$, $\mathtt{R}^-$: changes to R
begin
  $\mathtt{V}^+ = q^+(\mathtt{V}, \mathtt{R}^+, \mathtt{R}, \mathtt{S})$
  $\mathtt{V}^- = q^-(\mathtt{V}, \mathtt{R}^-, \mathtt{R}, \mathtt{S})$
  compute $\mathtt{V}^+$ with positive counts for inserted tuples
  compute $\mathtt{V}^-$ with negative counts for deleted tuples
  compute $(\mathtt{V} - \mathtt{V}^-) \cup \mathtt{V}^+$ by adding positive counts and subtracting negative counts
    deleting each tuple in V with count = 0;
end

materialized view. To avoid accessing the base relations so the view can be stored at a different site, the view should be maintainable using only the view and the differential relations. Such views are called *self-maintainable*.

Example 3.9 Consider the view SYSAN in Example 3.1. Let us write the view definition as SYSAN=q(EMP) meaning that the view is defined by a query q on EMP. We can compute the differential views using only the differential relations, i.e., $\mathtt{SYSAN}^+ = q(\mathtt{EMP}^+)$ and $\mathtt{SYSAN}^- = q(\mathtt{EMP}^-)$. Thus, the view SYSAN is self-maintainable. ♦

Self-maintainability depends on the views' expressiveness and can be defined with respect to the update type (insertion, deletion, or modification). Most SPJ views are not self-maintainable with respect to insertion but are often self-maintainable with respect to deletion and modification. For instance, an SPJ view is self-maintainable with respect to deletion of relation R if the key attributes of R are included in the view.

Example 3.10 Consider the view EG of Example 3.5. Let us add attribute ENO (which is key of EMP) in the view definition. This view is not self-maintainable with respect to insertion. For instance, after an insertion of an ASG tuple, we need to perform the join with EMP to get the corresponding ENAME to insert in the view. However, this view is self-maintainable with respect to deletion on EMP. For instance, if one EMP tuple is deleted, the view tuples having same ENO can be deleted. ♦

We discuss two optimizations that can significantly reduce the maintenance time of the COUNTING algorithm. The first optimization is to materialize views representing subqueries of the input query. A view is constructed by removing a subset of relations from the query. These views are increasingly smaller and build a hierarchy. F-IVM method constructs such a hierarchy, called a view trie, with the input query at the top, the relations at the leaves, and inner views defined by project-join-aggregate queries over their children. Updates to a relation are propagated bottom-up in this view trie. The views that are on the path from the updated relation to the root are maintained using the delta processing from the COUNTING algorithm. All other views remain unchanged; if they are materialized, then they may speed up this delta processing. For a restricted class of acyclic queries, called

q-hierarchical, such view trees allow for constant-time updates to any of the input relations.

The second optimization exploits the skew in the data. Values that appear very often in the database are deemed heavy, while all others are light. IVM$^\epsilon$ uses evaluation strategies that are sensitive to the heavy/light skew in the data and that use materialized views and delta computation like all aforementioned maintenance algorithms.

We exemplify these two optimizations for a query that counts the number of triangles in a graph. We would like to refresh this triangle count immediately and incrementally under one update to the data graph, which can be an edge insertion or deletion. Let us consider three copies `R`, `S`, and `T` of the binary edge relation of a graph with N edges. We record the multiplicities of tuples in the input relations and views, that is, the number of their derivations, in a separate column `P`. Assuming the schemas of the relations are `(A,B,P_R)`, `(B,C,P_S)`, and `(C,A,P_T)`, the triangle count query is

```
CREATE VIEW Q(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   R NATURAL JOIN S NATURAL JOIN T
```

The insertion or deletion of an edge triggers updates to each of the three relation copies. We discuss the case of updating `R`; the other two cases are treated similarly. We model this update as a relation `deltaR` consisting of a single tuple (a, b, p), where (a, b) defines the updated edge and p is the multiplicity. Following the formalism of generalized multiset relations, we model both inserts and deletes uniformly by allowing for multiplicities to be integers, that is, negative and positive numbers. Then, for inserting or deleting the edge three times, we set the multiplicity p to $+3$ or, respectively, to -3.

The COUNTING algorithm computes on the fly a delta query `deltaQ` that represents the change to the query result: This query is the same as `Q`, where we replace `R` by `deltaR`. This delta computation takes $\mathcal{O}(N)$ time since it needs to intersect two lists of possibly $\mathcal{O}(N)$ many C-values that are paired with b in `S` and with a in `T` (that is, the multiplicity of such pairs in `S` and `T` is nonzero).

The DBToaster approach speeds up the delta computation by precomputing three auxiliary views representing the update-independent parts of the delta queries for updates to the three relations:

```
CREATE VIEW V_ST(B, A, CNT) AS
  SELECT   B, A, SUM(P_S * P_T) as CNT
  FROM     S NATURAL JOIN T
  GROUP BY B, A

CREATE VIEW V_TR(C, B, CNT) AS
  SELECT   C, B, SUM(P_T * P_R) as CNT
  FROM     T NATURAL JOIN R
  GROUP BY C, B
```

```
CREATE VIEW V_RS(A, C, CNT) AS
  SELECT   A, C, SUM(P_R * P_S) as CNT
  FROM     R NATURAL JOIN S
  GROUP BY A, C
```

The view V_ST allows to compute the delta query deltaQ in $\mathcal{O}(1)$ time, since the join of deltaR and V_ST requires a constant-time lookup for $\langle a, b\rangle$ into V_ST. However, maintaining the views V_RS and V_TR, which are defined using R, still requires $\mathcal{O}(N)$ time.

The F-IVM method materializes only one of the three views, for instance, V_ST. In this case, the maintenance under updates to R takes $\mathcal{O}(1)$ time, but the maintenance of S and T under updates still takes $\mathcal{O}(N)$ time.

IVM$^\epsilon$ algorithm partitions the nodes in the graph depending on their degree, that is, on the number of directly connected nodes: The *heavy* nodes have degree greater than or equal to $N^{1/2}$, while the *light* nodes have degree less than $N^{1/2}$. This leads to a partition of each of the three copies R, S, and T of the edge relation into a heavy part R_h (S_h, T_h) and a light part R_l (S_l, T_l): a tuple $\langle a, b, p\rangle$ is in R_h if a is heavy and in R_l otherwise; similarly, a tuple $\langle b, c, p\rangle$ is in S_h if b is heavy and in S_l otherwise; finally, $\langle c, a, p\rangle$ is in T_h if c is heavy and in T_l otherwise. We can rewrite Q by replacing each of the three relations with the union of its two parts. The query Q is then equivalent to the union of eight skew-aware views Q_r,s,t, where $r, s, t \in \{h, l\}$:

```
CREATE VIEW   Q_r,s,t(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   R_r NATURAL JOIN S_s NATURAL JOIN T_t
```

Consider a single-tuple update deltaR_r $= \{(a, b, p)\}$ to the part R_r of relation R for $r \in \{h, l\}$. The delta computation for a view Q_r,s,t is then given by the following simpler query:

```
CREATE VIEW   deltaQ_r,s,t(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   deltaR_r NATURAL JOIN S_s NATURAL JOIN T_t
  WHERE  S_s.A = a AND T_t.B = b
```

IVM$^\epsilon$ adapts its maintenance strategy to each skew-aware view to achieve the sublinear update time. While most of these views trivially achieve the $\mathcal{O}(N^{1/2})$ upper bound, there is one exception. We next explain how to achieve this bound for maintaining each of these views.

The delta computation for the four views Q_r,l,t (for $r, t \in \{h, l\}$) is expressed as follows:

```
CREATE VIEW   deltaQ_r,l,t(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   deltaR_r NATURAL JOIN S_l NATURAL JOIN T_t
  WHERE  S_l.A = a AND T_t.B = b
```

It joins the parts S_l with T_t on C. Since the update deltaR_r sets B to b in S_l and b can only be a light value in S_l, there are at most $N^{1/2}$ C-values paired with b in S_l. The intersection of the set of C-values in S_l and T_t can then take at most $\mathcal{O}(N^{1/2})$ time.

The delta computation for the views Q_r,h,h is expressed similarly. Since all C-values in T_h are heavy, each of them has at least $N^{1/2}$ A-values. This also means there are at most $N^{1/2}$ heavy C-values. The intersection of the set of the heavy C-values in T_h with the C-values in S_h can then take at most $\mathcal{O}(N^{1/2})$ time.

However, the delta computation for the views Q_r,h,l for $r \in \{h, l\}$ needs linear time, since it requires iterating over all the C-values c paired with b in S_h and with a in T_l; the number of such C-values can be linear in the size of the database. In this case, IVM$^\epsilon$ precomputes the update-independent parts of the delta queries as auxiliary materialized views and then exploits these views to speed up the delta evaluation:

```
CREATE VIEW V_ST(B, A, CNT) AS
  SELECT B, A, SUM(P_S * P_T) as CNT
  FROM   S_h NATURAL JOIN T_l
  GROUP BY B, A
```

We materialize similar views V_RS and V_TR in case of updates to T and, respectively, S. Each of these views needs $\mathcal{O}(N^{3/2})$ space. We can now compute deltaQ_r,h,l using V_ST as

```
CREATE VIEW   deltaQ_r,h,l(CNT) AS
  SELECT SUM(P_R * CNT) as CNT
  FROM   deltaR_r NATURAL JOIN V_ST
  WHERE  V_ST.B = b AND V_ST.A = a
```

This takes $\mathcal{O}(1)$ time since we only need a lookup in V_ST to fetch the multiplicity of the edge (a, b) followed by the multiplication with p from deltaR_r.

3.2 Access Control

Access control is an important aspect of data security, the function of a database system that protects data against unauthorized access. Another important aspect is *data protection*, to prevent unauthorized users from understanding the physical content of data. This function is typically provided by file systems in the context of centralized and distributed operating systems. The main data protection approach is data encryption.

Access control must guarantee that only authorized users perform operations they are allowed to perform on the database. Many different users may have access to a large collection of data under the control of a single centralized or distributed system. The centralized or distributed DBMS must thus be able to restrict the access

of a subset of the database to a subset of the users. Access control has long been provided by operating systems as services of the file system. In this context, a centralized control is offered. Indeed, the central controller creates objects, and may allow particular users to perform particular operations (read, write, execute) on these objects. Also, objects are identified by their external names.

Access control in database systems differs in several aspects from that in traditional file systems. Authorizations must be refined so that different users have different rights on the same database objects. This requirement implies the ability to specify subsets of objects more precisely than by name and to distinguish between groups of users. In addition, the decentralized control of authorizations is of particular importance in a distributed context. In relational systems, authorizations can be uniformly controlled by database administrators using high-level constructs. For example, controlled objects can be specified by predicates in the same way as is a query qualification.

There are two main approaches to database access control. The first approach is called *discretionary access control (DAC)* and has long been provided by DBMS. DAC defines access rights based on the users, the type of access (e.g., **`SELECT`**, **`UPDATE`**), and the objects to be accessed. The second approach, called *mandatory access control (MAC)* further increases security by restricting access to classified data to cleared users. Support of MAC by major DBMSs is more recent and stems from increased security threats coming from the Internet. Other approaches go further into adding more semantics to access control, in particular, role-based access control, which considers users with different roles, and purpose-based access control, e.g., hippocratic databases, which associates purpose information with data, i.e., the reasons for data collection and access.

From solutions to access control in centralized systems, we derive those for distributed DBMSs. However, there is the additional complexity which stems from the fact that objects and users can be distributed. In what follows we first present discretionary and mandatory access control in centralized systems and then the additional problems and their solutions in distributed systems.

3.2.1 Discretionary Access Control

Three main actors are involved in DAC: the *subject* (e.g., users, groups of users) who trigger the execution of application programs; the *operations*, which are embedded in application programs; and the *database objects*, on which the operations are performed. Authorization control consists of checking whether a given triple (subject, operation, object) can be allowed to proceed (i.e., the user can execute the operation on the object). An authorization can be viewed as a triple (subject, operation type, object definition) which specifies that the subjects have the right to perform an operation of operation type on an object. To control authorizations properly, the DBMS requires the definition of subjects, objects, and access rights.

The introduction of a subject in the system is typically done by a pair (user name, password). The user name uniquely *identifies* the users of that name in the system, while the password, known only to the users of that name, *authenticates* the users. Both user name and password must be supplied in order to log in the system. This prevents people who do not know the password from entering the system with only the user name.

The objects to protect are subsets of the database. Relational systems provide finer and more general protection granularity than do earlier systems. In a file system, the protection granule is the file. In a relational system, objects can be defined by their type (view, relation, tuple, attribute) as well as by their content using selection predicates. Furthermore, the view mechanism as introduced in Sect. 3.1 permits the protection of objects simply by hiding subsets of relations (attributes or tuples) from unauthorized users.

A right expresses a relationship between a subject and an object for a particular set of operations. In an SQL-based relational DBMS, an operation is a high-level statement such as **SELECT**, **INSERT**, **UPDATE**, or **DELETE**, and rights are defined (granted or revoked) using the following statements:

GRANT ⟨operation type(s)⟩ **ON** ⟨object⟩ TO ⟨subject(s)⟩
REVOKE ⟨operation type(s)⟩ **FROM** ⟨object⟩ TO ⟨subject(s)⟩

The keyword *public* can be used to mean all users. Authorization control can be characterized based on who (the grantors) can grant the rights. To ease database administration, it is convenient to define user groups, as in operating systems, for the purpose of authorization. Once defined, a user group can be used as subject in **GRANT** and **REVOKE** statements.

In its simplest form, the control is centralized: a single user or user class, the database administrators, has all privileges on the database objects and is the only one allowed to use the **GRANT** and **REVOKE** statements. A more flexible form of control is decentralized: the creator of an object becomes its owner and is granted all privileges on it. In particular, there is the additional operation type **GRANT**, which transfers all the rights of the grantor performing the statement to the specified subjects. Therefore, the person receiving the right (the grantee) may subsequently grant privileges on that object. Thus, access control is discretionary in the sense that users with grant privilege can make access policy decisions. The revoking process is complex as it must be recursive. For example, if A, who granted B who granted C the **GRANT** privilege on object O, wants to revoke all the privileges of B on O, all the privileges of C on O must also be revoked. To perform revocation, the system must maintain a hierarchy of grants per object where the creator of the object is the root.

The privileges of the subjects over objects are recorded in the catalog (directory) as authorization rules. There are several ways to store the authorizations. The most convenient approach is to consider all the privileges as an *authorization matrix*, in which a row defines a subject, a column an object, and a matrix entry (for a pair ⟨subject, object⟩), the authorized operations. The authorized operations are specified by their operation type (e.g., **SELECT**, **UPDATE**). It is also customary to associate

with the operation type a predicate that further restricts the access to the object. The latter option is provided when the objects must be base relations and cannot be views. For example, one authorized operation for the pair ⟨Jones, relation EMP⟩ could be

```
SELECT WHERE TITLE = "Syst.Anal."
```

which authorizes Jones to access only the employee tuples for system analysts. Figure 3.5 gives an example of an authorization matrix where objects are either relations (EMP and ASG) or attributes (ENAME).

The authorization matrix can be stored in three ways: by row, by column, or by element. When the matrix is stored by *row*, each subject is associated with the list of objects that may be accessed together with the related access rights. This approach makes the enforcement of authorizations efficient, since all the rights of the logged-on user are together (in the user profile). However, the manipulation of access rights per object (e.g., making an object public) is not efficient since all subject profiles must be accessed. When the matrix is stored by *column*, each object is associated with the list of subjects who may access it with the corresponding access rights. The advantages and disadvantages of this approach are the reverse of the previous approach.

The respective advantages of the two approaches can be combined in the third approach, in which the matrix is stored by *element*, that is, by relation (subject, object, right). This relation can have indices on both subject and object, thereby providing fast-access right manipulation per subject and per object.

Directly managing relationships between many subjects and many objects gets complicated for database administrators. *Role-based access control* (RBAC) addresses this problem by adding roles, as a level of independence between subjects and objects. Roles correspond to various job functions (e.g., clerk, analyst, manager, etc.), users are assigned particular roles, and authorizations on objects are assigned to specific roles. Thus, users no longer acquire authorizations directly, but only through their roles. Since there are not that many roles, RBAC simplifies much access control, in particular when adding or modifying user accounts.

	EMP	ENAME	ASG
Casey	UPDATE	UPDATE	UPDATE
Jones	SELECT	SELECT	SELECT WHERE RESP $\neq$ "Manager"
Casey	NONE	SELECT	NONE

Fig. 3.5 Example of authorization matrix

3.2.2 Mandatory Access Control

DAC has some limitations. One problem is that a malicious user can access unauthorized data through an authorized user. For instance, consider user A who has authorized access to relations R and S and user B who has authorized access to relation S only. If B somehow manages to modify an application program used by A so it writes R data into S, then B can read unauthorized data without violating authorization rules.

MAC answers this problem and further improves security by defining different security levels for both subjects and data objects. Furthermore, unlike DAC, the access policy decisions are under the control of a single administrator, i.e., users cannot define their own policies and grant access to objects. MAC in databases is based on the well-known Bell-LaPadula model designed for operating system security. In this model, subjects are processes acting on a user's behalf; a process has a security level also called *clearance* derived from that of the user. In its simplest form, the security levels are Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U), and ordered as $TS > S > C > U$, where ">" means "more secure." Access in read and write modes by subjects is restricted by two simple rules:

1. A subject T is allowed to read an object of security level l only if $level(T) \geq l$.
2. A subject T is allowed to write an object of security level l only if $class(T) \leq l$.

Rule 1 (called "no read up") protects data from unauthorized disclosure, i.e., a subject at a given security level can only read objects at the same or lower security levels. For instance, a subject with secret clearance cannot read top-secret data. Rule 2 (called "no write down") protects data from unauthorized change, i.e., a subject at a given security level can only write objects at the same or higher security levels. For instance, a subject with top-secret clearance can only write top-secret data but cannot write secret data (which could then contain top-secret data).

In the relational model, data objects can be relations, tuples, or attributes. Thus, a relation can be classified at different levels: relation (i.e., all tuples in the relation have the same security level), tuple (i.e., every tuple has a security level), or attribute (i.e., every distinct attribute value has a security level). A classified relation is thus called *multilevel relation* to reflect that it will appear differently (with different data) to subjects with different clearances. For instance, a multilevel relation classified at the tuple level can be represented by adding a security level attribute to each tuple. Similarly, a multilevel relation classified at attribute level can be represented by adding a corresponding security level to each attribute. Figure 3.6 illustrates a multilevel relation PROJ* based on relation PROJ which is classified at the attribute level. Note that the additional security level attributes may increase significantly the size of the relation.

The entire relation also has a security level which is the lowest security level of any data it contains. For instance, relation PROJ* has security level C. A relation can then be accessed by any subject having a security level which is the same or

PROJ*

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S

Fig. 3.6 Multilevel relation PROJ⋆ classified at the attribute level

PROJ*C

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	Null	S	Null	S

Fig. 3.7 Confidential relation PROJ⋆C

PROJ**

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S
P3	C	Web Develop.	C	200000	C	Paris	C

Fig. 3.8 Multilevel relation with polyinstantiation

higher. However, a subject can only access data for which it has clearance. Thus, attributes for which a subject has no clearance will appear to the subject as null values with an associated security level which is the same as the subject. Figure 3.7 shows an instance of relation PROJ* as accessed by a subject at a confidential security level.

MAC has strong impact on the data model because users do not see the same data and have to deal with unexpected side-effects. One major side-effect is called *polyinstantiation*, which allows the same object to have different attribute values depending on the users' security level. Figure 3.8 illustrates a multirelation with polyinstantiated tuples. Tuple of primary key P3 has two instantiations, each one with a different security level. This may result from a subject T with security level C inserting a tuple with key="P3" in relation PROJ⋆ in Fig. 3.6. Because T (with confidential clearance level) should ignore the existence of tuple with key="P3" (classified as secret), the only practical solution is to add a second tuple with same key and different classification. However, a user with secret clearance would see both tuples with key="E3" and should interpret this unexpected effect.

3.2.3 Distributed Access Control

The additional problems of access control in a distributed environment stem from the fact that objects and subjects are distributed and that messages with sensitive data can be read by unauthorized users. These problems are: remote user authentication, management of discretionary access rules, handling of views and of user groups, and enforcing MAC.

Remote user authentication is necessary since any site of a distributed DBMS may accept programs initiated, and authorized, at remote sites. To prevent remote access by unauthorized users or applications (e.g., from a site that is not part of the distributed DBMS), users must also be identified and authenticated at the accessed site. Furthermore, instead of using passwords that could be obtained from sniffing messages, encrypted certificates could be used.

Three solutions are possible for managing authentication:

1. Authentication information is maintained at a central site for *global users* which can then be authenticated only once and then accessed from multiple sites.
2. The information for authenticating users (user name and password) is replicated at all sites in the catalog. Local programs, initiated at a remote site, must also indicate the user name and password.
3. All sites of the distributed DBMS identify and authenticate themselves similar to the way users do. Intersite communication is thus protected by the use of the site password. Once the initiating site has been authenticated, there is no need for authenticating their remote users.

The first solution simplifies password administration significantly and enables single authentication (also called single sign on). However, the central authentication site can be a single point of failure and a bottleneck. The second solution is more costly in terms of directory management given that the introduction of a new user is a distributed operation. However, users can access the distributed database from any site. The third solution is necessary if user information is not replicated. Nevertheless, it can also be used if there is replication of the user information. In this case it makes remote authentication more efficient. If user names and passwords are not replicated, they should be stored at the sites where the users access the system (i.e., the home site). The latter solution is based on the realistic assumption that users are more static, or at least they always access the distributed database from the same site.

Distributed authorization rules are expressed in the same way as centralized ones. Like view definitions, they must be stored in the catalog. They can be either fully replicated at each site or stored at the sites of the referenced objects. In the latter case the rules are duplicated only at the sites where the referenced objects are distributed. The main advantage of the fully replicated approach is that authorization can be processed by query modification at compile time. However, directory management is more costly because of data duplication. The second solution is better if locality of reference is very high. However, distributed authorization cannot be controlled at compile time.

Views may be considered to be objects by the authorization mechanism. Views are composite objects, that is, composed of other underlying objects. Therefore, granting access to a view translates into granting access to underlying objects. If view definition and authorization rules for all objects are fully replicated (as in many systems), this translation is rather simple and can be done locally. The translation is harder when the view definition and its underlying objects are all stored separately, as is the case with site autonomy assumption. In this situation, the translation is a totally distributed operation. The authorizations granted on views depend on the access rights of the view creator on the underlying objects. A solution is to record the association information at the site of each underlying object.

Handling user groups for the purpose of authorization simplifies distributed database administration. In a centralized DBMS, "all users" can be referred to as *public*. In a distributed DBMS, the same notion is useful, the public denoting all the users of the system. However an intermediate level is often introduced to specify the public at a particular site, e.g., denoted by public@site_s. More precise groups can be defined by the command

```
DEFINE GROUP ⟨group_id⟩ AS ⟨list of subject_ids⟩
```

The management of groups in a distributed environment poses some problems since the subjects of a group can be located at various sites and access to an object may be granted to several groups, which are themselves distributed. If group information and access rules are fully replicated at all sites, the enforcement of access rights is similar to that of a centralized system. However, maintaining this replication may be expensive. The problem is more difficult if site autonomy (with decentralized control) must be maintained. One solution enforces access rights by performing a remote query to the nodes holding the group definition. Another solution replicates a group definition at each node containing an object that may be accessed by subjects of that group. These solutions tend to decrease the degree of site autonomy.

Enforcing MAC in a distributed environment is made difficult by the possibility of indirect means, called *covert channels*, to access unauthorized data. For instance, consider a simple distributed DBMS architecture with two sites, each managing its database at a single security level, e.g., one site is confidential, while the other is secret. According to the "no write down" rule, an update operation from a subject with secret clearance could only be sent to the secret site. However, according to the "no read up" rule, a read query from the same secret subject could be sent to both the secret and the confidential sites. Since the query sent to the confidential site may contain secret information (e.g., in a select predicate), it is potentially a covert channel. To avoid such covert channels, a solution is to replicate part of the database so that a site at security level l contains all data that a subject at level l can access. For instance, the secret site would replicate confidential data so that it can entirely process secret queries. One problem with this architecture is the overhead of maintaining the consistency of replicas (see Chap. 6 on replication). Furthermore, although there are no covert channels for queries, there may still be covert channels

for update operations because the delays involved in synchronizing transactions may be exploited. The complete support for MAC in distributed database systems, therefore, requires significant extensions to transaction management techniques and to distributed query processing techniques.

3.3 Semantic Integrity Control

Another important and difficult problem for a database system is how to guarantee *database consistency*. A database state is said to be consistent if the database satisfies a set of constraints, called *semantic integrity constraints*. Maintaining a consistent database requires various mechanisms such as concurrency control, reliability, protection, and semantic integrity control, which are provided as part of transaction management. Semantic integrity control ensures database consistency by rejecting update transactions that lead to inconsistent database states, or by activating specific actions on the database state, which compensate for the effects of the update transactions. Note that the updated database must satisfy the set of integrity constraints.

In general, semantic integrity constraints are rules that represent the *knowledge* about the properties of an application. They define static or dynamic application properties that cannot be directly captured by the object and operation concepts of a data model. Thus the concept of an integrity rule is strongly connected with that of a data model in the sense that more semantic information about the application can be captured by means of these rules.

Two main types of integrity constraints can be distinguished: structural constraints and behavioral constraints. *Structural constraints* express basic semantic properties inherent to a model. Examples of such constraints are unique key constraints in the relational model, or one-to-many associations between objects in the object-oriented model. *Behavioral constraints*, on the other hand, regulate the application behavior. Thus they are essential in the database design process. They can express associations between objects, such as inclusion dependency in the relational model, or describe object properties and structures. The increasing variety of database applications and the development of database design aid tools call for powerful integrity constraints that can enrich the data model.

Integrity control appeared with data processing and evolved from procedural methods (in which the controls were embedded in application programs) to declarative methods. Declarative methods have emerged with the relational model to alleviate the problems of program/data dependency, code redundancy, and poor performance of the procedural methods. The idea is to express integrity constraints using assertions of predicate calculus. Thus a set of semantic integrity assertions defines database consistency. This approach allows one to easily declare and modify complex integrity constraints.

The main problem in supporting automatic semantic integrity control is that the cost of checking for constraint violation can be prohibitive. Enforcing integrity constraints is costly because it generally requires access to a large amount of data that are not directly involved in the database updates. The problem is more difficult when constraints are defined over a distributed database.

Various solutions have been investigated to design an integrity manager by combining optimization strategies. Their purpose is to (1) limit the number of constraints that need to be enforced, (2) decrease the number of data accesses to enforce a given constraint in the presence of an update transaction, (3) define a preventive strategy that detects inconsistencies in a way that avoids undoing updates, (4) perform as much integrity control as possible at compile time. A few of these solutions have been implemented, but they suffer from a lack of generality. Either they are restricted to a small set of assertions (more general constraints would have a prohibitive checking cost) or they only support restricted programs (e.g., single-tuple updates).

In this section, we present the solutions for semantic integrity control first in centralized systems and then in distributed systems. Since our context is the relational model, we consider only declarative methods.

3.3.1 Centralized Semantic Integrity Control

A semantic integrity manager has two main components: a language for expressing and manipulating integrity constraints, and an enforcement mechanism that performs specific actions to enforce database integrity upon update transactions.

3.3.1.1 Specification of Integrity Constraints

Integrity constraints are manipulated by the database administrator using a high-level language. In this section, we illustrate a declarative language for specifying integrity constraints. This language is much in the spirit of the standard SQL language, but with more generality. It allows one to specify, read, or drop integrity constraints. These constraints can be defined either at relation creation time or at any time, even if the relation already contains tuples. In both cases, however, the syntax is almost the same. For simplicity and without lack of generality, we assume that the effect of integrity constraint violation is to abort the violating transactions. However, the SQL standard provides means to express the propagation of update actions to correct inconsistencies, with the `CASCADING` clause within the constraint declaration. More generally, *triggers* (event-condition-action rules) can be used to automatically propagate updates, and thus to maintain semantic integrity. However, triggers are quite powerful and thus more difficult to support efficiently than specific integrity constraints.

In relational database systems, integrity constraints are defined as assertions. An assertion is a particular expression of tuple relational calculus, in which each variable is either universally (∀) or existentially (∃) quantified. Thus an assertion can be seen as a query qualification that is either true or false for each tuple in the Cartesian product of the relations determined by the tuple variables. We can distinguish between three types of integrity constraints: predefined, precondition, or general constraints.

Predefined constraints are based on simple keywords. Through them, it is possible to express concisely the more common constraints of the relational model, such as nonnull attribute, unique key, foreign key, or functional dependency. Examples 3.11–3.14 demonstrate predefined constraints.

Example 3.11 Employee number in relation EMP cannot be null.

```
ENO NOT NULL IN EMP
```

♦

Example 3.12 The pair (ENO, PNO) is the unique key in relation ASG.

```
(ENO, PNO) UNIQUE IN ASG
```

♦

Example 3.13 The project number PNO in relation ASG is a foreign key matching the primary key PNO of relation PROJ. In other words, a project referred to in relation ASG must exist in relation PROJ.

```
PNO IN ASG REFERENCES PNO IN PROJ
```

♦

Example 3.14 The employee number functionally determines the employee name.

```
ENO IN EMP DETERMINES ENAME
```

♦

Precondition constraints express conditions that must be satisfied by all tuples in a relation for a given update type. The update type, which might be **INSERT**, **DELETE**, or **MODIFY**, permits restricting the integrity control. To identify in the constraint definition the tuples that are subject to update, two variables, NEW and OLD, are implicitly defined. They range over new tuples (to be inserted) and old tuples (to be deleted), respectively. Precondition constraints can be expressed with the SQL **CHECK** statement enriched with the ability to specify the update type. The syntax of the **CHECK** statement is

```
CHECK ON ⟨relation name⟩ WHEN ⟨change type⟩
   (⟨qualification over relation name⟩)
```

Examples of precondition constraints are the following:

Example 3.15 The budget of a project is between 500K and 1000K.

```
CHECK ON PROJ (BUDGET+ >= 500000 AND  BUDGET <= 1000000)
```

♦

Example 3.16 Only the tuples whose budget is 0 may be deleted.

```
CHECK ON PROJ WHEN DELETE (BUDGET = 0)
```

♦

Example 3.17 The budget of a project can only increase.

```
CHECK ON PROJ (NEW.BUDGET > OLD.BUDGET AND
   NEW.PNO = OLD.PNO)
```

♦

General constraints are formulas of tuple relational calculus where all variables are quantified. The database system must ensure that those formulas are always true. General constraints are more concise than precompiled constraints since the former may involve more than one relation. For instance, at least three precompiled constraints are necessary to express a general constraint on three relations. A general constraint may be expressed with the following syntax:

```
CHECK ON list of ⟨variable name⟩:⟨relation name⟩,
   (⟨qualification⟩)
```

Examples of general constraints are given below.

Example 3.18 The constraint of Example 3.8 may also be expressed as

```
CHECK ON e1:EMP, e2:EMP
   (e1.ENAME = e2.ENAME IF e1.ENO = e2.ENO)
```

♦

Example 3.19 The total duration for all employees in the CAD project is less than 100.

```
CHECK ON g:ASG, j:PROJ (SUM(g.DUR WHERE
   g.PNO=j.PNO)<100 IF j.PNAME="CAD/CAM")
```

♦

3.3.1.2 Integrity Enforcement

We now focus on enforcing semantic integrity that consists of rejecting update transactions that violate some integrity constraints. A constraint is violated when it becomes false in the new database state produced by the update transaction. A major difficulty in designing an integrity manager is finding efficient enforcement algorithms. Two basic methods permit the rejection of inconsistent update transactions. The first one is based on the *detection* of inconsistencies. The update transaction u is executed, causing a change of the database state D to D_u. The enforcement algorithm verifies, by applying tests derived from these constraints, that all relevant constraints hold in state D_u. If state D_u is inconsistent, the DBMS can try either to reach another consistent state, D'_u, by modifying D_u with compensation actions or to restore state D by undoing u. Since these tests are applied *after* having changed the database state, they are generally called *posttests*. This approach may be inefficient if a large amount of work (the update of D) must be undone in the case of an integrity failure.

The second method is based on the *prevention* of inconsistencies. An update is executed only if it changes the database state to a consistent state. The tuples subject to the update transaction are either directly available (in the case of insert) or must be retrieved from the database (in the case of deletion or modification). The enforcement algorithm verifies that all relevant constraints will hold after updating those tuples. This is generally done by applying to those tuples tests that are derived from the integrity constraints. Given that these tests are applied *before* the database state is changed, they are generally called *pretests*. The preventive approach is more efficient than the detection approach since updates never need to be undone because of integrity violation.

The query modification algorithm is an example of a preventive method that is particularly efficient at enforcing domain constraints. It adds the assertion qualification to the query qualification by an AND operator so that the modified query can enforce integrity.

Example 3.20 The query for increasing the budget of the CAD/CAM project by 10%, which would be specified as

```
UPDATE PROJ
SET    BUDGET = BUDGET*1.1
WHERE  PNAME= "CAD/CAM"
```

will be transformed into the following query in order to enforce the domain constraint discussed in Example 3.9.

```
UPDATE PROJ
SET    BUDGET = BUDGET * 1.1
WHERE  PNAME= "CAD/CAM"
AND    NEW.BUDGET ≥ 500000
AND    NEW.BUDGET ≤ 1000000
```

♦

The query modification algorithm, which is well-known for its elegance, produces pretests at runtime by ANDing the assertion predicates with the update predicates of each instruction of the transaction. However, the algorithm only applies to tuple calculus formulas and can be specified as follows. Consider the assertion $(\forall x \in \mathtt{R})F(x)$, where F is a tuple calculus expression in which x is the only free variable. An update of R can be written as $(\forall x \in \mathtt{R})(Q(x) \Rightarrow update(x))$, where Q is a tuple calculus expression whose only free variable is x. Roughly speaking, the query modification consists in generating the update $(\forall x \in \mathtt{R})((Q(x)$ and $F(x)) \Rightarrow$update$(x))$. Thus x needs to be universally quantified.

Example 3.21 The foreign key constraint of Example 3.13 that can be rewritten as

$$\forall g \in \mathtt{ASG}, \exists j \in \mathtt{PROJ} : g.\mathtt{PNO} = j.\mathtt{PNO}$$

could not be processed by query modification because the variable j is not universally quantified. ♦

To handle more general constraints, pretests can be generated at constraint definition time, and enforced at runtime when updates occur. In the rest of this section, we present a general method. This method is based on the production, at constraint definition time, of pretests that are used subsequently to prevent the introduction of inconsistencies in the database. This is a general preventive method that handles the entire set of constraints introduced in the preceding section. It significantly reduces the proportion of the database that must be checked when enforcing assertions in the presence of updates. This is a major advantage when applied to a distributed environment.

The definition of pretest uses differential relations, as defined in Sect. 3.1.3. A *pretest* is a triple (R, U, C) in which R is a relation, U is an update type, and C is an assertion ranging over the differential relation(s) involved in an update of type U. When an integrity constraint I is defined, a set of pretests may be produced for the relations used by I. Whenever a relation involved in I is updated by a transaction u, the pretests that must be checked to enforce I are only those defined on I for the update type of u. The performance advantages of this approach are twofold. First, the number of assertions to enforce is minimized since only the pretests of type u need be checked. Second, the cost of enforcing a pretest is less than that of enforcing I since differential relations are, in general, much smaller than the base relations.

Pretests may be obtained by applying transformation rules to the original assertion. These rules are based on a syntactic analysis of the assertion and quantifier permutations. They permit the substitution of differential relations for base relations. Since the pretests are simpler than the original ones, the process that generates them is called *simplification*.

Example 3.22 Consider the modified expression of the foreign key constraint in Example 3.15. The pretests associated with this constraint are

$$(\mathtt{ASG}, \textbf{INSERT}, C_1), (\mathtt{PROJ}, \textbf{DELETE}, C_2), \text{ and } (\mathtt{PROJ}, \textbf{MODIFY}, C_3),$$

where C_1 is

$$\forall \text{ NEW} \in \texttt{ASG}^{+}, \exists j \in \texttt{PROJ}: \text{NEW.}\texttt{PNO} = j.\texttt{PNO}$$

C_2 is

$$\forall g \in \texttt{ASG}, \forall \text{ OLD} \in \texttt{PROJ}^{-} : g.\texttt{PNO} \neq \text{OLD.}\texttt{PNO}$$

and C_3 is

$$\forall g \in \texttt{ASG}, \forall \text{ OLD} \in \texttt{PROJ}^{-} \exists \text{ NEW} \in \texttt{PROJ}^{+} : g.\texttt{PNO} \neq \text{OLD.}\texttt{PNO} \text{ OR } \text{OLD.}\texttt{PNO} = \text{NEW.}\texttt{PNO}$$ ♦

The advantage provided by such pretests is obvious. For instance, a deletion on relation ASG does not incur any assertion checking.

The enforcement algorithm makes use of pretests and is specialized according to the class of the assertions. Three classes of constraints are distinguished: single-relation constraints, multirelation constraints, and constraints involving aggregate functions.

Let us now summarize the enforcement algorithm. Recall that an update transaction updates all tuples of relation R that satisfy some qualification. The algorithm acts in two steps. The first step generates the differential relations $\texttt{R}^{+}$ and $\texttt{R}^{-}$ from R. The second step simply consists of retrieving the tuples of $\texttt{R}^{+}$ and $\texttt{R}^{-}$, which do not satisfy the pretests. If no tuples are retrieved, the constraint is valid. Otherwise, it is violated.

Example 3.23 Suppose there is a deletion on PROJ. Enforcing (PROJ, **DELETE**, C_2) consists in generating the following statement:

result $\leftarrow$ retrieve all tuples of $\texttt{PROJ}^{-}$ where $\neg(C_2)$

Then, if the result is empty, the assertion is verified by the update and consistency is preserved. ♦

3.3.2 Distributed Semantic Integrity Control

In this section, we present algorithms for ensuring the semantic integrity of distributed databases. They are extensions of the simplification method discussed previously. In what follows, we assume global transaction management capabilities, as provided for homogeneous systems or multidatabase systems. Thus, the two main problems of designing an integrity manager for such a distributed DBMS are the definition and storage of constraints, and their enforcement. We will also discuss the issues involved in integrity constraint checking when there is no global transaction support.

3.3.2.1 Definition of Distributed Integrity Constraints

An integrity constraint is supposed to be expressed in predicate calculus. Each assertion is seen as a query qualification that is either true or false for each tuple in the Cartesian product of the relations determined by the tuple variables. Since assertions can involve data stored at different sites, the storage of the constraints must be decided so as to minimize the cost of integrity checking. There is a strategy based on a taxonomy of integrity constraints that distinguishes three classes:

1. *Individual constraints*: single-relation single-variable constraints. They refer only to tuples to be updated independently of the rest of the database. For instance, the domain constraint of Example 3.15 is an individual assertion.
2. *Set-oriented constraints*: include single-relation multivariable constraints such as functional dependency (Example 3.14) and multirelation multivariable constraints such as foreign key constraints (Example 3.13).
3. *Constraints involving aggregates*: require special processing because of the cost of evaluating the aggregates. The assertion in Example 3.19 is representative of a constraint of this class.

The definition of a new integrity constraint can be started at one of the sites that store the relations involved in the assertion. Remember that the relations can be fragmented. A fragmentation predicate is a particular case of assertion of class 1. Different fragments of the same relation can be located at different sites. Thus, defining an integrity assertion becomes a distributed operation, which is done in two steps. The first step is to transform the high-level assertions into pretests, using the techniques discussed in the preceding section. The next step is to store pretests according to the class of constraints. Constraints of class 3 are treated like those of class 1 or 2, depending on whether they are individual or set-oriented.

Individual Constraints

The constraint definition is sent to all other sites that contain fragments of the relation involved in the constraint. The constraint must be compatible with the relation data at each site. Compatibility can be checked at two levels: predicate and data. First, predicate compatibility is verified by comparing the constraint predicate with the fragment predicate. A constraint C is not compatible with a fragment predicate p if "C is true" implies that "p is false," and is compatible with p otherwise. If noncompatibility is found at one of the sites, the constraint definition is globally rejected because tuples of that fragment do not satisfy the integrity constraints. Second, if predicate compatibility has been found, the constraint is tested against the instance of the fragment. If it is not satisfied by that instance, the constraint is also globally rejected. If compatibility is found, the constraint is stored at each site. Note that the compatibility checks are performed only for pretests whose update type is "insert" (the tuples in the fragments are considered "inserted").

Example 3.24 Consider relation EMP, horizontally fragmented across three sites using the predicates

p_1 : $0 \leq$ ENO $<$ "E3"
p_2 : "E3" $\leq$ ENO $\leq$ "E6"
p_3 : ENO $>$ "E6"

and the domain constraint C: ENO $<$ "E4." Constraint C is compatible with p_1 (if C is true, p_1 is true) and p_2 (if C is true, p_2 is not necessarily false), but not with p_3 (if C is true, then p_3 is false). Therefore, constraint C should be globally rejected because the tuples at site 3 cannot satisfy C, and thus relation EMP does not satisfy C. ♦

Set-Oriented Constraints

Set-oriented constraints are multivariable; that is, they involve join predicates. Although the assertion predicate may be multirelation, a pretest is associated with a single relation. Therefore, the constraint definition can be sent to all the sites that store a fragment referenced by these variables. Compatibility checking also involves fragments of the relation used in the join predicate. Predicate compatibility is useless here, because it is impossible to infer that a fragment predicate p is false if the constraint C (based on a join predicate) is true. Therefore C must be checked for compatibility against the data. This compatibility check basically requires joining each fragment of the relation, say R, with all fragments of the other relation, say S, involved in the constraint predicate. This operation may be expensive and, as any join, should be optimized by the distributed query processor. Three cases, given in increasing cost of checking, can occur:

1. The fragmentation of R is derived (see Chap. 2) from that of S based on a semijoin on the attribute used in the assertion join predicate.
2. S is fragmented on join attribute.
3. S is not fragmented on join attribute.

In the first case, compatibility checking is cheap since the tuple of S matching a tuple of R is at the same site. In the second case, each tuple of R must be compared with at most one fragment of S, because the join attribute value of the tuple of R can be used to find the site of the corresponding fragment of S. In the third case, each tuple of R must be compared with all fragments of S. If compatibility is found for all tuples of R, the constraint can be stored at each site.

Example 3.25 Consider the set-oriented pretest (ASG, **INSERT**, C_1) defined in Example 3.16, where C_1 is

$\forall$ NEW $\in$ ASG$^+$, $\exists j \in$ PROJ: NEW.PNO $= j$.PNO

Let us consider the following three cases:

1. ASG is fragmented using the predicate

$$\text{ASG} \ltimes_{\text{PNO}} \text{PROJ}_i$$

where PROJ_I is a fragment of relation PROJ. In this case each tuple NEW of ASG has been placed at the same site as tuple j such that NEW.PNO = j.PNO. Since the fragmentation predicate is identical to that of C_1, compatibility checking does not incur communication.

2. PROJ is horizontally fragmented based on the two predicates

p_1 : PNO $<$ "P3"
p_2 : PNO $\geq$ "P3"

In this case each tuple NEW of ASG is compared with either fragment PROJ_1, if NEW.PNO $<$ "P3," or fragment PROJ_2, if NEW.PNO $\geq$ "P3."

3. PROJ is horizontally fragmented based on the two predicates

p_1 : PNAME = "CAD/CAM"
p_2 : PNAME $\neq$ "CAD/CAM"

In this case each tuple of ASG must be compared with both fragments PROJ_1 and PROJ_2.

♦

3.3.2.2 Enforcement of Distributed Integrity Constraints

Enforcing distributed integrity constraints is more complex than in centralized DBMSs, even with global transaction management support. The main problem is to decide where (at which site) to enforce the integrity constraints. The choice depends on the class of the constraint, the type of update, and the nature of the site where the update is issued (called the *query master site*). This site may, or may not, store the updated relation or some of the relations involved in the integrity constraints. The critical parameter we consider is the cost of transferring data, including messages, from one site to another. We now discuss the different types of strategies according to these criteria.

Individual Constraints

Two cases are considered. If the update transaction is an insert statement, all the tuples to be inserted are explicitly provided by the user. In this case, all individual constraints can be enforced at the site where the update is submitted. If the update is a qualified update (delete or modify statements), it is sent to the sites storing the relation that will be updated. The query processor executes the update qualification for each fragment. The resulting tuples at each site are combined into one temporary

relation in the case of a delete statement, or two, in the case of a modify statement (i.e., $\mathtt{R}^+$ and $\mathtt{R}^-$). Each site involved in the distributed update enforces the assertions relevant at that site (e.g., domain constraints when it is a delete).

Set-Oriented Constraints

We first study single-relation constraints by means of an example. Consider the functional dependency of Example 3.14. The pretest associated with update type **INSERT** is

$$(\mathtt{EMP},\ \ \textbf{INSERT}, C)$$

where C is

$$(\forall \mathrm{e} \in \mathtt{EMP})(\forall \mathrm{NEW1} \in \mathtt{EMP})(\forall \mathrm{NEW2} \in \mathtt{EMP}) \tag{1}$$

$$(\mathrm{NEW1}.\mathtt{ENO} = \mathrm{e}.\mathtt{ENO} \Rightarrow \mathrm{NEW1}.\mathtt{ENAME} = \mathrm{e}.\mathtt{ENAME}) \wedge \tag{2}$$

$$(\mathrm{NEW1}.\mathtt{ENO} = \mathrm{NEW2}.\mathtt{ENO} \Rightarrow \mathrm{NEW1}.\mathtt{ENAME} = \mathrm{NEW2}.\mathtt{ENAME} \tag{3}$$

The second line in the definition of C checks the constraint between the inserted tuples (NEW1) and the existing ones (e), while the third checks it between the inserted tuples themselves. That is why two variables (NEW1 and NEW2) are declared in the first line.

Consider now an update of EMP. First, the update qualification is executed by the query processor and returns one or two temporary relations, as in the case of individual constraints. These temporary relations are then sent to all sites storing EMP. Assume that the update is an **INSERT** statement. Then each site storing a fragment of EMP will enforce constraint C described above. Because e in C is universally quantified, C must be satisfied by the local data at each site. This is due to the fact that $\forall x \in \{a_1, \ldots, a_n\} f(x)$ is equivalent to $[f(a_1) \wedge f(a_2) \wedge \cdots \wedge f(a_n)]$. Thus the site where the update is submitted must receive for each site a message indicating that this constraint is satisfied and that it is a condition for all sites. If the constraint is not true for one site, this site sends an error message indicating that the constraint has been violated. The update is then invalid, and it is the responsibility of the integrity manager to decide if the entire transaction must be rejected using the global transaction manager.

Let us now consider multirelation constraints. For the sake of clarity, we assume that the integrity constraints do not have more than one tuple variable ranging over the same relation. Note that this is likely to be the most frequent case. As with single-relation constraints, the update is computed at the site where it was submitted. The enforcement is done at the query master site, using the ENFORCE algorithm given in Algorithm 3.2.

Algorithm 3.2: ENFORCE

Input: U: update type; R: relation
begin
  retrieve all compiled assertions (R, U, C_i)
  inconsistent ← **false**
  for *each compiled assertion* **do**
    result ← all new (respectively, old), tuples of R where $\neg(C_i)$
  end for
  if *card(result)* $\neq$ 0 **then**
    inconsistent ← **true**
  end if
  if ¬*inconsistent* **then**
    send the tuples to update to all the sites storing fragments of R
  else
    reject the update
  end if
end

Example 3.26 We illustrate this algorithm through an example based on the foreign key constraint of Example 3.13. Let u be an insertion of a new tuple into ASG. The previous algorithm uses the pretest (ASG, **INSERT**, C), where C is

$$\forall \text{NEW} \in \text{ASG}^+, \exists j \in \text{PROJ}: \text{NEW.PNO} = j.\text{PNO}$$

For this constraint, the retrieval statement is to retrieve all new tuples in ASG^+, where C is not true. This statement can be expressed in SQL as

```
SELECT NEW.*
FROM  ASG+ NEW, PROJ
WHERE  COUNT(PROJ.PNO WHERE NEW.PNO = PROJ.PNO)=0
```

Note that NEW.* denotes all the attributes of ASG^+. ♦

Thus the strategy is to send new tuples to sites storing relation PROJ in order to perform the joins, and then to centralize all results at the query master site. For each site storing a fragment of PROJ, the site joins the fragment with ASG^+ and sends the result to the query master site, which performs the union of all results. If the union is empty, the database is consistent. Otherwise, the update leads to an inconsistent state and should be rejected, using the global transaction manager. More sophisticated strategies that notify or compensate inconsistencies can also be devised.

Constraints Involving Aggregates

These constraints are among the most costly to test because they require the calculation of the aggregate functions. The aggregate functions generally manipulated are **MIN**, **MAX**, **SUM**, and **COUNT**. Each aggregate function contains a projection part and

a selection part. To enforce these constraints efficiently, it is possible to produce pretests that isolate redundant data which can be stored at each site storing the associated relation. This data is what we called *materialized views* in Sect. 3.1.2.

3.3.2.3 Summary of Distributed Integrity Control

The main problem of distributed integrity control is that the communication and processing costs of enforcing distributed constraints can be prohibitive. The two main issues in designing a distributed integrity manager are the definition of the distributed assertions and of the enforcement algorithms that minimize the cost of distributed integrity checking. We have shown in this chapter that distributed integrity control can be completely achieved, by extending a preventive method based on the compilation of semantic integrity constraints into pretests. The method is general since all types of constraints expressed in first-order predicate logic can be handled. It is compatible with fragment definition and minimizes intersite communication. A better performance of distributed integrity enforcement can be obtained if fragments are defined carefully. Therefore, the specification of distributed integrity constraints is an important aspect of the distributed database design process.

The method described above assumes global transaction support. Without global transaction support as in some loosely coupled multidatabase systems, the problem is more difficult. First, the interface between the constraint manager and the component DBMS is different since constraint checking can no longer be part of the global transaction validation. Instead, the component DBMSs should notify the integrity manager to perform constraint checking after some events, e.g., as a result of local transactions' commitments. This can be done using triggers whose events are updates to relations involved in global constraints. Second, if a global constraint violation is detected, since there is no way to specify global aborts, specific correcting transactions should be provided to produce global database states that are consistent. The solution is to have a family of protocols for global integrity checking. The root of the family is a simple strategy, based on the computation of differential relations (as in the previous method), which is shown to be safe (correctly identifies constraint violations) but inaccurate (may raise an error event though there is no constraint violation). Inaccuracy is due to the fact that producing differential relations at different times at different sites may yield *phantom* states for the global database, i.e., states that never existed. Extensions of the basic protocol with either timestamping or using local transaction commands are proposed to solve that problem.

3.4 Conclusion

Data control includes view management, access control, and semantic integrity control. In relational DBMSs, these functions can be uniformly achieved by enforcing rules that specify data manipulation control. Solutions initially designed for handling these functions in centralized systems have been significantly extended and enriched for distributed systems, in particular, support for materialized views and group-based discretionary access control. Semantic integrity control has received less attention and is generally not well supported by distributed DBMS products.

Full data control is more complex and costly in terms of performance in distributed systems. The two main issues for efficiently performing data control are the definition and storage of the rules (site selection) and the design of enforcement algorithms which minimize communication costs. The problem is difficult since increased functionality (and generality) tends to increase site communication. The problem is simplified if control rules are fully replicated at all sites and harder if site autonomy is to be preserved. In addition, specific optimizations can be done to minimize the cost of data control but with extra overhead such as managing materialized views or redundant data. Thus the specification of distributed data control must be included in the distributed database design so that the cost of control for update programs is also considered.

3.5 Bibliographic Notes

Data control is well-understood in centralized systems and all major DBMSs provide extensive support for it. Research on data control in distributed systems started in the mid-1980s with the R* project at IBM Research and has increased much since then to address new important applications such as data warehousing or data integration.

Most of the work on view management has concerned updates through views and support for materialized views. The two basic papers on centralized view management are [Chamberlin et al. 1975] and [Stonebraker 1975]. The first reference presents an integrated solution for view and authorization management in the System R project at IBM Research. The second reference describes the query modification technique proposed in the INGRES project at UC Berkeley for uniformly handling views, authorizations, and semantic integrity control. This method was presented in Sect. 3.1.

Theoretical solutions to the problem of view updates are given in [Bancilhon and Spyratos 1981, Dayal and Bernstein 1978, Keller 1982]. In the seminal paper on view update semantics [Bancilhon and Spyratos 1981], the authors formalize the view invariance property after updating, and show how a large class of views including joins can be updated. Semantic information about the base relations is

particularly useful for finding unique propagation of updates. However, the current commercial systems are very restrictive in supporting updates through views.

Materialized views have received much attention in the context of data warehousing. The notion of snapshot for optimizing view derivation in distributed database systems is due to [Adiba and Lindsay 1980], and generalized in Adiba [1981] as a unified mechanism for managing views and snapshots, as well as fragmented and replicated data. Self-maintainability of materialized views with respect to the kind of updates (insertion, deletion, or modification) is addressed in [Gupta et al. 1996]. A thorough paper on materialized view management can be found in Gupta and Mumick [1999], with the main techniques to perform incremental maintenance of materialized views. The counting algorithm which we presented in Sect. 3.1.3 was proposed in [Gupta et al. 1993]. We introduced two recent important optimizations that have been proposed to significantly reduce the maintenance time of the counting algorithm, following the formalism of generalized multiset relations [Koch 2010]. The first optimization is to materialize views representing subqueries of the input query [Koch et al. 2014, Berkholz et al. 2017, Nikolic and Olteanu 2018]. The second optimization exploits the skew in the data [Kara et al. 2019].

Security in computer systems in general is presented in [Hoffman 1977]. Security in centralized database systems is presented in [Lunt and Fernández 1990, Castano et al. 1995]. Discretionary access control (DAC) in distributed systems has first received much attention in the context of the R* project. The access control mechanism of System R [Griffiths and Wade 1976] is extended in [Wilms and Lindsay 1981] to handle groups of users and to run in a distributed environment. Mandatory access control (MAC) for distributed DBMS has recently gained much interest. The seminal paper on MAC is the Bell and LaPadula model originally designed for operating system security [Bell and Lapuda 1976]. MAC for databases is described in [Lunt and Fernández 1990, Jajodia and Sandhu 1991]. A good introduction to multilevel security in relational DBMS can be found in [Rjaibi 2004]. Transaction management in multilevel secure DBMS is addressed in [Ray et al. 2000, Jajodia et al. 2001]. Extensions of MAC for distributed DBMS are proposed in [Thuraisingham 2001]. Role-based access control (RBAC) [Ferraiolo and Kuhn 1992] extends DAC and MAC by adding roles, as a level of independence between subjects and objects. Hippocratic databases [Sandhu et al. 1996] associate purpose information with data, i.e., the reasons for data collection and access.

The content of Sect. 3.3 comes largely from the work on semantic integrity control described in [Simon and Valduriez 1984, 1986, 1987]. In particular, [Simon and Valduriez 1986] extend a preventive strategy for centralized integrity control based on pretests to run in a distributed environment, assuming global transaction support. The initial idea of declarative methods, that is, to use assertions of predicate logic to specify integrity constraints, is due to [Florentin 1974]. The most important declarative methods are in [Bernstein et al. 1980a, Blaustein 1981, Nicolas 1982, Simon and Valduriez 1984, Stonebraker 1975]. The notion of concrete views for storing redundant data is described in [Bernstein and Blaustein 1982]. Note that concrete views are useful in optimizing the enforcement of constraints involving aggregates. Civelek et al. [1988], Sheth et al. [1988b], and Sheth et al.

[1988a] describe systems and tools for data control, particularly view management. Semantic integrity checking in loosely coupled multidatabase systems without global transaction support is addressed in [Grefen and Widom 1997].

Exercises

Problem 3.1 Define in SQL-like syntax a view of the engineering database V(ENO, ENAME, PNO, RESP), where the duration is 24. Is view V updatable? Assume that relations EMP and ASG are horizontally fragmented based on access frequencies as follows:

Site 1	Site 2	Site 3
EMP_1	EMP_2	
	ASG_1	ASG_2

where

$$EMP_1 = \sigma_{TITLE \neq \text{“Engineer”}}(EMP)$$
$$EMP_2 = \sigma_{TITLE = \text{“Engineer”}}(EMP)$$
$$ASG_1 = \sigma_{0<DUR<36}(ASG)$$
$$ASG_2 = \sigma_{DUR \geq 36}(ASG)$$

At which site(s) should the definition of V be stored without being fully replicated, to increase locality of reference?

Problem 3.2 Express the following query: names of employees in view V who work on the CAD/CAM project.

Problem 3.3 (*) Assume that relation PROJ is horizontally fragmented as

$$PROJ_1 = \sigma_{PNAME = \text{“CAD/CAM”}}(PROJ)$$
$$PROJ_2 = \sigma_{PNAME \neq \text{“CAD/CAM”}}(PROJ)$$

Modify the query obtained in Problem 3.2 to a query expressed on the fragments.

Problem 3.4 (**) Propose a distributed algorithm to efficiently refresh a snapshot at one site derived by projection from a relation horizontally fragmented at two other sites. Give an example query on the view and base relations which produces an inconsistent result.

Problem 3.5 (*) Consider the view EG of Example 3.5 which uses relations EMP and ASG as base data and assume its state is derived from that of Example 3.1, so that EG has 9 tuples (see Fig. 3.4). Assume that tuple ⟨E3, P3, Consultant, 10⟩ from ASG is updated to ⟨E3, P3, Engineer, 10⟩. Apply the basic counting algorithm for

refreshing the view EG. What projected attributes should be added to view EG to make it self-maintainable?

Problem 3.6 Propose a relation schema for storing the access rights associated with user groups in a distributed database catalog, and give a fragmentation scheme for that relation, assuming that all members of a group are at the same site.

Problem 3.7 (**) Give an algorithm for executing the **REVOKE** statement in a distributed DBMS, assuming that the **GRANT** privilege can be granted only to a group of users where all its members are at the same site.

Problem 3.8 (**) Consider the multilevel relation PROJ** in Fig. 3.8. Assuming that there are only two classification levels for attributes (S and C), propose an allocation of PROJ** on two sites using fragmentation and replication that avoids covert channels on read queries. Discuss the constraints on updates for this allocation to work.

Problem 3.9 Using the integrity constraint specification language of this chapter, express an integrity constraint which states that the duration spent in a project cannot exceed 48 months.

Problem 3.10 (*) Define the pretests associated with integrity constraints covered in Examples 3.11–3.14.

Problem 3.11 Assume the following vertical fragmentation of relations EMP, ASG, and PROJ:

Site 1	Site 2	Site 3	Site 4
EMP_1	EMP_2		
	PROJ_1	PROJ_2	
		ASG_1	ASG_2

where

$$
\begin{aligned}
\text{EMP}_1 &= \Pi_{\text{ENO, ENAME}}(\text{EMP}) \\
\text{EMP}_2 &= \Pi_{\text{ENO, TITLE}}(\text{EMP}) \\
\text{PROJ}_1 &= \Pi_{\text{PNO, PNAME}}(\text{PROJ}) \\
\text{PROJ}_2 &= \Pi_{\text{PNO, BUDGET}}(\text{PROJ}) \\
\text{ASG}_1 &= \Pi_{\text{ENO, PNO, RESP}}(\text{ASG}) \\
\text{ASG}_2 &= \Pi_{\text{ENO, PNO, DUR}}(\text{ASG})
\end{aligned}
$$

Where should the pretests obtained in Problem 3.9 be stored?

Problem 3.12 (**) Consider the following set-oriented constraint:

```
CHECK ON e:EMP, a:ASG
  (e.ENO = a.ENO AND (e.TITLE = "Programmer")
  IF a.RESP = "Programmer")
```

What does it mean? Assuming that EMP and ASG are allocated as in the previous exercise, define the corresponding pretests and their storage. Apply algorithm ENFORCE for an update of type **INSERT** in ASG.

Problem 3.13 (**) Assume a distributed multidatabase system with no global transaction support. Assume also that there are two sites, each with a (different) EMP relation and an integrity manager that communicates with the component DBMS. Suppose that we want to have a global unique key constraint on EMP. Propose a simple strategy using differential relations to check this constraint. Discuss the possible actions when a constraint is violated.

Chapter 4
Distributed Query Processing

By hiding the low-level details about the physical organization of the data, relational database languages allow the expression of complex queries in a concise and simple manner. In particular, to construct the answer to the query, the user does not precisely specify the procedure to follow; this procedure is actually devised by a module, called a *query processor*. This relieves the user from query optimization, a time-consuming task that is best handled by the query processor, since it can exploit a large amount of useful information about the data.

Because it is a critical performance issue, query processing has received (and continues to receive) considerable attention in the context of both centralized and distributed DBMSs. However, the query processing problem is much more difficult in distributed environments, because a larger number of parameters affect the performance of distributed queries. In particular, the relations involved in a distributed query may be fragmented and/or replicated, thereby inducing communication costs. Furthermore, with many sites to access, query response time may become very high.

In this chapter, we give a detailed presentation of query processing in distributed DBMSs. The context chosen is that of relational calculus and relational algebra, because of their generality and wide use in distributed DBMSs. As we saw in Chap. 2, distributed relations are implemented by fragments, with the objective of increasing reference locality, and sometimes parallel execution for the most important queries. The role of a distributed query processor is to map a high-level query (assumed to be expressed in relational calculus) on a distributed database (i.e., a set of global relations) into a sequence of database operators (of relational algebra) on relation fragments. Several important functions characterize this mapping. First, the *calculus query* must be *decomposed* into a sequence of relational operators called an *algebraic query*. Second, the data accessed by the query must be *localized* so that the operators on relations are translated to bear on local data (fragments). Finally, the algebraic query on fragments must be extended with communication operators and *optimized* with respect to a cost function to be minimized. This

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_4

cost function typically refers to computing resources such as disk I/Os, CPUs, and communication networks.

This chapter is organized as follows: Section 4.1 gives an overview of distributed query processing. In Sect. 4.2, we describe data localization, with emphasis on reduction and simplification techniques for the four following types of fragmentation: horizontal, vertical, derived, and hybrid. In Sect. 4.3, we discuss the major optimization issue, which deals with the join ordering in distributed queries. We also examine alternative join strategies based on semijoin. In Sect. 4.4, we discuss the distributed cost model. In Sect. 4.5, we illustrate the use of the techniques in three basic distributed query optimization approaches: dynamic, static, and hybrid. In Sect. 4.5, we discuss adaptive query processing.

We assume that the reader is familiar with basic query processing notions in centralized DBMSs as covered in most undergraduate database courses and textbooks.

4.1 Overview

This section introduces distributed query processing. First, in Sect. 4.1.1, we discuss the query processing problem. Then, in Sect. 4.1.2, we introduce query optimization. Finally, in Sect. 4.1.3, we introduce the different layers of query processing starting from a distributed query down to the execution of operators on local sites.

4.1.1 Query Processing Problem

The main function of a query processor is to transform a high-level query (typically, in relational calculus) into an equivalent lower-level query (typically, in some variation of relational algebra). The low-level query actually implements the execution strategy for the query. The transformation must achieve both correctness and efficiency. It is correct if the low-level query has the same semantics as the original query, that is, if both queries produce the same result. The well-defined mapping from relational calculus to relational algebra makes the correctness issue easy. But producing an efficient execution strategy is more difficult. A relational calculus query may have many equivalent and correct transformations into relational algebra. Since each equivalent execution strategy can lead to very different consumptions of computer resources, the main difficulty is to select the execution strategy that minimizes resource consumption.

Example 4.1 We consider the following subset of the engineering database schema:

```
EMP(ENO, ENAME, TITLE)
ASG(ENO, PNO, RESP, DUR)
```

and the following simple user query:

"Find the names of employees who are managing a project"

The expression of the query in relational calculus using the SQL syntax (with natural join) is

```
SELECT ENAME
FROM   EMP NATURAL JOIN ASG
WHERE  RESP = "Manager"
```

Two equivalent relational algebra queries that are correct transformations of the query above are

$$\Pi_{\text{ENAME}}(\sigma_{\text{RESP="Manager"} \wedge \text{EMP.ENO=ASG.ENO}}(\text{EMP} \times \text{ASG}))$$

and

$$\Pi_{\text{ENAME}}(\bowtie_{\text{ENO}} (\sigma_{\text{RESP="Manager"}}(\text{ASG})))$$

It is intuitively obvious that the second query which avoids the Cartesian product of EMP and ASG consumes much less computing resources than the first, and thus should be retained. ♦

In a distributed system, relational algebra is not enough to express execution strategies. It must be supplemented with operators for exchanging data between sites. Besides the choice of ordering relational algebra operators, the distributed query processor must also select the best sites to process data, and possibly the way data should be transformed. This increases the solution space from which to choose the distributed execution strategy, making distributed query processing significantly more difficult than centralized query processing.

Example 4.2 This example illustrates the importance of site selection and communication for a chosen relational algebra query against a fragmented database. We consider the following query of Example 4.1:

$$\Pi_{\text{ENAME}}(\text{EMP} \bowtie_{\text{ENO}} (\sigma_{\text{RESP="Manager"}}(\text{ASG})))$$

We assume that relations EMP and ASG are horizontally fragmented as follows:

$$\text{EMP}_1 = \sigma_{\text{ENO} \leq \text{"E3"}}(\text{EMP})$$

$$\text{EMP}_2 = \sigma_{\text{ENO} > \text{"E3"}}(\text{EMP})$$

$$\text{ASG}_1 = \sigma_{\text{ENO} \leq \text{"E3"}}(\text{ASG})$$

$$\text{ASG}_2 = \sigma_{\text{ENO} > \text{"E3"}}(\text{ASG})$$

Fragments $\texttt{ASG}_1$, $\texttt{ASG}_2$, $\texttt{EMP}_1$, and $\texttt{EMP}_2$ are stored at sites 1, 2, 3, and 4, respectively, and the result is expected at site 5.

For the sake of simplicity, we ignore the project operator in the following. Two equivalent distributed execution strategies for the above query are shown in Fig. 4.1. An arrow from site i to site j labeled with R indicates that relation R is transferred from site i to site j. Strategy A exploits the fact that relations EMP and ASG are fragmented the same way in order to perform the select and join operator in parallel. Strategy B is a default strategy (always works) that simply centralizes all the operand data at the result site before processing the query.

To evaluate the resource consumption of these two strategies, we use a very simple cost model. We assume that a tuple access, denoted by $tupacc$, is 1 unit (which we leave unspecified) and a tuple transfer, denoted $tuptrans$, is 10 units. We assume that relations EMP and ASG have 400 and 1000 tuples, respectively, and that there are 20 managers in relation ASG. We also assume that data is uniformly distributed among sites. Finally, we assume that relations ASG and EMP are locally clustered on attributes RESP and ENO, respectively. Therefore, there is direct access to tuples of ASG (respectively, EMP) based on the value of attribute RESP (respectively, ENO).

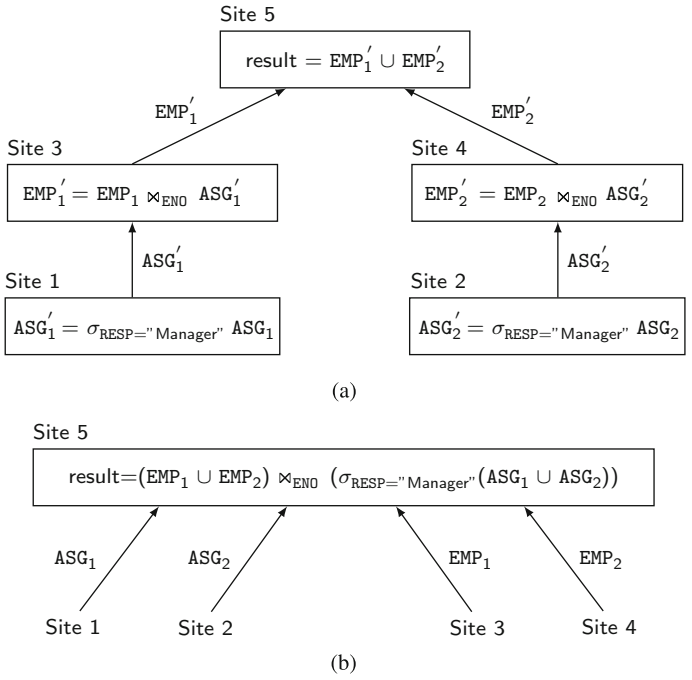

Fig. 4.1 Equivalent distributed execution strategies. (**a**) Strategy A. (**b**) Strategy B

The total cost of strategy A can be derived as follows:

1.	Produce ASG′ by selecting ASG requires $(10 + 10) * tupacc$	=	20
2.	Transfer ASG′ to the sites of EMP requires $(10 + 10) * tuptrans$	=	200
3.	Produce EMP′ by joining ASG′ and EMP requires $(10 + 10) * tupacc * 2$	=	40
4.	Transfer EMP′ to result site requires $(10 + 10) * tuptrans$	=	200
	The total cost is		460

The cost of strategy B can be derived as follows:

1.	Transfer EMP to site 5 requires $400 * tuptrans$	=	4,000
2.	Transfer ASG to site 5 requires $1000 * tuptrans$	=	10,000
3.	Produce ASG′ by selecting ASG requires $1000 * tupacc$	=	1,000
4.	Join EMP and ASG′ requires $400 * 20 * tupacc$	=	8,000
	The total cost is		23,000

In strategy A, the join of ASG′ and EMP (step 3) can exploit the clustered index on ENO of EMP. Thus, EMP is accessed only once for each tuple of ASG′. In strategy B, we assume that the access methods to relations EMP and ASG based on attributes RESP and ENO are lost because of data transfer. This is a reasonable assumption in practice. We assume that the join of EMP and ASG′ in step 4 is done by the default nested loop algorithm (that simply performs the Cartesian product of the two input relations). Strategy A is better by a factor of 50, which is quite significant. Furthermore, it provides better distribution of work among sites. The difference would be even higher if we assumed slower communication and/or higher degree of fragmentation. ♦

4.1.2 Query Optimization

Query optimization refers to the process of producing a query execution plan (QEP), which represents an execution strategy for the query. This QEP minimizes an objective cost function. A query optimizer, the software module that performs query optimization, is usually seen as consisting of three components: a search space, a cost model, and a search strategy.

4.1.2.1 Search Space

The *search space* is the set of alternative execution plans that represent the input query. These plans are equivalent, in the sense that they yield the same result,

but they differ in the execution order of operators and the way these operators are implemented, and therefore in their performance. The search space is obtained by applying transformation rules, such as those for relational algebra.

4.1.2.2 Cost Model

The *cost model* is used to predict the cost of any given execution plan, and to compare equivalent plans so as to choose the best one. To be accurate, the cost model must have good knowledge about the distributed execution environment, using statistics on the data and cost functions.

In a distributed database, statistics typically bear on fragments, and include fragment cardinality and size as well as the size and number of distinct values of each attribute. To minimize the probability of error, more detailed statistics such as histograms of attribute values are sometimes used at the expense of higher management cost. The accuracy of statistics is achieved by periodic updating.

A good measure of cost is the *total cost* that will be incurred in processing the query. Total cost is the sum of all times incurred in processing the operators of the query at various sites and in intersite communication. Another good measure is the *response time* of the query, which is the time elapsed for executing the query. Since operators can be executed in parallel at different sites, the response time of a query may be significantly less than its total cost.

In a distributed database system, the total cost to be minimized includes CPU, I/O, and communication costs. The CPU cost is incurred when performing operators on data in main memory. The I/O cost is the time necessary for disk accesses. This cost can be minimized by reducing the number of disk accesses through fast access methods to the data and efficient use of main memory (buffer management). The communication cost is the time needed for exchanging data between sites participating in the execution of the query. This cost is incurred in processing the messages (formatting/deformatting), and in transmitting the data over the communication network.

The communication cost component is probably the most important factor considered in distributed databases. Most of the early proposals for distributed query optimization assumed that the communication cost largely dominates local processing cost (I/O and CPU cost), and thus ignored the latter. However, modern distributed processing environments have much faster communication networks, whose bandwidth is comparable to that of disks. Therefore, the solution is to have a weighted combination of these three cost components since they all contribute significantly to the total cost of evaluating a query.

In this chapter, we consider relational algebra as a basis to express the output of query processing. Therefore, the complexity of relational algebra operators, which directly affects their execution time, dictates some principles useful to a query processor. These principles can help in choosing the final execution strategy.

The simplest way of defining complexity is in terms of relation cardinalities independent of physical implementation details such as fragmentation and storage

structures. Complexity is $\mathcal{O}(n)$ for unary operators, where n denotes the relation cardinality, if the resulting tuples may be obtained independently of each other. Complexity is $\mathcal{O}(n \log n)$ for binary operators if each tuple of one relation must be compared with each tuple of the other on the basis of the equality of selected attributes. This complexity assumes that tuples of each relation must be sorted on the comparison attributes. However, using hashing and enough memory to hold one hashed relation can reduce the complexity of binary operators to $\mathcal{O}(n)$. Project with duplicate elimination and grouping operators require that each tuple of the relation be compared with each other tuple, and thus also have $\mathcal{O}(n \log n)$ complexity. Finally, complexity is $\mathcal{O}(n^2)$ for the Cartesian product of two relations because each tuple of one relation must be combined with each tuple of the other.

4.1.2.3 Search Strategy

The *search strategy* explores the search space and selects the best plan, using the cost model. It defines which plans are examined and in which order. The details of the distributed environment are captured by the search space and the cost model.

An immediate method for query optimization is to search the solution space, exhaustively predict the cost of each strategy, and select the strategy with minimum cost. Although this method is effective in selecting the best strategy, it may incur a significant processing cost for the optimization itself. The problem is that the solution space can be large; that is, there may be many equivalent strategies, even when the query involves a small number of relations. The problem becomes worse as the number of relations or fragments increases (e.g., becomes greater than 10). Having high optimization cost is not necessarily bad, particularly if query optimization is done once for many subsequent executions of the query.

The most popular search strategy used by query optimizers is *dynamic programming*, which was first proposed in the System R project at IBM Research. It proceeds by *building* plans, starting from base relations, joining one more relation at each step until complete plans are obtained. Dynamic programming builds all possible plans, breadth-first, before it chooses the "best" plan. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are *pruned* (i.e., discarded) as soon as possible.

For very complex queries, making the search space large, *randomized* strategies such as Iterative Improvement and Simulated Annealing can be used. They try to find a very good solution, not necessarily the best one, but with a good trade-off between optimization time and execution time.

Another, complementary solution is to restrict the solution space so that only a few strategies are considered. In both centralized and distributed systems, a common heuristic is to minimize the size of intermediate relations. This can be done by performing unary operators first, and ordering the binary operators by the increasing sizes of their intermediate relations.

A query may be optimized at different times relative to the actual time of query execution. Optimization can be done *statically* before executing the query or *dynamically* as the query is executed. Static query optimization is done at query compilation time. Thus the cost of optimization may be amortized over multiple query executions. Therefore, this timing is appropriate for use with the exhaustive search method. Since the sizes of the intermediate relations of a strategy are not known until runtime, they must be estimated using database statistics. Errors in these estimates can lead to the choice of suboptimal strategies.

Dynamic query optimization proceeds at query execution time. At any point of execution, the choice of the best next operator can be based on accurate knowledge of the results of the operators executed previously. Therefore, database statistics are not needed to estimate the size of intermediate results. However, they may still be useful in choosing the first operators. The main advantage over static query optimization is that the actual sizes of intermediate relations are available to the query processor, thereby minimizing the probability of a bad choice. The main shortcoming is that query optimization, an expensive task, must be repeated for each execution of the query. Therefore, this approach is best for ad hoc queries.

Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates. The approach is basically static, but dynamic query optimization may take place at runtime when a high difference between predicted sizes and actual size of intermediate relations is detected.

4.1.3 Layers Of Query Processing

The problem of query processing can itself be decomposed into several subproblems, corresponding to various layers. In Fig. 4.2, a generic layering scheme for query processing is shown where each layer solves a well-defined subproblem. To simplify the discussion, let us assume a static query processor that does not exploit replicated fragments. The input is a query on global data expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution is hidden. Four main layers are involved in distributed query processing. The first three layers map the input query into a distributed query execution plan (distributed QEP) . They perform the functions of *query decomposition*, *data localization*, and *global query optimization*. Query decomposition and data localization correspond to query rewriting. The first three layers are performed by a central control site and use schema information stored in the global directory. The fourth layer performs *distributed query execution* by executing the plan and returns the answer to the query. It is done by the local sites and the control site. In the remainder of this section, we introduce these four layers.

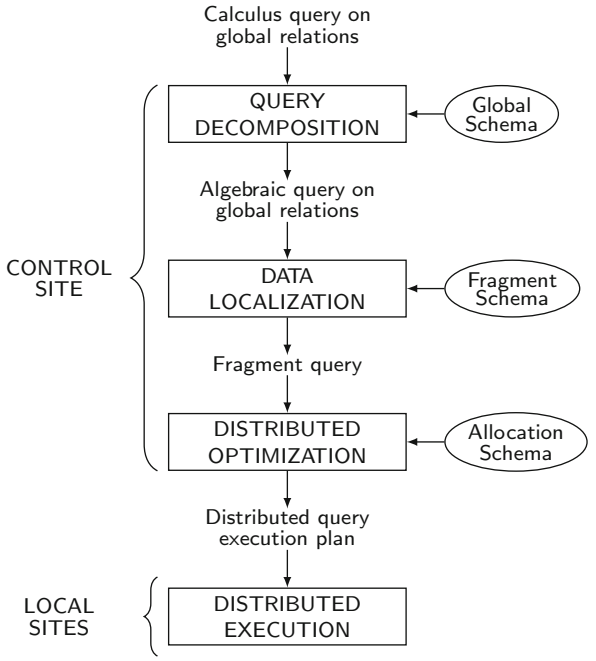

Fig. 4.2 Generic layering scheme for distributed query processing

4.1.3.1 Query Decomposition

The first layer decomposes the calculus query into an algebraic query on global relations. The information needed for this transformation is found in the global conceptual schema describing the global relations. However, the information about data distribution is not used here but in the next layer. The techniques used by this layer are those of a centralized DBMS, so we only briefly remind them in this chapter.

Query decomposition can be viewed as four successive steps. First, the calculus query is rewritten in a *normalized* form that is suitable for subsequent manipulation. Normalization of a query generally involves the manipulation of the query quantifiers and of the query qualification by applying logical operator priority.

Second, the normalized query is *analyzed* semantically so that incorrect queries are detected and rejected as early as possible. A query is semantically incorrect if components of it do not contribute in any way to the generation of the result. In the context of relational calculus, it is not possible to determine the semantic correctness of general queries. However, it is possible to do so for a large class of relational queries, i.e., those which do not contain disjunction and negation. This is based on the representation of the query as a graph, called a *query graph* or *connection graph*. We define this graph for the most useful kinds of queries involving selection, projection, and join operators. In a query graph, one node indicates the result

relation, and any other node indicates an operand relation. An edge between two nodes that are not results represents a join, whereas an edge whose destination node is the result represents a project. Furthermore, a nonresult node may be labeled by a selection or a self-join (join of the relation with itself) predicate. An important subgraph of the query graph is the *join graph*, in which only the joins are considered.

Third, the correct query (still expressed in relational calculus) is *simplified*. One way to simplify a query is to eliminate redundant predicates. Note that redundant queries are likely to arise when a query is the result of system transformations applied to the user query. As seen in Chap. 3, such transformations are used for performing distributed data control (views, protection, and semantic integrity control).

Fourth, the calculus query is *restructured* as an algebraic query. Recall from Sect. 4.1.1 that several algebraic queries can be derived from the same calculus query, and that some algebraic queries are "better" than others. The quality of an algebraic query is defined in terms of expected performance. The traditional way to do this transformation towards a "better" algebraic specification is to start with an initial algebraic query and transform it in order to find a "good" one. The initial algebraic query is derived immediately from the calculus query by translating the predicates and the target statement into relational operators as they appear in the query. This directly translated algebra query is then restructured through transformation rules. The algebraic query generated by this layer is good in the sense that the worse executions are typically avoided. For instance, a relation will be accessed only once, even if there are several select predicates. However, this query is generally far from providing an optimal execution, since information about data distribution and fragment allocation is not used at this layer.

4.1.3.2 Data Localization

The input to the second layer is an algebraic query on global relations. The main role of the second layer is to localize the query's data using data distribution information in the fragment schema. In Chap. 2 we saw that relations are fragmented and stored in disjoint subsets, called fragments, each being stored at a different site. This layer determines which fragments are involved in the query and transforms the distributed query into a query on fragments. Fragmentation is defined through fragmentation rules that can be expressed as relational operators. A global relation can be reconstructed by applying the fragmentation rules, and then deriving a program, called a *materialization program*, of relational algebra operators which then acts on fragments. Localization involves two steps. First, the query is mapped into a fragment query by substituting each relation by its materialization program. Second, the fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition layer. As in the decomposition layer, the final fragment query is generally far from optimal because information regarding fragments is not utilized.

4.1.3.3 Distributed Optimization

The input to the third layer is an algebraic query on fragments, i.e., a fragment query. The goal of query optimization is to find an execution strategy for the query which is close to optimal. An execution strategy for a distributed query can be described with relational algebra operators and *communication primitives* (send/receive operators) for transferring data between sites. The previous layers have already optimized the query, for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as fragment allocation and cardinalities. In addition, communication operators are not yet specified. By permuting the ordering of operators within one query on fragments, many equivalent queries may be found.

Query optimization consists of finding the "best" ordering of operators in the query, including communication operators, that minimizes a cost function. The cost function, often defined in terms of time units, refers to the use of computing resources such as disk, CPU cost, and network. Generally, it is a weighted combination of I/O, CPU, and communication costs. To select the ordering of operators it is necessary to predict execution costs of alternative candidate orderings. Determining execution costs before query execution (i.e., static optimization) is based on fragment statistics and the formulas for estimating the cardinalities of results of relational operators. Thus the optimization decisions depend on the allocation of fragments and available statistics on fragments which are recorder in the allocation schema.

An important aspect of query optimization is *join ordering*, since permutations of the joins within the query may lead to improvements of orders of magnitude. The output of the query optimization layer is an optimized algebraic query with communication operators included on fragments. It is typically represented and saved (for future executions) as a distributed QEP.

4.1.3.4 Distributed Execution

The last layer is performed by all the sites that have fragments involved in the query. Each subquery executing at one site, called a *local query*, is optimized using the local schema of the site and executed. At this time, the algorithms to perform the relational operators may be chosen. Local optimization uses the algorithms of centralized systems.

The classical implementation of relational operators in database systems is based on the iterator model, which provides pipelined parallelism within operator trees. It is a simple pull model that executes operators starting from the root operator node (that produces the result) to the leaf nodes (that access the base relations). Thus, the intermediate results of operators do not need to be materialized as tuples are produced on demand and can be consumed by subsequent operators. However, it requires operators to be implemented in pipeline mode, using an open-next-close interface. Each operator must be implemented as an iterator with three functions:

1. Open(): initializes the operator's internal state, e.g., allocate a hash table;
2. Next(): produces and returns the next result tuple or null;
3. Close(): cleans up all allocated resources, after all tuples have been processed.

Thus, an iterator provides the iteration component of a while loop, i.e., initialization, increment, loop termination condition, and final cleaning. Executing a QEP proceeds as follows. First, execution is initialized by calling Open() on the root operator of the operator tree, which then forwards the Open() call through the entire plan using the operators themselves. Then the root operator iteratively produces its next result record by forwarding the Next() call through the operator tree as needed. Execution terminates when the last Open() call returns "end" to the root operator.

To illustrate the implementation of a relational operator using the open-next-close interface, let us consider the nested loop join operator that performs R ⋈ S on attribute A. The Open() and Next() functions are as follows:

```
Function Open()
     R.Open() ;
     S.Open() ;
     r := R.Next() ;

Function Next()
     while (r ≠ null) do
          (while (s:=S.Next()) ≠ null) do
               if r.A=s.A then return(r,s);
          S.close() ;
          S.open() ;
          r:=R.next() ; )
     return null;
```

It is not always possible to implement an operator in pipelined mode. Such operators are *blocking*, i.e., need to materialize their input data in memory or disk before they can produce any output. Examples of blocking operators are sorting and hash join. If the data is already sorted, then merge-join, grouping, and duplicate elimination can be implemented in pipelined mode.

4.2 Data Localization

Data localization translates an algebraic query on global relations into a fragment query using information stored in the fragment schema. A naive way to do this is to generate a query where each global relation is substituted by its materialization program. This can be viewed as replacing the leaves of the operator trie of the distributed query with subtrees corresponding to the materialized programs. In general, this approach is inefficient because important restructurings and simplifications of the fragment query can still be made. In the remainder of this section, for each type of fragmentation we present *reduction techniques* that generate simpler and

optimized queries. We use the transformation rules and the heuristics, such as pushing unary operators down the trie.

4.2.1 Reduction for Primary Horizontal Fragmentation

The horizontal fragmentation function distributes a relation based on selection predicates. The following example is used in the subsequent discussions.

Example 4.3 Relation EMP(ENO, ENAME, TITLE) can be split into three horizontal fragments EMP_1, EMP_2, and EMP_3, defined as follows:

$$\text{EMP}_1 = \sigma_{\text{ENO} \leq \text{"E3"}}(\text{EMP})$$

$$\text{EMP}_2 = \sigma_{\text{"E3"} < \text{ENO} \leq \text{"E6"}}(\text{EMP})$$

$$\text{EMP}_3 = \sigma_{\text{ENO} > \text{"E6"}}(\text{EMP})$$

The materialization program for a horizontally fragmented relation is the union of the fragments. In our example, we have

$$\text{EMP} = \text{EMP}_1 \cup \text{EMP}_2 \cup \text{EMP}_3$$

Thus the materialized form of any query specified on EMP is obtained by replacing it by $(\text{EMP}_1 \cup \text{EMP}_2 \cup \text{EMP}_3)$. ♦

The reduction of queries on horizontally fragmented relations consists primarily of determining, after restructuring the subtrees, those that will produce empty relations, and removing them. Horizontal fragmentation can be exploited to simplify both selection and join operators.

4.2.1.1 Reduction with Selection

Selections on fragments that have a qualification contradicting the qualification of the fragmentation rule generate empty relations. Given a relation R that has been horizontally fragmented as $\text{R}_1, \text{R}_2, \ldots, \text{R}_w$, where $\text{R}_j = \sigma_{p_j}(\text{R})$, the rule can be stated formally as follows:

Rule 1 $\sigma_{p_i}(\text{R}_j) = \phi$ if $\forall x$ in R $: \neg(p_i(x) \wedge p_j(x))$

where p_i and p_j are selection predicates, x denotes a tuple, and $p(x)$ denotes "predicate p holds for x."

For example, the selection predicate ENO="E1" conflicts with the predicates of fragments EMP_2 and EMP_3 of Example 4.3 (i.e., no tuple in EMP_2 and EMP_3 can satisfy this predicate). Determining the contradicting predicates requires theorem-

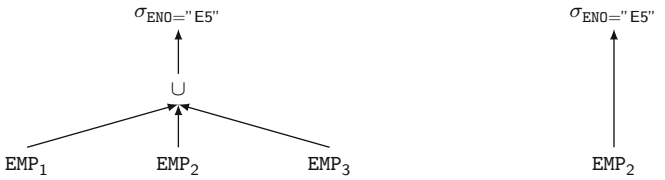

Fig. 4.3 Reduction for horizontal fragmentation (with selection). (**a**) Fragment query. (**b**) Reduced query

proving techniques if the predicates are quite general. However, DBMSs generally simplify predicate comparison by supporting only simple predicates for defining fragmentation rules (by the database administrator).

Example 4.4 We now illustrate reduction by horizontal fragmentation using the following example query:

```
SELECT *
FROM   EMP
WHERE  ENO = "E5"
```

Applying the naive approach to localize EMP using EMP_1, EMP_2, and EMP_3 gives the fragment query of Fig. 4.3a. By commuting the selection with the union operator, it is easy to detect that the selection predicate contradicts the predicates of EMP_1 and EMP_3, thereby producing empty relations. The reduced query is simply applied to EMP_2 as shown in Fig. 4.3b. ♦

4.2.2 Reduction with Join

Joins on horizontally fragmented relations can be simplified when the joined relations are fragmented according to the join attribute. The simplification consists of distributing joins over unions and eliminating useless joins. The distribution of join over union can be stated as:

$$(\text{R}_1 \cup \text{R}_2) \bowtie \text{S} = (\text{R}_1 \bowtie \text{S}) \cup (\text{R}_2 \bowtie \text{S})$$

where R_i are fragments of R and S is a relation.

With this transformation, unions can be moved up in the operator trie so that all possible joins of fragments are exhibited. Useless joins of fragments can be determined when the qualifications of the joined fragments are contradicting, thus yielding an empty result. Assuming that fragments R_i and R_j are defined, respec-

tively, according to predicates p_i and p_j on the same attribute, the simplification rule can be stated as follows:

Rule 2 $R_i \bowtie R_j = \phi$ if $\forall x$ in R_i, $\forall y$ in $R_j : \neg(p_i(x) \wedge p_j(y))$

The determination of useless joins and their elimination using rule 2 can thus be performed by looking only at the fragment predicates. The application of this rule allows the join of two relations to be implemented as parallel partial joins of fragments. It is not always the case that the reduced query is better (i.e., simpler) than the fragment query. The fragment query is better when there are a large number of partial joins in the reduced query. This case arises when there are few contradicting fragmentation predicates. The worst case occurs when each fragment of one relation must be joined with each fragment of the other relation. This is tantamount to the Cartesian product of the two sets of fragments, with each set corresponding to one relation. The reduced query is better when the number of partial joins is small. For example, if both relations are fragmented using the same predicates, the number of partial joins is equal to the number of fragments of each relation. One advantage of the reduced query is that the partial joins can be done in parallel, and thus increase response time.

Example 4.5 Assume that relation EMP is fragmented as EMP_1, EMP_2, EMP_3, as above, and that relation ASG is fragmented as

$$ASG_1 = \sigma_{ENO \leq "E3"}(ASG)$$

$$ASG_2 = \sigma_{ENO > "E3"}(ASG)$$

EMP_1 and ASG_1 are defined by the same predicate. Furthermore, the predicate defining ASG_2 is the union of the predicates defining EMP_2 and EMP_3. Now consider the join query

```
SELECT *
FROM   EMP NATURAL JOIN ASG
```

The equivalent fragment query is given in Fig. 4.4a. The query reduced by distributing joins over unions and applying rule 2 can be implemented as a union of three partial joins that can be done in parallel (Fig. 4.4b). ♦

4.2.3 Reduction for Vertical Fragmentation

The vertical fragmentation function distributes a relation based on projection attributes. Since the reconstruction operator for vertical fragmentation is the join, the materialization program for a vertically fragmented relation consists of the join of the fragments on the common attribute. For vertical fragmentation, we use the following example.

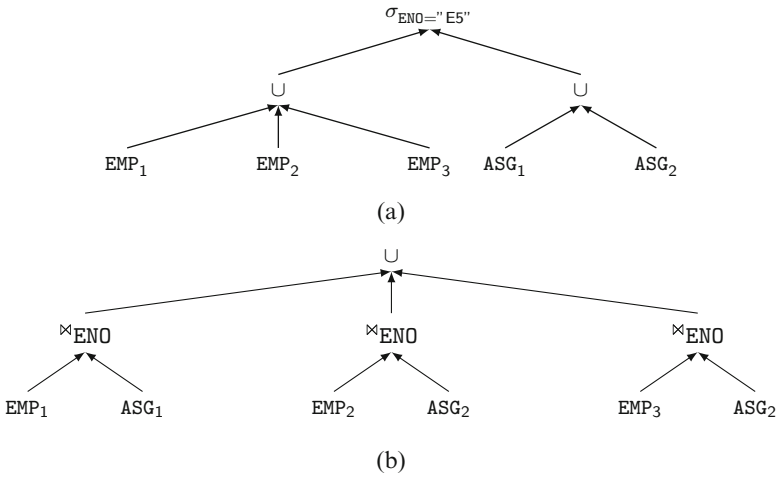

Fig. 4.4 Reduction by horizontal fragmentation (with join). (**a**) Fragment query. (**b**) Reduced query

Example 4.6 Relation EMP can be divided into two vertical fragments where the key attribute ENO is duplicated:

$$\text{EMP}_1 = \Pi_{\text{ENO,ENAME}}(\text{EMP})$$

$$\text{EMP}_2 = \Pi_{\text{ENO,TITLE}}(\text{EMP})$$

The materialization program is

$$\text{EMP} = \text{EMP}_1 \bowtie_{\text{ENO}} \text{EMP}_2$$ ♦

Similar to horizontal fragmentation, queries on vertical fragments can be reduced by determining the useless intermediate relations and removing the subtrees that produce them. Projections on a vertical fragment that has no attributes in common with the projection attributes (except the key of the relation) produce useless, though not empty relations. Given a relation R, defined over attributes $\text{A} = \{\text{A}_1, \ldots, \text{A}_n\}$, which is vertically fragmented as $\text{R}_i = \Pi_{\text{A}'}(\text{R})$, where $\text{A}' \subseteq \text{A}$, the rule can be formally stated as follows:

Rule 3 $\Pi_{\text{D,K}}(\text{R}_i)$ is useless if the set of projection attributes D is not in A'

Example 4.7 Let us illustrate the application of this rule using the following example query in SQL:

```
SELECT ENAME
FROM   EMP
```

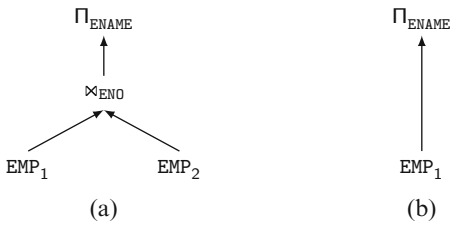

Fig. 4.5 Reduction for vertical fragmentation. (**a**) Fragment query. (**b**) Reduced query

The equivalent fragment query on $\texttt{EMP}_1$ and $\texttt{EMP}_2$ (as obtained in Example 4.4) is given in Fig. 4.5a. By commuting the projection with the join (i.e., projecting on ENO, ENAME), we can see that the projection on $\texttt{EMP}_2$ is useless because ENAME is not in $\texttt{EMP}_2$. Therefore, the projection needs to apply only to $\texttt{EMP}_1$, as shown in Fig. 4.5b. ♦

4.2.4 Reduction for Derived Fragmentation

As we saw in previous sections, the join operator, which is probably the most important operator because it is both frequent and expensive, can be optimized by using primary horizontal fragmentation when the joined relations are fragmented according to the join attributes. In this case the join of two relations is implemented as a union of partial joins. However, this method precludes one of the relations from being fragmented on a different attribute used for selection. Derived horizontal fragmentation is another way of distributing two relations so that the joint processing of selection and join is improved. Typically, if relation R is subject to derived horizontal fragmentation due to relation S, the fragments of R and S that have the same join attribute values are located at the same site. In addition, S can be fragmented according to a selection predicate.

Since tuples of R are placed according to the tuples of S, derived fragmentation should be used only for one-to-many (hierarchical) relationships of the form $\texttt{S} \rightarrow \texttt{R}$, where a tuple of S can match with n tuples of R, but a tuple of R matches with exactly one tuple of S. Note that derived fragmentation could be used for many-to-many relationships provided that tuples of S (that match with n tuples of R) are replicated. For simplicity, we assume and advise that derived fragmentation be used only for hierarchical relationships.

Example 4.8 Given a one-to-many relationship from EMP to ASG, relation ASG(ENO, PNO, RESP, DUR) can be indirectly fragmented according to the following rules:

$$\texttt{ASG}_1 = \texttt{ASG} \ltimes_{\texttt{ENO}} \texttt{EMP}_1$$

$$\texttt{ASG}_2 = \texttt{ASG} \ltimes_{\texttt{ENO}} \texttt{EMP}_2$$

Recall from Chap. 2 that $\texttt{EMP}_1$ and $\texttt{EMP}_2$ are fragmented as follows:

$$\texttt{EMP}_1 = \sigma_{\texttt{TITLE}=\text{"Programmer"}}(\texttt{EMP})$$

$$\texttt{EMP}_2 = \sigma_{\texttt{TITLE}\neq\text{"Programmer"}}(\texttt{EMP})$$

The materialization program for a horizontally fragmented relation is the union of the fragments. In our example, we have

$$\texttt{ASG} = \texttt{ASG}_1 \cup \texttt{ASG}_2$$

♦

Queries on derived fragments can also be reduced. Since this type of fragmentation is useful for optimizing join queries, a useful transformation is to distribute joins over unions (used in the materialization programs) and to apply rule 2 introduced earlier. Because the fragmentation rules indicate what the matching tuples are, certain joins will produce empty relations if the fragmentation predicates conflict. For example, the predicates of $\texttt{ASG}_1$ and $\texttt{EMP}_2$ conflict; thus, we have

$$\texttt{ASG}_1 \ltimes \texttt{EMP}_2 = \phi$$

Contrary to the reduction with join discussed previously, the reduced query is always preferable to the fragment query because the number of partial joins usually equals the number of fragments of R.

Example 4.9 The reduction by derived fragmentation is illustrated by applying it to the following SQL query, which retrieves all attributes of tuples from EMP and ASG that have the same value of ENO and the title "Mech. Eng.":

```
SELECT *
FROM   EMP NATURAL JOIN ASG
WHERE  TITLE = "Mech. Eng."
```

The fragment query on fragments $\texttt{EMP}_1$, $\texttt{EMP}_2$, $\texttt{ASG}_1$, and $\texttt{ASG}_2$ defined previously is given in Fig. 4.6a. By pushing selection down to fragments $\texttt{EMP}_1$ and $\texttt{EMP}_2$, the query reduces to that of Fig. 4.6b. This is because the selection predicate conflicts with that of $\texttt{EMP}_1$, and thus $\texttt{EMP}_1$ can be removed. In order to discover conflicting join predicates, we distribute joins over unions. This produces the trie of Fig. 4.6c. The left subtree joins two fragments, $\texttt{ASG}_1$ and $\texttt{EMP}_2$, whose qualifications conflict because of predicates TITLE = "Programmer" in $\texttt{ASG}_1$, and TITLE $\neq$ "Programmer" in $\texttt{EMP}_2$. Therefore the left subtree which produces an empty relation can be removed, and the reduced query of Fig. 4.6d is obtained. The resulting query is made simpler, illustrating the value of fragmentation in improving the performance of distributed queries. ♦

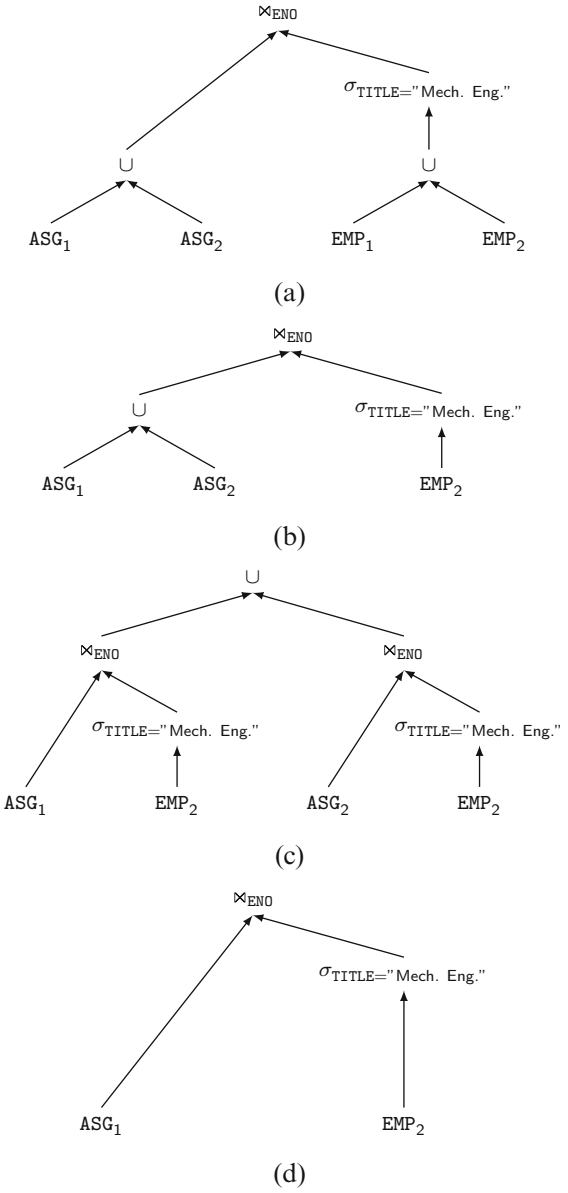

Fig. 4.6 Reduction for indirect fragmentation. (**a**) Fragment query. (**b**) Query after pushing selection down. (**c**) Query after moving unions up. (**d**) Reduced query after eliminating the left subtree

4.2.5 Reduction for Hybrid Fragmentation

Hybrid fragmentation is obtained by combining the fragmentation functions discussed above. The goal of hybrid fragmentation is to support, efficiently, queries involving projection, selection, and join. Note that the optimization of an operator or of a combination of operators is always done at the expense of other operators. For example, hybrid fragmentation based on selection–projection will make selection only, or projection only, less efficient than with horizontal fragmentation (or vertical fragmentation). The materialization program for a hybrid fragmented relation uses unions and joins of fragments.

Example 4.10 Here is an example of hybrid fragmentation of relation EMP:

$$\text{EMP}_1 = \sigma_{\text{ENO} \leq \text{"E4"}}(\Pi_{\text{ENO,ENAME}}(\text{EMP}))$$

$$\text{EMP}_2 = \sigma_{\text{ENO} > \text{"E4"}}(\Pi_{\text{ENO,ENAME}}(\text{EMP}))$$

$$\text{EMP}_3 = \Pi_{\text{ENO,TITLE}}(\text{EMP})$$

In our example, the materialization program is

$$\text{EMP} = (\text{EMP}_1 \cup \text{EMP}_2) \bowtie_{\text{ENO}} \text{EMP}_3$$

♦

Queries on hybrid fragments can be reduced by combining the rules used, respectively, in primary horizontal, vertical, and derived horizontal fragmentation. These rules can be summarized as follows:

1. Remove empty relations generated by contradicting selections on horizontal fragments.
2. Remove useless relations generated by projections on vertical fragments.
3. Distribute joins over unions in order to isolate and remove useless joins.

Example 4.11 The following example query in SQL illustrates the application of rules (1) and (2) to the horizontal–vertical fragmentation of relation EMP into EMP_1, EMP_2, and EMP_3 given above:

```
SELECT ENAME
FROM   EMP
WHERE  ENO="E5"
```

The fragment query of Fig. 4.7a can be reduced by first pushing selection down, eliminating fragment EMP_1, and then pushing projection down, eliminating fragment EMP_3. The reduced query is given in Fig. 4.7b. ♦

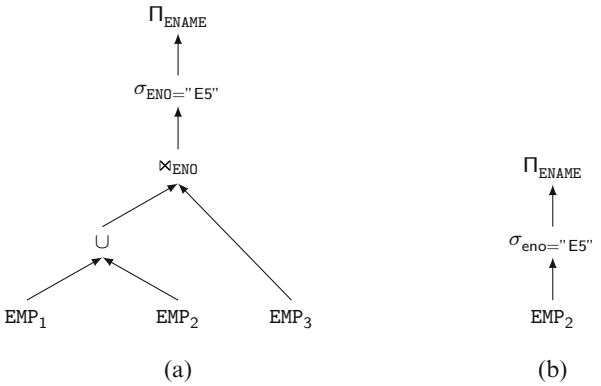

Fig. 4.7 Reduction for hybrid fragmentation. (**a**) Fragment query. (**b**) Reduced query

4.3 Join Ordering in Distributed Queries

Ordering joins is an important aspect of centralized query optimization. Join ordering in a distributed context is even more important since joins between fragments may increase the communication time. Therefore, the search space investigated by a distributed query optimizer concentrates on join trees (see next section). Two basic approaches exist to order joins in distributed queries. One tries to optimize the ordering of joins directly, whereas the other replaces joins by combinations of semijoins in order to minimize communication costs.

4.3.1 Join Trees

QEPs are typically abstracted by means of operator trees, which define the order in which the operators are executed. They are enriched with additional information, such as the best algorithm chosen for each operator. For a given query, the search space can thus be defined as the set of equivalent operator trees that can be produced using transformation rules. To characterize query optimizers, it is useful to concentrate on *join trees*, which are operator trees whose operators are join or Cartesian product. This is because permutations of the join order have the most important effect on performance of relational queries.

Example 4.12 Consider the following query:

```
SELECT ENAME, RESP
FROM   EMP NATURAL JOIN ASG NATURAL JOIN PROJ
```

Figure 4.8 illustrates three equivalent join trees for that query, which are obtained by exploiting the associativity of binary operators. Each of these join trees can be

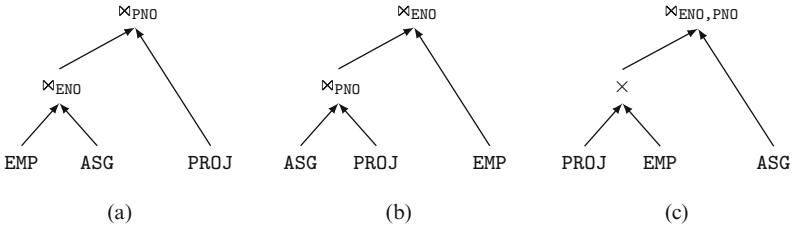

Fig. 4.8 Equivalent join trees

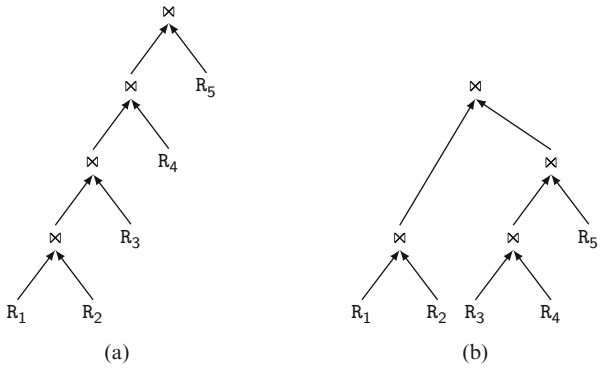

Fig. 4.9 The two major shapes of join trees. (**a**) Linear join trie. (**b**) Bushy join trie

assigned a cost based on the estimated cost of each operator. Join trie (c) which starts with a Cartesian product may have a much higher cost than the other join trees. ♦

For a complex query (involving many relations and many operators), the number of equivalent operator trees can be very high. For instance, the number of alternative join trees that can be produced by applying the commutativity and associativity rules is $\mathcal{O}(N!)$ for N relations. Investigating a large search space may make optimization time prohibitive, sometimes much more expensive than the actual execution time. Therefore, query optimizers typically restrict the size of the search space they consider. The first restriction is to use heuristics. The most common heuristic is to perform selection and projection when accessing base relations. Another common heuristic is to avoid Cartesian products that are not required by the query. For instance, in Fig. 4.8, operator trie (c) would not be part of the search space considered by the optimizer.

Another important restriction is with respect to the shape of the join trie. Two kinds of join trees are usually distinguished: linear versus bushy trees (see Fig. 4.9). A *linear trie* is a trie such that at least one operand of each operator node is a base relation. A *left linear trie* is a linear trie where the right subtree of a join node is always a leaf node corresponding to a base relation. A *bushy trie* is more general and may have operators with no base relations as operands (i.e., both operands are intermediate relations). By considering only linear trees, the size of the search space

is reduced to $\mathcal{O}(2^N)$. However, in a distributed environment, bushy trees are useful in exhibiting parallelism. For example, in join trie (b) of Fig. 4.9, operators $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$ can be done in parallel.

4.3.2 Join Ordering

Some algorithms optimize the ordering of joins directly without using semijoins. The purpose of this section is to stress the difficulty that join ordering presents and to motivate the subsequent section, which deals with the use of semijoins to optimize join queries.

A number of assumptions are necessary to concentrate on the main issues. Since the query is expressed on fragments, we do not need to distinguish between fragments of the same relation and fragments of different relations. To simplify notation, we use the term *relation* to designate a fragment stored at a particular site. Also, to concentrate on join ordering, we ignore local processing time, assuming that reducers (selection, projection) are executed locally either before or during the join (remember that doing selection first is not always efficient). Therefore, we consider only join queries whose operand relations are stored at different sites. We assume that relation transfers are done in a set-at-a-time mode rather than in a tuple-at-a-time mode. Finally, we ignore the transfer time for producing the data at a result site.

Let us first concentrate on the simpler problem of operand transfer in a single join. The query is $R \bowtie S$, where R and S are relations stored at different sites. The obvious choice of the relation to transfer is to send the smaller relation to the site of the larger one, which gives rise to two possibilities, as shown in Fig. 4.10. To make this choice we need to evaluate the sizes of R and S (we assume there is a function $size()$). We now consider the case where there are more than two relations to join. As in the case of a single join, the objective of the join-ordering algorithm is to transmit smaller operands. The difficulty stems from the fact that the join operators may reduce or increase the size of the intermediate results. Thus, estimating the size of join results is mandatory, but also difficult. A solution is to estimate the communication costs of all alternative strategies and to choose the best one. However, as discussed earlier, the number of strategies grows rapidly with the number of relations. This approach makes optimization costly, although this overhead is amortized rapidly if the query is executed frequently.

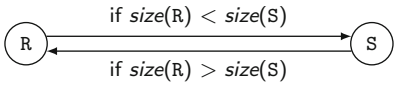

Fig. 4.10 Transfer of operands in binary operator

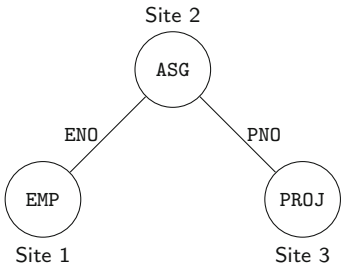

Fig. 4.11 Join graph of distributed query

Example 4.13 Consider the following query expressed in relational algebra:

$$\texttt{PROJ} \bowtie_{\texttt{PNO}} \texttt{ASG} \bowtie_{\texttt{ENO}} \texttt{EMP}$$

This query can be represented by its join graph in Fig. 4.11. Note that we have made certain assumptions about the locations of the three relations. This query can be executed in at least five different ways. We describe these strategies by the following programs, where (R $\rightarrow$ site j) stands for "relation R is transferred to site j."

1. EMP $\rightarrow$ site 2;
Site 2 computes $\texttt{EMP}' = \texttt{EMP} \bowtie \texttt{ASG}$;
$\texttt{EMP}' \rightarrow$ site 3;
Site 3 computes $\texttt{EMP}' \bowtie \texttt{PROJ}$.
2. ASG $\rightarrow$ site 1;
Site 1 computes $\texttt{EMP}' = \texttt{EMP} \bowtie \texttt{ASG}$;
$\texttt{EMP}' \rightarrow$ site 3;
Site 3 computes $\texttt{EMP}' \bowtie \texttt{PROJ}$.
3. ASG $\rightarrow$ site 3;
Site 3 computes $\texttt{ASG}' = \texttt{ASG} \bowtie \texttt{PROJ}$;
$\texttt{ASG}' \rightarrow$ site 1;
Site 1 computes $\texttt{ASG}' \bowtie \texttt{EMP}$.
4. PROJ $\rightarrow$ site 2;
Site 2 computes $\texttt{PROJ}' = \texttt{PROJ} \bowtie \texttt{ASG}$;
$\texttt{PROJ}' \rightarrow$ site 1;
Site 1 computes $\texttt{PROJ}' \bowtie \texttt{EMP}$.
5. EMP $\rightarrow$ site 2;
PROJ $\rightarrow$ site 2;
Site 2 computes $\texttt{EMP} \bowtie \texttt{PROJ} \bowtie \texttt{ASG}$

To select one of these programs, the following sizes must be known or predicted: $size(\texttt{EMP})$, $size(\texttt{ASG})$, $size(\texttt{PROJ})$, $size(\texttt{EMP} \bowtie \texttt{ASG})$, and $size(\texttt{ASG} \bowtie \texttt{PROJ})$. Furthermore, if it is the response time that is being considered, the optimization must take into account the fact that transfers can be done in parallel with strategy 5. An alternative to enumerating all the solutions is to use heuristics that consider only

the sizes of the operand relations by assuming, for example, that the cardinality of the resulting join is the product of operand cardinalities. In this case, relations are ordered by increasing sizes and the order of execution is given by this ordering and the join graph. For instance, the order (EMP, ASG, PROJ) could use strategy 1, while the order (PROJ, ASG, EMP) could use strategy 4. ♦

4.3.3 Semijoin-Based Algorithms

The semijoin operator has the important property of reducing the size of the operand relation. When the main cost component considered by the query processor is communication, a semijoin is particularly useful for improving the processing of distributed join operators as it reduces the size of data exchanged between sites. However, using semijoins may result in an increase in the number of messages and in the local processing time. The early distributed DBMSs, such as SDD-1, which were designed for slow wide area networks, make extensive use of semijoins. Nevertheless, semijoins are still beneficial in the context of fast networks when they induce a strong reduction of the join operand. Therefore, some algorithms aim at selecting an optimal combination of joins and semijoins.

In this section, we show how the semijoin operator can be used to decrease the total time of join queries. We are making the same assumptions as in Sect. 4.3.2. The main shortcoming of the join approach described in the preceding section is that entire operand relations must be transferred between sites. The semijoin acts as a size reducer for a relation much as a selection does.

The join of two relations R and S over attribute A, stored at sites 1 and 2, respectively, can be computed by replacing one or both operand relations by a semijoin with the other relation, using the following rules:

$$\begin{aligned} \mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S} &\Leftrightarrow (\mathtt{R} \ltimes_{\mathtt{A}} \mathtt{S}) \bowtie_{\mathtt{A}} \mathtt{S} \\ &\Leftrightarrow \mathtt{R} \bowtie_{\mathtt{A}} (\mathtt{S} \ltimes_{\mathtt{A}} \mathtt{R}) \\ &\Leftrightarrow (\mathtt{R} \ltimes_{\mathtt{A}} \mathtt{S}) \bowtie_{\mathtt{A}} (\mathtt{S} \ltimes_{\mathtt{A}} \mathtt{R}) \end{aligned}$$

The choice between one of the three semijoin strategies requires estimating their respective costs.

The use of the semijoin is beneficial if the cost to produce and send it to the other site is less than the cost of sending the whole operand relation and of doing the actual join. To illustrate the potential benefit of the semijoin, let us compare the costs of the two alternatives: $\mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S}$ versus $(\mathtt{R} \ltimes_{\mathtt{A}} \mathtt{S}) \bowtie_{\mathtt{A}} \mathtt{S}$, assuming that $size(\mathtt{R}) < size(\mathtt{S})$.

The following program, using the notation of Sect. 4.3.2, uses the semijoin operator:

1. $\Pi_{\mathtt{A}}(\mathtt{S}) \rightarrow$ site 1
2. Site 1 computes $\mathtt{R}' = \mathtt{R} \ltimes_{\mathtt{A}} \mathtt{S}$

3. $\mathtt{R}' \rightarrow$ site 2
4. Site 2 computes $\mathtt{R}' \ltimes_{\mathtt{A}} \mathtt{S}$

For simplicity, let us ignore the constant T_{MSG} in the communication time assuming that the term $T_{TR} * size(\mathtt{R})$ is much larger. We can then compare the two alternatives in terms of the transmitted data size. The cost of the join-based algorithm is that of transferring relation R to site 2. The cost of the semijoin-based algorithm is the cost of steps 1 and 3 above. Therefore, the semijoin approach is better if

$$size(\Pi_{\mathtt{A}}(\mathtt{S})) + size(\mathtt{R} \ltimes_{\mathtt{A}} \mathtt{S}) < size(\mathtt{R})$$

The semijoin approach is better if the semijoin acts as a sufficient reducer, that is, if a few tuples of R participate in the join. The join approach is better if almost all tuples of R participate in the join, because the semijoin approach requires an additional transfer of a projection on the join attribute. The cost of the projection step can be minimized by encoding the result of the projection in bit arrays, thereby reducing the cost of transferring the joined attribute values. It is important to note that neither approach is systematically the best; they should be considered as complementary.

More generally, the semijoin can be useful in reducing the size of the operand relations involved in multiple join queries. However, query optimization becomes more complex in these cases. Consider again the join graph of relations EMP,ASG, and PROJ given in Fig. 4.11. We can apply the previous join algorithm using semijoins to each individual join. Thus an example of a program to compute $\mathtt{EMP} \bowtie \mathtt{ASG} \bowtie \mathtt{PROJ}$ is $\mathtt{EMP}' \bowtie \mathtt{ASG}' \bowtie \mathtt{PROJ}$, where $\mathtt{EMP}' = \mathtt{EMP} \ltimes \mathtt{ASG}$ and $\mathtt{ASG}' = \mathtt{ASG} \ltimes \mathtt{PROJ}$.

However, we may further reduce the size of an operand relation by using more than one semijoin. For example, $\mathtt{EMP}'$ can be replaced in the preceding program by $\mathtt{EMP}''$ derived as

$$\mathtt{EMP}'' = \mathtt{EMP} \ltimes (\mathtt{ASG} \ltimes \mathtt{PROJ})$$

since if $size(\mathtt{ASG} \ltimes \mathtt{PROJ}) \leq size(\mathtt{ASG})$, we have $size(\mathtt{EMP}'') \leq size(\mathtt{EMP}')$. In this way, EMP can be reduced by the sequence of semijoins: $\mathtt{EMP} \ltimes (\mathtt{ASG} \ltimes \mathtt{PROJ})$. Such a sequence of semijoins is called a *semijoin program* for EMP. Similarly, semijoin programs can be found for any relation in a query. For example, PROJ could be reduced by the semijoin program $\mathtt{PROJ} \ltimes (\mathtt{ASG} \ltimes \mathtt{EMP})$. However, not all of the relations involved in a query need to be reduced; in particular, we can ignore those relations that are not involved in the final joins.

For a given relation, there exist several potential semijoin programs. The number of possibilities is in fact exponential in the number of relations. But there is one optimal semijoin program, called the *full reducer*, which reduces each relation R more than the others. The problem is to find the full reducer. A simple method is to

evaluate the size reduction of all possible semijoin programs and to select the best one. The problems with the enumerative method are twofold:

1. There is a class of queries, called *cyclic queries*, that have cycles in their join graph and for which full reducers cannot be found.
2. For other queries, called *tree queries*, full reducers exist, but the number of candidate semijoin programs is exponential in the number of relations, which makes the enumerative approach NP-hard.

In what follows, we discuss solutions to these problems.

Example 4.14 Consider the following relations, where attribute CITY has been added to relations EMP (renamed ET), PROJ (renamed PT), and ASG (renamed AT) of the engineering database. Attribute CITY of AT corresponds to the city where the employee is identified by ENO lives.

```
ET(ENO, ENAME, TITLE, CITY)
AT(ENO, PNO, RESP, DUR, CITY)
PT(PNO, PNAME, BUDGET, CITY)
```

The following SQL query retrieves the names of all employees living in the city in which their project is located together with the project name.

```
SELECT ENAME, PNAME
FROM   ET NATURAL JOIN AT NATURAL JOIN PT
       NATURAL JOIN ET
```

As illustrated in Fig. 4.12a, this query is cyclic. ♦

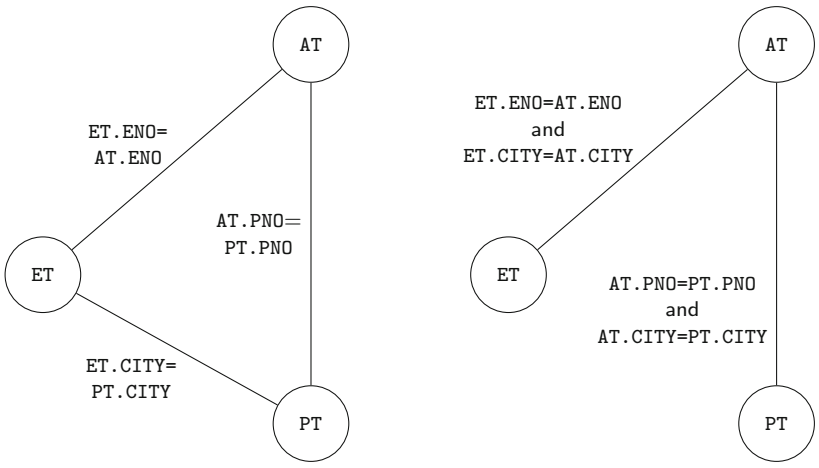

Fig. 4.12 Transformation of cyclic query. (**a**) Cyclic query. (**b**) Equivalent acyclic query

No full reducer exists for the query in Example 4.14. In fact, it is possible to derive semijoin programs for reducing it, but the number of operators is multiplied by the number of tuples in each relation, making the approach inefficient. One solution consists of transforming the cyclic graph into a trie by removing one arc of the graph and by adding appropriate predicates to the other arcs such that the removed predicate is preserved by transitivity. In the example of Fig. 4.12b, where the arc (ET, PT) is removed, the additional predicate ET.CITY = AT.CITY and AT.CITY = PT.CITY imply ET.CITY = PT.CITY by transitivity. Thus the acyclic query is equivalent to the cyclic query.

Although full reducers for trie queries exist, the problem of finding them is NP-hard. However, there is an important class of queries, called *chained queries*, for which a polynomial algorithm exists. A chained query has a join graph where relations can be ordered, and each relation joins only with the next relation in the order. Furthermore, the result of the query is at the end of the chain. For instance, the query in Fig. 4.11 is a chain query. Because of the difficulty of implementing an algorithm with full reducers, most systems use single semijoins to reduce the relation size.

4.3.4 Join Versus Semijoin

Compared with the join, the semijoin induces more operators but possibly on smaller operands. Figure 4.13 illustrates these differences with an equivalent pair of join and semijoin strategies for the query whose join graph is given in Fig. 4.11. The join EMP ⋈ ASG is done by sending one relation, e.g., ASG, to the site of the other one, e.g., EMP, to complete the join locally. When a semijoin is used, however, the

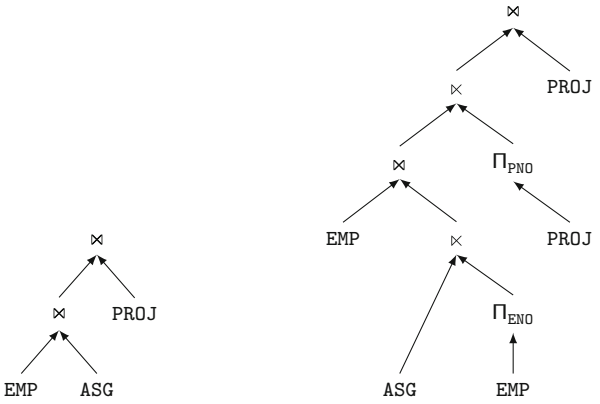

Fig. 4.13 Join versus semijoin approaches. (**a**) Join approach. (**b**) Semijoin approach

transfer of relation ASG is avoided. Instead, it is replaced by the transfer of the join attribute values of relation EMP to the site of relation ASG, followed by the transfer of the matching tuples of relation ASG to the site of relation EMP, where the join is completed. If the semijoin has good selectivity, then the semijoin approach can result in significant savings in communication time. The semijoin approach may also decrease the local processing time, by exploiting indices on the join attribute. Let us consider again the join EMP $\bowtie$ ASG, assuming that there is a selection on ASG and an index on the join attribute of ASG. Without semijoin, we would perform the selection of ASG first, and then send the result relation to the site of EMP to complete the join. Thus, the index on the join attribute of ASG cannot be used (because the join takes place at the site of EMP). Using the semijoin approach, both the selection and the semijoin ASG $\ltimes$ EMP would take place at the site of ASG, and can be performed efficiently using indices.

Semijoins can still be beneficial with fast networks if they have very good selectivity and are implemented with bit arrays. A bit array $BA[1 : n]$ is useful in encoding the join attribute values present in one relation. Let us consider the semijoin R $\ltimes$ S. Then $BA[i]$ is set to 1 if there exists a join attribute value A $= val$ in relation S such that $h(val) = i$, where h is a hash function. Otherwise, $BA[i]$ is set to 0. Such a bit array is much smaller than a list of join attribute values. Therefore, transferring the bit array instead of the join attribute values to the site of relation R saves communication time. The semijoin can be completed as follows. Each tuple of relation R, whose join attribute value is val, belongs to the semijoin if $BA[h(val)] = 1$.

4.4 Distributed Cost Model

An optimizer's cost model includes cost functions to predict the cost of operators, statistics and base data, and formulas to evaluate the sizes of intermediate results. The cost is in terms of execution time, so a cost function represents the execution time of a query.

4.4.1 Cost Functions

The cost of a distributed execution strategy can be expressed with respect to either the total time or the response time. The total time is the sum of all time (also referred to as cost) components, while the response time is the elapsed time from the initiation to the completion of the query. A general formula for determining the total time can be specified as follows:

$$Total\_time = T_{CPU} * \#insts + T_{I/O} * \#I/Os + T_{MSG} * \#msgs + T_{TR} * \#bytes$$

The first two components measure the local processing time, where T_{CPU} is the time of a CPU instruction and $T_{I/O}$ is the time of a disk I/O. The communication time is depicted by the two last components. T_{MSG} is the fixed time of initiating and receiving a message, while T_{TR} is the time it takes to transmit a data unit from one site to another. The data unit is given here in terms of bytes ($\#bytes$ is the sum of the sizes of all messages), but could be in different units (e.g., packets). A typical assumption is that T_{TR} is constant. This might not be true for wide area networks, where some sites are farther away than others. However, this assumption greatly simplifies query optimization. Thus the communication time of transferring $\#bytes$ of data from one site to another is assumed to be a linear function of $\#bytes$:

$$CT(\#bytes) = T_{MSG} + T_{TR} * \#bytes$$

Costs are generally expressed in terms of time units, which in turn can be translated into other units (e.g., dollars).

The relative values of the cost coefficients characterize the distributed database environment. The topology of the network greatly influences the ratio between these components. In a wide area network such as the Internet, the communication time is generally the dominant factor. In local area networks, however, there is more of a balance among the components. Thus, most early distributed DBMSs designed for wide area networks have ignored the local processing cost and concentrated on minimizing the communication cost. Distributed DBMSs designed for local area networks, on the other hand, consider all three cost components. The new faster networks (both wide area and local area) have improved the above ratios in favor of communication cost when all things are equal. However, communication is still the dominant time factor in wide area networks such as the Internet because of the longer distances that data is retrieved from (or shipped to).

When the response time of the query is the objective function of the optimizer, parallel local processing and parallel communications must also be considered. A general formula for response time is

$$\begin{aligned} Response\_time = {} & T_{CPU} * seq\_\#insts + T_{I/O} * seq\_\#I/Os \\ & + T_{MSG} * seq\_\#msgs + T_{TR} * seq\_\#bytes \end{aligned}$$

where $seq\_\#x$, in which x can be instructions ($insts$), I/O, messages ($msgs$), or $bytes$, is the maximum number of x which must be done sequentially for the execution of the query. Thus any processing and communication done in parallel is ignored.

Example 4.15 Let us illustrate the difference between total cost and response time using the example of Fig. 4.14, which computes the answer to a query at site 3 with data from sites 1 and 2. For simplicity, we assume that only communication cost is considered.

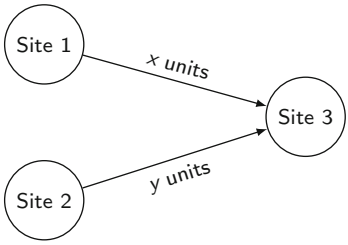

Fig. 4.14 Example of data transfers for a query

Assume that T_{MSG} and T_{TR} are expressed in time units. The total time of transferring x data units from site 1 to site 3 and y data units from site 2 to site 3 is

$$Total\_time = 2\, T_{MSG} + T_{TR} * (x + y)$$

The response time of the same query can be approximated as

$$Response\_time = max\{T_{MSG} + T_{TR} * x, T_{MSG} + T_{TR} * y\}$$

since the transfers can be done in parallel. ♦

Minimizing response time is achieved by increasing the degree of parallel execution. This does not, however, imply that the total time is also minimized. On the contrary, it can increase the total time, for example, by having more parallel local processing and transmissions. Minimizing the total time implies that the utilization of the resources improves, thus increasing the system throughput. In practice, a compromise between the two is desired. In Sect. 4.5 we present algorithms that can optimize a combination of total time and response time, with more weight on one of them.

4.4.2 Database Statistics

The main factor affecting the performance of an execution strategy is the size of the intermediate relations that are produced during the execution. When a subsequent operator is located at a different site, the intermediate relation must be transmitted over the network. Therefore, it is of prime interest to estimate the size of the intermediate results of relational algebra operators in order to minimize the size of data transfers. This estimation is based on statistical information about the base relations and formulas to predict the cardinalities of the results of the relational operators. There is a direct trade-off between the precision of the statistics and the cost of managing them, the more precise statistics being the more costly. For a relation R fragmented as R_1, R_2, ..., R_r, the statistical data typically are the following:

1. For each attribute A of relation R its length (in number of bytes), denoted by $length(\mathtt{A})$, the cardinality of its domain, denoted by $card(dom[\mathtt{A}])$, which gives the number of unique values in $dom[\mathtt{A}]$, and in the case the domain is defined on a set of values that can be ordered (e.g., integers or reals), the minimum and maximum possible values, denoted by $min(\mathtt{A})$ and $max(\mathtt{A})$
2. For each attribute A of each fragment $\mathtt{R}_i$, the number of distinct values of A, with the cardinality of the projection of fragment $\mathtt{R}_i$ on A, denoted by $card(\Pi_{\mathtt{A}}(\mathtt{R}_i))$.
3. The number of tuples in each fragment $\mathtt{R}_i$, denoted by $card(\mathtt{R}_i)$.

In addition, for each attribute A, there may be a histogram that approximates the frequency distribution of the attribute within a number of buckets, each corresponding to a range of values.

These statistics are useful to predict the size of intermediate relations. Remember that in Chap. 2 we defined the size of an intermediate relation R as follows:

$$size(\mathtt{R}) = card(\mathtt{R}) * length(\mathtt{R})$$

where $length(\mathtt{R})$ is the length (in bytes) of a tuple of R, and $card(\mathtt{R})$ is the number of tuples in R.

The estimation of $card(\mathtt{R})$ requires the use of formulas. Two simplifying assumptions are commonly made about the database. The distribution of attribute values in a relation is supposed to be uniform, and all attributes are independent, meaning that the value of an attribute does not affect the value of any other attribute. These two assumptions are often wrong in practice, but they make the problem tractable. Based on these assumptions, we can use simple formulas for estimating the cardinalities of the results of the basic relational algebra operators, based on their selectivity. The *selectivity factor* of an operator, that is, the proportion of tuples of an operand relation that participate in the result of that operation, is denoted by $SF(op)$, where op is the operation. It is a real value between 0 and 1. A low value (e.g., 0.001) corresponds to a good (or high) selectivity, while a high value (e.g., 0.5) to a bad (or low) selectivity.

Let us illustrate with the two major operators, i.e., selection and join. The cardinality of selection is

$$card(\sigma_F(\mathtt{R})) = SF(\sigma_F(\mathtt{R})) * card(\mathtt{R})$$

where $SF(\sigma_F(\mathtt{R}))$ can be computed as follows for the basic predicates:

$$SF(\sigma_{\mathtt{A}=value}(\mathtt{R})) = \frac{1}{card(\Pi_{\mathtt{A}}(\mathtt{R}))}$$

$$SF(\sigma_{\mathtt{A}>value}(\mathtt{R})) = \frac{max(\mathtt{A}) - value}{max(\mathtt{A}) - min(\mathtt{A})}$$

$$SF(\sigma_{\mathtt{A}<value}(\mathtt{R})) = \frac{value - min(\mathtt{A})}{max(\mathtt{A}) - min(\mathtt{A})}$$

The cardinality of join is

$$card(\mathtt{R} \bowtie \mathtt{S}) = SF(\mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S}) * card(\mathtt{R}) * card(\mathtt{S})$$

There is no general way to estimate $SF(\mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S})$ without additional information. Thus, a simple approximation is to use a constant, e.g., 0.01, which reflects the known join selectivity. However, there is a case, which occurs frequently, where the estimation is accurate. If relation R is equijoined with relation S over attribute A where A is a key of R and a foreign key of S, the join selectivity factor can be approximated as

$$SF(\mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S}) = \frac{1}{card(\mathtt{R})}$$

because each tuple of S matches with at most one tuple of R.

4.5 Distributed Query Optimization

In this section, we illustrate the use of the techniques presented in earlier sections within the context of three basic query optimization algorithms. First, we present the dynamic and static approaches. Then, we present a hybrid approach.

4.5.1 Dynamic Approach

We illustrate the dynamic approach with the algorithm of Distributed INGRES. The objective function of the algorithm is to minimize a combination of both the communication time and the response time. However, these two objectives may be conflicting. For instance, increasing communication time (by means of parallelism) may well decrease response time. Thus, the function can give a greater weight to one or the other. Note that this query optimization algorithm ignores the cost of transmitting the data to the result site. The algorithm also takes advantage of fragmentation, but only horizontal fragmentation is handled for simplicity.

Since both general and broadcast networks are considered, the optimizer takes into account the network topology. In broadcast networks, the same data unit can be transmitted from one site to all the other sites in a single transfer, and the algorithm explicitly takes advantage of this capability. For example, broadcasting is used to replicate fragments and then to maximize the degree of parallelism.

The input to the algorithm is a query expressed in tuple relational calculus (in conjunctive normal form) and schema information (the network type, as well as the location and size of each fragment). This algorithm is executed by the site, called the *master site*, where the query is initiated. The algorithm, which we call Dynamic-QOA, is given in Algorithm 4.1.

Algorithm 4.1: Dynamic-QOA

```
Input: MRQ: multirelation query
Output: result of the last multirelation query
begin
    for each detachable ORQ_i in MRQ do                    {ORQ is monorelation query}
        run(ORQ_i)                                                          (1)
    end for
    {MRQ replaced by n irreducible queries}
    MRQ'_list ← REDUCE(MRQ)                                                 (2)
    while n ≠ 0 do                          {n is the number of irreducible queries}   (3)
        {choose next irreducible query involving the smallest fragments}
        MRQ' ← SELECT_QUERY(MRQ'_list)                                      (3.1)
        {determine fragments to transfer and processing site for MRQ'}
        Fragment-site-list ← SELECT_STRATEGY(MRQ')                          (3.2)
        {move the selected fragments to the selected sites}
        for each pair (F, S) in Fragment-site-list do
            move fragment F to site S                                       (3.3)
        end for
        execute MRQ'                                                        (3.4)
        n ← n − 1
    end while
    {output is the result of the last MRQ'}
end
```

All monorelation queries (e.g., selection and projection) that can be detached are first processed locally (step 1). Then the reduction algorithm is applied to the original query (step 2). Reduction is a technique that isolates all irreducible subqueries and monorelation subqueries by detachment. Monorelation subqueries are ignored because they have already been processed in step 1. Thus the REDUCE procedure produces a sequence of irreducible subqueries $q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n$, with at most one relation in common between two consecutive subqueries.

Based on the list of irreducible queries isolated in step 2 and the size of each fragment, the next subquery, MRQ', which hasat least two variables, is chosen at step 3.1 and steps 3.2–3.4 are applied to it. Steps 3.1 and 3.2 are discussed below. Step 3.2 selects the best strategy to process the query MRQ'. This strategy is described by a list of pairs (F, S), in which F is a fragment to transfer to the processing site S. Step 3.3 transfers all the fragments to their processing sites. Finally, step 3.4 executes the query MRQ'. If there are remaining subqueries, the algorithm goes back to step 3 and performs the next iteration. Otherwise, it terminates.

Optimization occurs in steps 3.1 and 3.2. The algorithm has produced subqueries with several components and their dependency order (similar to the one given by a relational algebra trie). At step 3.1, a simple choice for the next subquery is to take the next one having no predecessor and involving the smaller fragments. This minimizes the size of the intermediate results. For example, if a query q has the subqueries q_1, q_2, and q_3, with dependencies $q_1 \rightarrow q_3, q_2 \rightarrow q_3$, and if the fragments referred to by q_1 are smaller than those referred to by q_2, then q_1 is

selected. Depending on the network, this choice can also be affected by the number of sites having relevant fragments.

The subquery selected must then be executed. Since the relation involved in a subquery may be stored at different sites and even fragmented, the subquery may nevertheless be further subdivided.

Example 4.16 Let us consider the following query:

"Names of employees working on the CAD/CAM project"

This query can be expressed in SQL by the following query q_1 on the engineering database:

```
q1 :  SELECT EMP.ENAME
      FROM   EMP NATURAL JOIN ASG NATURAL JOIN PROJ
      WHERE  PNAME="CAD/CAM"
```

Assume that relations EMP, ASG, and PROJ are stored as follows, where relation EMP is fragmented.

Site 1	Site 2
EMP_1	EMP_2
ASG	PROJ

There are several possible strategies, including the following:

1. Execute the entire query (EMP $\bowtie$ ASG $\bowtie$ PROJ) by moving EMP_1 and ASG to site 2.
2. Execute (EMP $\bowtie$ ASG) $\bowtie$ PROJ by moving (EMP_1 $\bowtie$ ASG) and ASG to site 2, and so on.

The choice between the possible strategies requires an estimate of the size of the intermediate results. For example, if $size(\text{EMP} \bowtie \text{ASG}) > size(\text{EMP}_1)$, strategy 1 is preferred to strategy 2. Therefore, an estimate of the size of joins is required. ♦

At step 3.2, the next optimization problem is to determine how to execute the subquery by selecting the fragments that will be moved and the sites where the processing will take place. For an n-relation subquery, fragments from $n-1$ relations must be moved to the site(s) of fragments of the remaining relation, say R_p, and then replicated there. Also, the remaining relation may be further partitioned into k "equalized" fragments in order to increase parallelism. This method is called *fragment-and-replicate* and performs a substitution of fragments rather than of tuples. The selection of the remaining relation and of the number of processing sites k on which it should be partitioned is based on the objective function and the topology of the network. Remember that replication is cheaper in broadcast networks than in point-to-point networks. Furthermore, the choice of the number of processing sites involves a trade-off between response time and total time. A larger

number of sites decreases response time (by parallel processing) but increases total time, in particular increasing communication costs.

Formulas to minimize either communication time or processing time use as input the location of fragments, their size, and the network type. They can minimize both costs but with a priority to one. To illustrate these formulas, we give the rules for minimizing communication time. The rule for minimizing response time is even more complex. We use the following assumptions. There are n relations $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n$ involved in the query. $\mathtt{R}_i^j$ denotes the fragment of $\mathtt{R}_i$ stored at site j. There are m sites in the network. Finally, $CT_k(\#bytes)$ denotes the communication time of transferring $\#bytes$ to k sites, with $1 \leq k \leq m$. The rule for minimizing communication time considers the types of networks separately. Let us first concentrate on a broadcast network. In this case we have

$$CT_k(\#bytes) = CT_1(\#bytes)$$

The rule can be stated as

if $\max_{j=1,m}(\sum_{i=1}^{n} size(\mathtt{R}_i^j)) > \max_{i=1,n}(size(\mathtt{R}_i))$
then
    the processing site is the j with the
        largest amount of data
else
    $\mathtt{R}_p$ is the largest relation **and**
    site of $\mathtt{R}_p$ is the processing site

If the inequality predicate is satisfied, one site contains an amount of data useful to the query larger than the size of the largest relation. Therefore, this site should be the processing site. If the predicate is not satisfied, one relation is larger than the maximum useful amount of data at one site. Therefore, this relation should be the $\mathtt{R}_p$, and the processing sites are those which have its fragments.

Let us now consider the case of the point-to-point networks. In this case we have

$$CT_k(\#bytes) = k * CT_1(\#bytes)$$

The choice of $\mathtt{R}_p$ that minimizes communication is obviously the largest relation. Assuming that the sites are arranged by decreasing order of amounts of useful data for the query, that is,

$$\sum_{i=1}^{n} size(R_i^j) > \sum_{i=1}^{n} size(R_i^{j+1})$$

the choice of k, the number of sites at which processing needs to be done, is given as

if $\sum_{i \neq p}(size(\mathtt{R}_i) - size(\mathtt{R}_i^1)) > size(\mathtt{R}_p^1)$
then
$k = 1$
else
k `is the largest` j `such that` $\sum_{i \neq p}(size(\mathtt{R}_i) - size(\mathtt{R}_i^j)) \leq size(\mathtt{R}_p^j)$

This rule chooses a site as the processing site only if the amount of data it must receive is smaller than the additional amount of data it would have to send if it were not a processing site. Obviously, the then-part of the rule assumes that site 1 stores a fragment of $\mathtt{R}_p$.

Example 4.17 Let us consider the query PROJ ⋈ ASG, where PROJ and ASG are fragmented. Assume that the allocation of fragments and their sizes are as follows (in kilobytes):

	Site 1	Site 2	Site 3	Site 4
PROJ	1000	1000	1000	1000
ASG			2000	

With a point-to-point network, the best strategy is to send each $\mathtt{PROJ}_i$ to site 3, which requires a transfer of 3000 kbytes, versus 6000 kbytes if ASG is sent to sites 1, 2, and 4. However, with a broadcast network, the best strategy is to send ASG (in a single transfer) to sites 1, 2, and 4, which incurs a transfer of 2000 kbytes. The latter strategy is faster and maximizes response time because the joins can be done in parallel. ♦

This dynamic query optimization algorithm is characterized by a limited search of the solution space, where an optimization decision is taken for each step without concerning itself with the consequences of that decision on global optimization. However, the algorithm is able to correct a local decision that proves to be incorrect.

4.5.2 Static Approach

We illustrate the static approach with the algorithm of R*, which has been the basis for many distributed query optimizers. This algorithm performs an exhaustive search of all alternative strategies in order to choose the one with the least cost. Although predicting and enumerating these strategies may be costly, the overhead of exhaustive search is rapidly amortized if the query is executed frequently. Query compilation is a distributed task, coordinated by a *master site*, where the query is initiated. The optimizer of the master site makes all intersite decisions, such as the selection of the execution sites and the fragments as well as the method for transferring data. The *apprentice sites*, which are the other sites that have relations involved in the query, make the remaining local decisions (such as the ordering of

Algorithm 4.2: Static*-QOA

```
Input: QT: query trie
Output: strat: minimum cost strategy
begin
    for each relation R_i ∈ QT do
        for each access path AP_ij to R_i do
            compute cost(AP_ij)
        end for
        best_AP_i ← AP_ij with minimum cost
    end for
    for each order (R_i1, R_i2, ··· , R_in) with i = 1, ··· , n! do
        build strategy (...((best AP_i1 ⋈ R_i2) ⋈ R_i3) ⋈ ... ⋈ R_in)
        compute the cost of strategy
    end for
    strat ← strategy with minimum cost
    for each site k storing a relation involved in QT do
        LS_k ← local strategy (strategy, k)
        send (LS_k, site k)                    {each local strategy is optimized at site k}
    end for
end
```

joins at a site) and generate local access plans for the query. The objective function of the optimizer is the general total time function, including local processing and communications costs.

We now summarize this query optimization algorithm. The input to the algorithm is a fragment query expressed as a relational algebra trie (the query trie), the location of relations, and their statistics. The algorithm is described by the procedure Static-QOA in Algorithm 4.2.

The optimizer must select the join ordering, the join algorithm (nested-loop or merge-join), and the access path for each fragment (e.g., clustered index, sequential scan, etc.). These decisions are based on statistics and formulas used to estimate the size of intermediate results and access path information. In addition, the optimizer must select the sites of join results and the method of transferring data between sites. To join two relations, there are three candidate sites: the site of the first relation, the site of the second relation, or a third site (e.g., the site of a third relation to be joined with). Two methods are supported for intersite data transfers.

1. *Ship-whole*. The entire relation is shipped to the join site and stored in a temporary relation before being joined. If the join algorithm is merge-join, the relation does not need to be stored, and the join site can process incoming tuples in a pipeline mode, as they arrive.
2. *Fetch-as-needed*. The external relation is sequentially scanned, and for each tuple the join value is sent to the site of the internal relation, which selects the internal tuples matching the value and sends the selected tuples to the site of the external relation. This method, also called *bindjoin*, is equivalent to the semijoin of the internal relation with each external tuple.

The trade-off between these two methods is obvious. Ship-whole generates a larger data transfer but fewer messages than fetch-as-needed. It is intuitively better to ship-whole relations when they are small. On the contrary, if the relation is large and the join has good selectivity (only a few matching tuples), the relevant tuples should be fetched as needed. The optimizer does not consider all possible combinations of join methods with transfer methods since some of them are not worthwhile. For example, it would be useless to transfer the external relation using fetch-as-needed in the nested loop join algorithm, because all the outer tuples must be processed anyway and therefore should be transferred as a whole.

Given the join of an external relation R with an internal relation S on attribute A, there are four join strategies. In what follows we describe each strategy in detail and provide a simplified cost formula for each, where LT denotes local processing time (I/O + CPU time) and CT denotes communication time. For simplicity, we ignore the cost of producing the result. For convenience, we denote by s the average number of tuples of S that match one tuple of R:

$$s = \frac{card(\mathtt{S} \ltimes_{\mathtt{A}} \mathtt{R})}{card(\mathtt{R})}$$

Strategy 1. Ship the entire external relation to the site of the internal relation. In this case the external tuples can be joined with S as they arrive. Thus we have

$$\begin{aligned} Total\_time = {} & LT(\text{retrieve } card(\mathtt{R}) \text{ tuples from } \mathtt{R}) \\ & + CT(size(\mathtt{R})) \\ & + LT(\text{retrieve } s \text{ tuples from } \mathtt{S}) * card(\mathtt{R}) \end{aligned}$$

Strategy 2. Ship the entire internal relation to the site of the external relation. In this case, the internal tuples cannot be joined as they arrive, and they need to be stored in a temporary relation T. Thus we have

$$\begin{aligned} Total\_time = {} & LT(\text{retrieve } card(\mathtt{S}) \text{ tuples from } \mathtt{S}) \\ & + CT(size(\mathtt{S})) \\ & + LT(\text{store } card(\mathtt{S}) \text{ tuples in } \mathtt{T}) \\ & + LT(\text{retrieve } card(\mathtt{R}) \text{ tuples from } \mathtt{R}) \\ & + LT(\text{retrieve } s \text{ tuples from } \mathtt{T}) * card(R) \end{aligned}$$

Strategy 3. Fetch tuples of the internal relation as needed for each tuple of the external relation. In this case, for each tuple in R, the join attribute (A) value is sent

to the site of S. Then the s tuples of S which match that value are retrieved and sent to the site of R to be joined as they arrive. Thus we have

$$\begin{aligned} Total\_time = {} & LT(\text{retrieve } card(\mathtt{R}) \text{ tuples from } \mathtt{R}) \\ & + CT(length(\mathtt{A})) * card(\mathtt{R}) \\ & + LT(\text{retrieve } s \text{ tuples from } \mathtt{S}) * card(\mathtt{R}) \\ & + CT(s * length(\mathtt{S})) * card(\mathtt{R}) \end{aligned}$$

Strategy 4. Move both relations to a third site and compute the join there. In this case the internal relation is first moved to a third site and stored in a temporary relation T. Then the external relation is moved to the third site and its tuples are joined with T as they arrive. Thus we have

$$\begin{aligned} Total\_time = {} & LT(\text{retrieve } card(\mathtt{S}) \text{ tuples from } \mathtt{S}) \\ & + CT(size(\mathtt{S})) \\ & + LT(\text{store } card(\mathtt{S}) \text{ tuples in } \mathtt{T}) \\ & + LT(\text{retrieve } card(\mathtt{R}) \text{ tuples from } \mathtt{R}) \\ & + CT(size(\mathtt{R})) \\ & + LT(\text{retrieve } s \text{ tuples from } \mathtt{T}) * card(\mathtt{R}) \end{aligned}$$

Example 4.18 Let us consider a query that consists of the join of relations PROJ, the external relation, and ASG, the internal relation, on attribute PNO. We assume that PROJ and ASG are stored at two different sites and that there is an index on attribute PNO for relation ASG. The possible execution strategies for the query are as follows:

1. Ship-whole PROJ to site of ASG.
2. Ship-whole ASG to site of PROJ.
3. Fetch ASG tuples as needed for each tuple of PROJ.
4. Move ASG and PROJ to a third site.

The optimization algorithm predicts the total time of each strategy and selects the cheapest. Given that there is no operator following the join PROJ $\bowtie$ ASG, strategy 4 obviously incurs the highest cost since both relations must be transferred. If $size(\mathtt{PROJ})$ is much larger than $size(\mathtt{ASG})$, strategy 2 minimizes the communication time and is likely to be the best if local processing time is not too high compared to strategies 1 and 3. Note that the local processing time of strategies 1 and 3 is probably much better than that of strategy 2 since they exploit the index on the join attribute.

If strategy 2 is not the best, the choice is between strategies 1 and 3. Local processing costs in both of these alternatives are identical. If PROJ is large and only a few tuples of ASG match, strategy 3 probably incurs the least communication time and is the best. Otherwise, that is, if PROJ is small or many tuples of ASG match, strategy 1 should be the best. ♦

Conceptually, the algorithm can be viewed as an exhaustive search among all alternatives that are defined by the permutation of the relation join order, join methods (including the selection of the join algorithm), result site, access path to the internal relation, and intersite transfer mode. Such an algorithm has a combinatorial complexity in the number of relations involved. Actually, the algorithm significantly reduces the number of alternatives by using dynamic programming and the heuristics. With dynamic programming, the trie of alternatives is dynamically constructed and pruned by eliminating the inefficient choices.

Performance evaluation of the algorithm in the context of both high-speed networks (similar to local networks) and medium-speed wide area networks confirms the significant contribution of local processing costs, even for wide area networks. It is shown in particular that for the distributed join, transferring the entire internal relation outperforms the fetch-as-needed method.

4.5.3 Hybrid Approach

Dynamic and static query optimization both have advantages and drawbacks. Dynamic query optimization mixes optimization and execution and thus can make accurate optimization choices at runtime. However, query optimization is repeated for each execution of the query. Therefore, this approach is best for ad hoc queries. Static query optimization, done at compilation time, amortizes the cost of optimization over multiple query executions. The accuracy of the cost model is thus critical to predict the costs of candidate QEPs. This approach is best for queries embedded in stored procedures, and has been adopted by all commercial DBMSs.

However, even with a sophisticated cost model, there is an important problem that prevents accurate cost estimation and comparison of QEPs at compile time. The problem is that the actual bindings of parameter values in embedded queries are not known until runtime. Consider, for instance, the selection predicate WHERE R.A=$a, where $a is a parameter value. To estimate the cardinality of this selection, the optimizer must rely on the assumption of uniform distribution of A values in R and cannot make use of histograms. Since there is a runtime binding of the parameter a, the accurate selectivity of $\sigma_{\mathtt{A=\$a}}(\mathtt{R})$ cannot be estimated until runtime. Thus, it can make major estimation errors that can lead to the choice of suboptimal QEPs. In addition to unknown bindings of parameter values in embedded queries, sites may become unavailable or overloaded at runtime. Furthermore, relations (or relation fragments) may be replicated at several sites. Thus, site and copy selection should be done at runtime to increase availability and load balancing of the system.

Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates. The approach is basically static, but further optimization decisions may take place at runtime. A general solution is to produce *dynamic QEPs* which include carefully selected optimization decisions to be made at runtime using "choose-plan" operators. The choose-plan operator links two or more equivalent subplans of a QEP that are incomparable at compile time because important runtime information (e.g., parameter bindings) is missing to estimate costs. The execution of a choose-plan operator yields the comparison of the subplans based on actual costs and the selection of the best one. Choose-plan nodes can be inserted anywhere in a QEP. This approach is general enough to incorporate site and copy selection decisions. However, the search space of alternative subplans linked by choose-plan operators becomes much larger and may result in heavy static plans and much higher startup time. Therefore, several hybrid techniques have been proposed to optimize queries in distributed systems. They essentially rely on the following two-step approach:

1. At compile time, generate a static plan that specifies the ordering of operators and the access methods, without considering where relations are stored.
2. At startup time, generate an execution plan by carrying out site and copy selection and allocating the operators to the sites.

Example 4.19 Consider the following query expressed in relational algebra:

$$\sigma(\mathtt{R}_1) \bowtie \mathtt{R}_2 \bowtie \mathtt{R}_3$$

Figure 4.15 shows a two-step plan for this query. The static plan shows the relational operator ordering as produced by a centralized query optimizer. The runtime plan extends the static plan with site and copy selection and communication between sites. For instance, the first selection is allocated at site S_1 on copy $\mathtt{R}_{11}$ of relation $\mathtt{R}_1$ and sends its result to site S_3 to be joined with $\mathtt{R}_{23}$ and so on. ♦

The first step can be done by a centralized query optimizer. It may also include choose-plan operators so that runtime bindings can be used at startup time to make accurate cost estimations. The second step carries out site and copy selection, possibly in addition to choose-plan operator execution. Furthermore, it can optimize the load balancing of the system. In the rest of this section, we illustrate this second step.

We consider a distributed database system with a set of sites $S = \{S_1, .., S_n\}$. A query Q is represented as an ordered sequence of subqueries $Q = \{q_1, .., q_m\}$. Each subquery q_i is the maximum processing unit that accesses a single base relation and communicates with its neighboring subqueries. For instance, in Fig. 4.15, there are three subqueries, one for $\mathtt{R}_1$, one for $\mathtt{R}_2$, and one for $\mathtt{R}_3$. Each site S_i has a load, denoted by $load(S_i)$, which reflects the number of queries currently submitted. The load can be expressed in different ways, e.g., as the number of I/O bound and CPU bound queries at the site. The average load of the system is defined as:

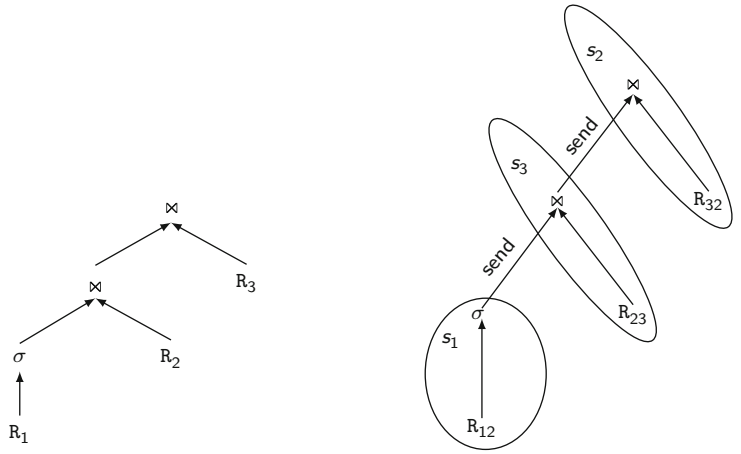

Fig. 4.15 A two-step plan. (**a**) Static plan. (**b**) Runtime plan

$$Avg\_load(S) = \frac{\sum_{i=1}^{n} load(S_i)}{n}$$

The balance of the system for a given allocation of subqueries to sites can be measured as the variance of the site loads using the following *unbalance factor*:

$$UF(S) = \frac{1}{n} \sum_{i=1}^{n} (load(S_i) - Avg\_load(S))^2$$

As the system gets balanced, its unbalance factor approaches 0 (perfect balance). For example, with $load(S_1) = load(S_2) = 0$, the unbalance factor of $\{S_1, S_2\} = 100$, while with $load(S_1)$ and $load(S_1)$, it is 0.

The problem addressed by the second step of two-step query optimization can be formalized as the following subquery allocation problem. Given

1. a set of sites $S = \{S_1, .., S_n\}$ with the load of each site;
2. a query $Q = \{q_1, .., q_m\}$; and
3. for each subquery q_i in Q, a feasible allocation set of sites $S_q = \{S_1, \ldots, S_k\}$ where each site stores a copy of the relation involved in q_i;

the objective is to find an optimal allocation on Q to S such that

1. $UF(S)$ is minimized, and
2. the total communication cost is minimized.

There is an algorithm that finds near-optimal solutions in a reasonable amount of time. The algorithm, which we describe in Algorithm 4.3 for linear join trees, uses several heuristics. The first heuristic (step 1) is to start by allocating subqueries with least allocation flexibility, i.e., with the smaller feasible allocation sets of sites. Thus, subqueries with a few candidate sites are allocated earlier. Another heuristic (step 2) is to consider the sites with least load and best benefit. The benefit of a site is defined as the number of subqueries already allocated to the site and measures the communication cost savings from allocating the subquery to the site. Finally, in step 3 of the algorithm, the load information of any unallocated subquery that has a selected site in its feasible allocation set is recomputed.

Example 4.20 Consider the following query Q expressed in relational algebra:

$$\sigma(\mathrm{R}_1) \bowtie \mathrm{R}_2 \bowtie \mathrm{R}_3 \bowtie \mathrm{R}_4$$

Figure 4.16 shows the placement of the copies of the 4 relations at the 4 sites, and the site loads. We assume that Q is decomposed as $Q = \{q_1, q_2, q_3, q_4\}$, where q_1 is associated with R_1, q_2 with R_2 joined with the result of q_1, q_3 with R_3 joined with the result of q_2, and q_4 with R_4 joined with the result of q_3. The SQAllocation

Algorithm 4.3: SQAllocation

Input: Q: $q_1, \ldots, q_m$
Feasible allocation sets: $F_{q_1}, \ldots, F_{q_m}$
Loads: $load(F_1), \ldots, load(F_m)$
Output: an allocation of Q to S
begin
  for *each* q *in* Q **do**
    compute($load(F_q)$)
  end for
  while Q *not empty* **do**
    {select subquery a for allocation}
    $a \leftarrow q \in Q$ with least allocation flexibility (1)
    {select best site b for a}
    $b \leftarrow f \in F_a$ with least load and best benefit (2)
    $Q \leftarrow Q - a$
    {recompute loads of remaining feasible allocation sets if necessary} (3)
    for *each* $q \in Q$ *where* $b \in F_q$ **do**
      compute($load(F_q)$)
    end for
  end while
end

Sites	Load	R_1	R_2	R_3	R_4
s_1	1	R_{11}		R_{31}	R_{41}
s_2	2		R_{22}		
s_3	2	R_{13}		R_{33}	
s_4	2	R_{14}	R_{24}		

Fig. 4.16 Example data placement and load

algorithm performs 4 iterations. In the first one, it selects q_4 which has the least allocation flexibility, allocates it to S_1, and updates the load of S_1 to 2. In the second iteration, the next subqueries to be selected are either q_2 or q_3 since they have the same allocation flexibility. Let us choose q_2 and assume it gets allocated to S_2 (it could be allocated to S_4 which has the same load as S_2). The load of S_2 is increased to 3. In the third iteration, the next subquery selected is q_3 and it is allocated to S_1 which has the same load as S_3 but a benefit of 1 (versus 0 for S_3) as a result of the allocation of q_4. The load of S_1 is increased to 3. Finally, in the last iteration, q_1 gets allocated to either S_3 or S_4 which have the least loads. If in the second iteration q_2 was allocated to S_4 instead of to S_2, then the fourth iteration would have allocated q_1 to S_4 because of a benefit of 1. This would have produced a better execution plan with less communication. This illustrates that two-step optimization can still miss optimal plans. ♦

This algorithm has reasonable complexity. It considers each subquery in turn, considering each potential site, selects a current one for allocation, and sorts the list of remaining subqueries. Thus, its complexity can be expressed as $\mathcal{O}(max(m * n, m^2 * log_2 m))$.

Finally, the algorithm includes a refining phase to further optimize join processing and decide whether or not to use semijoins. Although it minimizes communication given a static plan, two-step query optimization may generate runtime plans that have higher communication cost than the optimal plan. This is because the first step is carried out ignoring data location and its impact on communication cost. For instance, consider the runtime plan in Fig. 4.15 and assume that the third subquery on $\mathtt{R}_3$ is allocated to site S_1 (instead of site S_2). In this case, the plan that does the join (or Cartesian product) of the result of the selection of $\mathtt{R}_1$ with $\mathtt{R}_3$ first at site S_1 may be better since it minimizes communication. A solution to this problem is to perform plan reorganization using operator trie transformations at startup time.

4.6 Adaptive Query Processing

The underlying assumption so far is that the distributed query processor has sufficient knowledge about query runtime conditions in order to produce an efficient QEP and the runtime conditions remain stable during execution. This is a fair assumption for queries with few database relations running in a controlled environment. However, this assumption is inappropriate for changing environments with large numbers of relations and unpredictable runtime conditions.

Example 4.21 Consider the QEP in Fig. 4.17 with relations `EMP`, `ASG`, `PROJ`, and `PAY` at sites S_1, S_2, S_3, S_4, respectively. The crossed arrow indicates that, for some reason (e.g., failure), site S_2 (where `ASG` is stored) is not available at the beginning of execution. Let us assume, for simplicity, that the query is to be executed according to the iterator execution model, such that tuples flow from the left most relation.

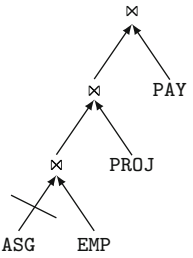

Fig. 4.17 Query execution plan with a blocked relation

Because of the unavailability of S_2, the entire pipeline is blocked, waiting for ASG tuples to be produced. However, with some reorganization of the plan, some other operators could be evaluated while waiting for S_2, for instance, to evaluate the join of EMP and PAY. ♦

This simple example illustrates that a typical static plan cannot cope with unpredictable data source unavailability. More complex examples involve continuous queries, expensive predicates, and data skew. The main solution is to have some adaptive behavior during query processing, i.e., *adaptive query processing*. Adaptive query processing is a form of dynamic query processing, with a feedback loop between the execution environment and the query optimizer in order to react to unforeseen variations of runtime conditions. A query processing system is defined as adaptive if it receives information from the execution environment and determines its behavior according to that information in an iterative manner.

In this section, we first provide a general presentation of the adaptive query processing process. Then, we present the eddy approach that provides a powerful framework for adaptive query processing.

4.6.1 Adaptive Query Processing Process

Adaptive query processing adds to the traditional query processing process the following activities: monitoring, assessing, and reacting. These activities are logically implemented in the query processing system by sensors, assessment components, and reaction components, respectively. Monitoring involves measuring some environment parameters within a time window, and reporting them to the assessment component. The latter analyzes the reports and considers thresholds to arrive at an adaptive reaction plan. Finally, the reaction plan is communicated to the reaction component that applies the reactions to query execution.

Typically, an adaptive process specifies the frequency with which each component will be executed. There is a trade-off between reactiveness, in which higher values lead to eager reactions, and the overhead caused by the adaptive process. A generic representation of the adaptive process is given by the function

$f_{adapt}(E, T) \rightarrow Ad$, where E is a set of monitored environment parameters, T is a set of threshold values, and Ad is a possibly empty set of adaptive reactions. The elements of E, T, and Ad, called adaptive elements, obviously may vary in a number of ways depending on the application. The most important elements are the monitoring parameters and the adaptive reactions. We now describe them.

4.6.1.1 Monitoring Parameters

Monitoring query runtime parameters involves placing sensors at key places of the QEP and defining observation windows, during which sensors collect information. It also requires the specification of a communication mechanism to pass collected information to the assessment component. Examples of candidates for monitoring are:

- Memory size. Monitoring available memory size allows, for instance, operators to react to memory shortage or memory increase.
- Data arrival rates. Monitoring the variation on data arrival rates may enable the query processor to do useful work while waiting for a blocked data source.
- Actual statistics. Database statistics in a distributed environment tend to be inaccurate, if at all available. Monitoring the actual size of relations and intermediate results may lead to important modifications in the QEP. Furthermore, the usual data assumptions, in which the selectivity of predicates over attributes in a relation is mutually independent, can be abandoned and real selectivity values can be computed.
- Operator execution cost. Monitoring the actual cost of operator execution, including production rates, is useful for better operator scheduling. Furthermore, monitoring the size of the queues placed before operators may avoid overloading.
- Network throughput. Monitoring network throughput may be helpful to define the block size to retrieve data. In a lower throughput network, the system may react with larger block sizes to reduce network penalty.

4.6.1.2 Adaptive Reactions

Adaptive reactions modify query execution behavior according to the decisions taken by the assessment component. Important adaptive reactions are:

- Change schedule: modifies the order in which operators in the QEP get scheduled. *Query scrambling* reacts by a *change schedule* of the plan to avoid stalling on a blocked data source during query evaluation. Eddy adopts finer reaction where operator scheduling can be decided on a tuple basis.
- Operator replacement: replaces a physical operator by an equivalent one. For example, depending on the available memory, the system may choose between a nested loop join or a hash join. Operator replacement may also change the plan by introducing a new operator to join the intermediate results produced by

a previous adaptive reaction. Query scrambling, for instance, may introduce new operators to evaluate joins between the results of *change schedule* reactions.
- Data refragmentation: considers the dynamic fragmentation of a relation. Static partitioning of a relation tends to produce load imbalance between sites. For example, information partitioned according to their associated geographical region may exhibit different access rates during the day because of the time differences in users' locations.
- Plan reformulation: computes a new QEP to replace an inefficient one. The optimizer considers actual statistics and state information, collected on the fly, to produce a new plan.

4.6.2 Eddy Approach

Eddy is a general framework for adaptive query processing over distributed relations. For simplicity, we only consider select-project-join (SPJ) queries. Select operators can include expensive predicates. The process of generating a QEP from an input SPJ query begins by producing an operator trie of the join graph G of the input query. The choice among join algorithms and relation access methods favors adaptiveness. A QEP can be modeled as a tuple $Q = \langle D, P, C \rangle$, where D is a set of database relations, P is a set of query predicates with associated algorithms, and C is a set of ordering constraints that must be followed during execution. Observe that multiple valid operator trees can be derived from G that obey the constraints in C, by exploring the search space with different predicate orders. There is no need to find an optimal QEP during query compilation. Instead, operator ordering is done on the fly on a tuple-per-tuple basis (i.e., tuple routing). The process of QEP compilation is completed by adding the *eddy operator* which is an n-ary operator placed between the relations in D and query predicates in P.

Example 4.22 Consider a three-relation query $Q = (\sigma_p(\texttt{R}) \bowtie \texttt{S} \bowtie \texttt{T})$, where joins are equijoins. Assume that the only access method to relation `T` is through an index on join attribute `T.A`, i.e., the second join can only be an index join over `T.A`. Assume also that σ_p is an expensive predicate (e.g., a predicate over the results of running a program over values of `T.B`). Under these assumptions, the QEP is defined as $D = \{\texttt{R}, \texttt{S}, \texttt{T}\}$, $P = \{\sigma_p(\texttt{R}), \texttt{R} \bowtie \texttt{S}, \texttt{S} \bowtie \texttt{T}\}$, and $C = \{\texttt{S} \prec \texttt{T}\}$. The constraint $\prec$ imposes `S` tuples to probe `T` tuples, based on the index on `T.A`.

Figure 4.18 shows a QEP produced by the compilation of query Q with eddy. An ellipse corresponds to a physical operator (i.e., either eddy operator or an algorithm implementing a predicate $p \in P$). As usual, the bottom of the plan presents the source relations. In the absence of a scan access method, the access to relation `T` is wrapped by the join $\texttt{S} \bowtie \texttt{T}$, thus does not appear as a source relation. The arrows specify pipeline dataflow following a producer–consumer relationship. Finally, an arrow departing from the eddy models the production of output tuples. ♦

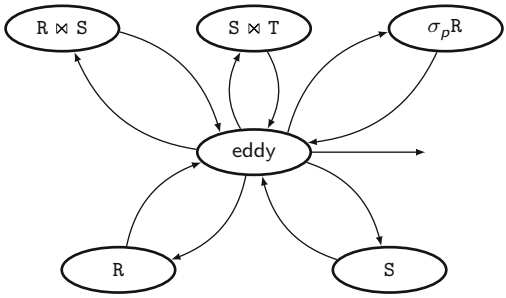

Fig. 4.18 A query execution plan with eddy

Eddy provides fine-grained adaptiveness by deciding on the fly how to route tuples through predicates according to a scheduling policy. During query execution, tuples in source relations are retrieved and staged into an input buffer managed by the eddy operator. Eddy responds to relation unavailability by simply reading from another relation and staging tuples in the buffer pool.

The flexibility of choosing the currently available source relation is obtained by relaxing the fixed order of predicates in a QEP. In eddy, there is no fixed QEP and each tuple follows its own path through predicates according to the constraints in the plan and its own history of predicate evaluation.

The tuple-based routing strategy produces a new QEP topology. The eddy operator together with its managed predicates forms a circular dataflow in which tuples leave the eddy operator to be evaluated by the predicates, which in turn bounce back output tuples to the eddy operator. A tuple leaves the circular dataflow either when it is eliminated by a predicate evaluation or the eddy operator realizes that the tuple has passed through all the predicates in its list. The lack of a fixed QEP requires each tuple to register the set of predicates it is eligible for. For example, in Fig. 4.18, S tuples are eligible for the two join predicates but are not eligible for predicate $\sigma_p(\mathtt{R})$.

4.7 Conclusion

In this chapter, we provided a detailed presentation of query processing in distributed DBMSs. We first introduced the problem of distributed query processing. The main assumption is that the input query is expressed in relational calculus since that is the case with most current distributed DBMS. The complexity of the problem is proportional to the expressive power and the abstraction capability of the query language.

The query processing problem is very difficult to understand in distributed environments because many elements are involved. However, the problem may be divided into several subproblems which are easier to solve individually. Therefore, we have proposed a generic layering scheme for describing distributed query

processing. Four main functions have been isolated: query decomposition, data localization, distributed optimization, and distributed execution. These functions successively refine the query by adding more details about the processing environment.

Then, we described data localization, with emphasis on reduction and simplification techniques for the four following types of fragmentation: horizontal, vertical, derived, and hybrid. The query produced by the data localization layer is good in the sense that the worse executions are avoided. However, the subsequent layers usually perform important optimizations, as they add to the query increasing detail about the processing environment.

Next, we discussed the major optimization issue, which deals with the join ordering in distributed queries, including alternative join strategies based on semijoin, and the definition of a distributed cost model.

We illustrated the use of the join and semijoin techniques in three basic distributed query optimization algorithms: dynamic, static, and hybrid. The static and dynamic distributed optimization approaches have the same advantages and disadvantages as in centralized systems. The hybrid approach is best in today's dynamic environments as it delays important decisions such as copy selection and allocation of subqueries to sites at query startup time. Thus, it can better increase availability and load balancing of the system. We illustrated the hybrid approach with two-step query optimization which first generates a static plan that specifies the operators ordering as in a centralized system and then generates an execution plan at startup time, by carrying out site and copy selection and allocating the operators to the sites.

Finally, we discussed adaptive query processing, to deal with the dynamic behavior of the local DBMSs. Adaptive query processing addresses this problem with a dynamic approach whereby the query optimizer communicates at runtime with the execution environment in order to react to unforeseen variations of runtime conditions.

4.8 Bibliographic Notes

There are several survey papers on query processing and query optimization in the context of the relational model. Graefe [1993] provides a detailed survey.

The iterator execution model which has been the basis for many query processor implementation was proposed in the context of the Volcano extensible query evaluation system [Graefe 1994]. The seminal paper on cost-based query optimization [Selinger et al. 1979] was the first to propose a cost model with database statistics (see Sect. 4.4.2) and the use of a dynamic programming search strategy (see Sect. 4.1.1). Randomized strategies, such as Iterative Improvement [Swami 1989] and Simulated Annealing [Ioannidis and Wong 1987] have been proposed to achieve a good trade-off between optimization time and execution time.

The most complete survey on distributed query processing is by Kossmann [2000] and deals with both distributed DBMSs and multidatabase systems. The

paper presents the traditional phases of query processing in centralized and distributed systems, and describes the various techniques for distributed query processing. Distributed cost models are discussed in several papers such as [Lohman et al., Khoshafian and Valduriez 1987].

Data localization is treated in detail by Ceri and Pelagatti [1983] for horizontally partitioned relations which are referred to as multirelations. The formal properties of horizontal and vertical fragmentation are used by Ceri et al. [1986] to characterize distributed joins over fragmented relations.

The theory of semijoins and their value for distributed query processing has been covered in [Bernstein and Chiu 1981], [Chiu and Ho 1980], and [Kambayashi et al. 1982]. The semijoin-based approach to distributed query optimization was proposed by Bernstein et al. [1981] for SDD-1 system [Wong 1977]. Full reducer semijoin programs are investigated by Chiu and Ho [1980], Kambayashi et al. [1982]. The problem of finding full reducers is NP-hard. However, for chained queries, a polynomial algorithm exists [Chiu and Ho 1980, Ullman 1982]. The cost of semijoins can be minimized by using bit arrays [Valduriez 1982]. Some other query processing algorithms aim at selecting an optimal combination of joins and semijoins [Özsoyoglu and Zhou 1987, Wah and Lien 1985].

The dynamic approach to distributed query optimization was first proposed for Distributed INGRES [Epstein et al. 1978]. The algorithm takes advantage of the network topology (general or broadcast networks) and uses the reduction algorithm [Wong and Youssefi 1976] that isolates all irreducible subqueries and monorelation subqueries by detachment.

The static approach to distributed query optimization was first proposed for R* [Selinger and Adiba 1980]. It is one of the first papers to recognize the significance of local processing on the performance of distributed queries. Experimental validation by Lohman et al. and Mackert and Lohman [1986a,b] have confirmed this important statement. The method fetch-as-needed of R* is called bindjoin in [Haas et al. 1997a].

A general hybrid approach to query optimization is to use choose-plan operators [Cole and Graefe 1994]. Several hybrid approaches based on two-step query optimization have been proposed for distributed systems [Carey and Lu 1986, Du et al. 1995, Evrendilek et al. 1997]. The content of Sect. 4.5.3 is based on the seminal paper on two-step query optimization by Carey and Lu [1986]. Du et al. [1995] propose efficient operators to transform linear join trees (produced by the first step) into bushy trees which exhibit more parallelism. Evrendilek et al. [1997] propose a solution to maximize intersite join parallelism in the second step.

Adaptive query processing is surveyed in [Hellerstein et al. 2000, Gounaris et al. 2002]. The seminal paper on the eddy approach which we used to illustrate adaptive query processing in Sect. 4.6 is [Avnur and Hellerstein 2000]. Other important techniques for adaptive query processing are query scrambling [Amsaleg et al. 1996, Urhan et al. 1998], ripple joins [Haas and Hellerstein 1999b], adaptive partitioning [Shah et al. 2003], and cherry picking [Porto et al. 2003]. Major extensions to eddy are state modules [Raman et al. 2003] and distributed eddies [Tian and DeWitt 2003].

Exercises

Problem 4.1 Assume that relation PROJ of the sample database is horizontally fragmented as follows:

$$\text{PROJ}_1 = \sigma_{\text{PNO} \leq \text{"P2"}}(\text{PROJ})$$

$$\text{PROJ}_2 = \sigma_{\text{PNO} > \text{"P2"}}(\text{PROJ})$$

Transform the following query into a reduced query on fragments:

```
SELECT ENO, PNAME
FROM   PROJ NATURAL JOIN ASG
WHERE  PNO = "P4"
```

Problem 4.2 (*) Assume that relation PROJ is horizontally fragmented as in Problem 4.1, and that relation ASG is horizontally fragmented as

$$\text{ASG}_1 = \sigma_{\text{PNO} \leq \text{"P2"}}(\text{ASG})$$

$$\text{ASG}_2 = \sigma_{\text{"P2"} < \text{PNO} \leq \text{"P3"}}(\text{ASG})$$

$$\text{ASG}_3 = \sigma_{\text{PNO} > \text{"P3"}}(\text{ASG})$$

Transform the following query into a reduced query on fragments, and determine whether it is better than the fragment query:

```
SELECT RESP, BUDGET
FROM   ASG NATURAL JOIN PROJ
WHERE  PNAME = "CAD/CAM"
```

Problem 4.3 (**) Assume that relation PROJ is fragmented as in Problem 4.1. Furthermore, relation ASG is indirectly fragmented as

$$\text{ASG}_1 = \text{ASG} \ltimes_{\text{PNO}} \text{PROJ}_1$$

$$\text{ASG}_2 = \text{ASG} \ltimes_{\text{PNO}} \text{PROJ}_2$$

and relation EMP is vertically fragmented as

$$\text{EMP}_1 = \Pi_{\text{ENO,ENAME}}(\text{EMP})$$

$$\text{EMP}_2 = \Pi_{\text{ENO,TITLE}}(\text{EMP})$$

vnine
Transform the following query into a reduced query on fragments:

```
SELECT ENAME
FROM   EMP NATURAL JOIN ASG NATURAL JOIN PROJ
WHERE  PNAME = "Instrumentation"
```

Problem 4.4 Consider the join graph of Fig. 4.11 and the following information: $size(\texttt{EMP}) = 100$, $size(\texttt{ASG}) = 200$, $size(\texttt{PROJ}) = 300$, $size(\texttt{EMP} \bowtie \texttt{ASG}) = 300$, and $size(\texttt{ASG} \bowtie \texttt{PROJ}) = 200$. Describe an optimal join program based on the objective function of total transmission time.

Problem 4.5 Consider the join graph of Fig. 4.11 and make the same assumptions as in Problem 4.4. Describe an optimal join program that minimizes response time (consider only communication).

Problem 4.6 Consider the join graph of Fig. 4.11, and give a program (possibly not optimal) that reduces each relation fully by semijoins.

Problem 4.7 (*) Consider the join graph of Fig. 4.11 and the fragmentation depicted in Fig. 4.19. Also assume that $size(\texttt{EMP} \bowtie \texttt{ASG}) = 2000$ and $size(\texttt{ASG} \bowtie \texttt{PROJ}) = 1000$. Apply the dynamic distributed query optimization algorithm in Sect. 4.5.1 in two cases, general network and broadcast network, so that communication time is minimized.

Problem 4.8 (**) Consider the following query on our engineering database:

```
SELECT ENAME,SAL
FROM   PAY NATURAL JOIN EMP NATURAL JOIN ASG
       NATURAL JOIN PROJ
WHERE  (BUDGET>200000 OR DUR>24)
AND    (DUR>24 OR PNAME = "CAD/CAM")
```

Assume that relations EMP, ASG, PROJ, and PAY have been stored at sites 1, 2, and 3 according to the table in Fig. 4.20. Assume also that the transfer rate between any two sites is equal and that data transfer is 100 times slower than data processing performed by any site. Finally, assume that $size(\texttt{R} \bowtie \texttt{S}) = max(size(\texttt{R}), size(\texttt{S}))$ for any two relations R and S, and the selectivity factor of the disjunctive selection of the query is 0.5. Compose a distributed program that computes the answer to the query and minimizes total time.

Relation	Site 1	Site 2	Site 3
EMP	1000	1000	1000
ASG		2000	
PROJ	1000		

Fig. 4.19 Fragmentation

Relation	Site 1	Site 2	Site 3
EMP	2000		
ASG		3000	
PROJ			1000
PAY			500

Fig. 4.20 Fragmentation statistics

Problem 4.9 (**) In Sect. 4.5.3, we described Algorithm 4.3 for linear join trees. Extend this algorithm to support bushy join trees. Apply it to the bushy join trie in Fig. 4.9 using the data placement and site loads shown in Fig. 4.16.

Problem 4.10 (**) Consider three relations `R(A,B)`, `S(B,D)` and `T(D,E)`, and query $Q(\sigma_p^1(\texttt{R}) \bowtie_1 \texttt{S} \bowtie_2 \texttt{T})$, where $\bowtie_1$ and $\bowtie_2$ are natural joins. Assume that `S` has an index on attribute `B` and `T` has an index on attribute `D`. Furthermore, σ_p^1 is an expensive predicate (i.e., a predicate over the results of running a program over values of `R.A`). Using the eddy approach for adaptive query processing, answer the following questions:

(a) Propose the set C of constraints on Q to produce an eddy-based QEP.
(b) Give a join graph G for Q.
(c) Using C and G, propose an eddy-based QEP.
(d) Propose a second QEP that uses State Modules. Discuss the advantages obtained by using state modules in this QEP.

Problem 4.11 (**) Propose a data structure to store tuples in the eddy buffer pool to help choosing quickly the next tuple to be evaluated according to user-specified preference, for instance, produce first results earlier.

Chapter 5
Distributed Transaction Processing

The concept of a *transaction* is used in database systems as a basic unit of consistent and reliable computing. Thus, queries are executed as transactions once their execution strategies are determined and they are translated into primitive database operations. Transactions ensure that database consistency and durability are maintained when concurrent access occurs to the same data item (with at least one of these being an update) and when failures occur.

The terms *consistent* and *reliable* in transaction definition need to be defined more precisely. We differentiate between *database consistency* and *transaction consistency*.

A database is in a *consistent state* if it obeys all of the consistency (integrity) constraints defined over it (see Chap. 3). State changes occur due to modifications, insertions, and deletions (together called *updates*). Of course, we want to ensure that the database never enters an inconsistent state. Note that the database can be (and usually is) temporarily inconsistent during the execution of a transaction. The important point is that the database should be consistent when the transaction terminates (Fig. 5.1).

Transaction consistency, on the other hand, refers to the operations of concurrent transactions. We would like the database to remain in a consistent state even if there are a number of user requests that are concurrently accessing (reading or updating) the database.

Reliability refers to both the *resiliency* of a system to various types of failures and its capability to *recover* from them. A resilient system is tolerant of system failures and can continue to provide services even when failures occur. A recoverable DBMS is one that can get to a consistent state (by moving back to a previous consistent state or forward to a new consistent state) following various types of failures.

Transaction management deals with the problems of always keeping the database in a consistent state even when concurrent accesses and failures occur. The issues in managing concurrent transactions are well-known in centralized DBMSs and can be found in many textbooks. In this chapter, we investigate these issues within

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_5

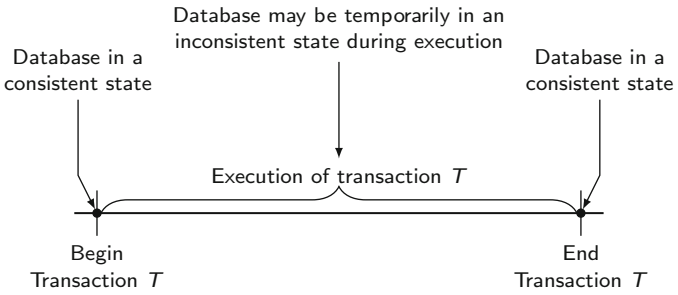

Fig. 5.1 A transaction model

the context of distributed DBMSs focusing on distributed concurrency control and distributed reliability and recovery. We expect that the reader has familiarity with the basic transaction management concepts and techniques as commonly covered in undergraduate database courses and books. We provide a brief refresher in Sect. 5.1. More detailed discussion of the fundamental transaction processing concepts is in Appendix C. For now, we ignore data replication issues; the following chapter is devoted to that topic. DBMSs are typically classified as On-Line Transaction Processing (OLTP) or On-Line Analytical Processing (OLAP). *On-Line Transaction Processing* applications, such as airline reservation or banking systems, are high-throughput transaction-oriented. They need extensive data control and availability, high multiuser throughput and predictable, fast response times. In contrast, *On-line Analytical Processing* applications, such as trend analysis or forecasting, need to analyze historical, summarized data coming from a number of operational databases. They use complex queries over potentially very large tables. Most OLAP applications do not need the most current versions of the data, and thus do not need direct access to most up-to-date operational data. In this chapter, we focus on OLTP systems and consider OLAP systems in Chap. 7.

The organization of this chapter is as follows. In Sect. 5.1 we provide a quick introduction to the basic terminology that is used in this chapter, and revisit the architectural model defined in Chap. 1 to highlight the modifications that are necessary to support transaction management. Section 5.2 provides an in-depth treatment of serializability-based distributed concurrency control techniques, while Sect. 5.3 considers concurrency control under snapshot isolation. Section 5.4 discusses distributed reliability techniques focusing on distributed commit, termination, and recovery protocols.

5.1 Background and Terminology

Our objective in this section is to provide a very brief introduction to the concepts and terminology that we use in the rest of the chapter. As mentioned previously, our

objective is not to provide a deep overview of the fundamental concepts—those can be found in Appendix C—but to introduce the basic terminology that will be helpful in the rest of the chapter. We also discuss how the system architecture needs to be revised to accommodate transactions.

As indicated before, a transaction is a unit of consistent and reliable computation. Each transaction begins with a Begin_transaction command, includes a series of Read and Write operations, and ends with either a Commit or an Abort. Commit, when processed, ensures that the updates that the transaction has made to the database are permanent from that point on, while Abort undoes the transaction's actions so, as far as the database is concerned, it is as if the transaction has never been executed. Each transaction is characterized by its *read set* (*RS*) that includes the data items that it reads, and its *write set* (*WS*) of the data items that it writes. The read set and write set of a transaction need not be mutually exclusive. The union of the read set and write set of a transaction constitutes its *base set* ($BS = RS \cup WS$).

Typical DBMS transaction services provide ACID properties:

1. *Atomicity* ensures that transaction executions are atomic, i.e., either all of the actions of a transaction are reflected in the database or none of it are.
2. *Consistency* refers to a transaction being a correct execution (i.e., the transaction code is correct and when it executes on a database that is consistent, it will leave it in a consistent state).
3. *Isolation* indicates that the effects of concurrent transactions are shielded from each other until they commit—this is how the correctness of concurrently executing transactions are ensured (i.e., executing transactions concurrently does not break database consistency).
4. *Durability* refers to that property of transactions that ensures that the effects of committed transactions on the database are permanent and will survive system crashes.

Concurrency control algorithms that we discuss in Sect. 5.2 enforce the isolation property so that concurrent transactions see a consistent database state and leave the database in a consistent state, while reliability measures we discuss in Sect. 5.4 enforce atomicity and durability. Consistency in terms of ensuring that a given transaction does not do anything incorrect to the database is typically handled by integrity enforcement as discussed in Chap. 3.

Concurrency control algorithms implement a notion of "correct concurrent execution." The most common correctness notion is *serializability* that requires that the history generated by the concurrent execution of transactions is equivalent to some serial history (i.e., a sequential execution of these transactions). Given that a transaction maps one consistent database state to another consistent database state, any serial execution order is, by definition, correct; if the concurrent execution history is equivalent to one of these orders, it must also be correct. In Sect. 5.3, we introduce a more relaxed correctness notion called *snapshot isolation* (SI). Concurrency control algorithms are basically concerned with enforcing different levels of isolation among concurrent transactions very efficiently.

When a transaction commits, its actions need to be made permanent. This requires management of *transaction logs* where each action of a transaction is recorded. The commit protocols ensure that database updates as well as logs are saved into persistent storage so that they are made permanent. Abort protocols, on the other hand, use the logs to erase all actions of the aborted transaction from the database. When recovery from system crashes is needed, the logs are consulted to bring the database to a consistent state.

The introduction of transactions to the DBMS workload along with read-only queries requires revisiting the architectural model introduced in Chap. 1. The revision is an expansion of the role of the distributed execution monitor.

The distributed execution monitor consists of two modules: a *transaction manager* (TM) and a *scheduler* (SC). The transaction manager is responsible for coordinating the execution of the database operations on behalf of an application. The scheduler, on the other hand, is responsible for the implementation of a specific concurrency control algorithm for synchronizing access to the database.

A third component that participates in the management of distributed transactions is the local recovery managers (LRM) that exist at each site. Their function is to implement the local procedures by which the local database can be recovered to a consistent state following a failure.

Each transaction originates at one site, which we will call its *originating site*. The execution of the database operations of a transaction is coordinated by the TM at that transaction's originating site. We refer to the TM at the originating site as the *coordinator* or the *coordinating TM*.

A transaction manager implements an interface for the application programs to the transaction commands identified earlier: Begin_transaction, Read, Write, Commit, and Abort. The processing of each of these commands in a nonreplicated distributed DBMS is discussed below at an abstract level. For simplicity, we concentrate on the interface to the TM; the details are presented in the following sections.

1. Begin_transaction. This is an indicator to the coordinating TM that a new transaction is starting. The TM does some bookkeeping by recording the transaction's name, the originating application, and so on, in a main memory log (called the *volatile log*).
2. Read. If the data item is stored locally, its value is read and returned to the transaction. Otherwise, the coordinating TM finds where the data item is stored and requests its value to be returned (after appropriate concurrency control measures are taken). The site where the data item is read inserts a log record in the volatile log.
3. Write. If the data item is stored locally, its value is updated (in coordination with the data processor). Otherwise, the coordinating TM finds where the data item is located and requests the update to be carried out at that site (after appropriate concurrency control measures are taken). Again, the site that executes the write inserts a log record in the volatile log.

4. Commit. The TM coordinates the sites involved in updating data items on behalf of this transaction so that the updates are made durable at every site. WAL protocol is executed to move volatile log records to a log on disk (called the *stable log*).
5. Abort. The TM makes sure that no effects of the transaction are reflected in any of the databases at the sites where it updated data items. The log is used to execute the undo (rollback) protocol.

In providing these services, a TM can communicate with SCs and data processors at the same or at different sites. This arrangement is depicted in Fig. 5.2.

As we indicated in Chap. 1, the architectural model that we have described is only an abstraction that serves a pedagogical purpose. It enables the separation of many of the transaction management issues and their independent and isolated discussion. In Sect. 5.2, as we discuss the scheduling algorithm, we focus on the interface between a TM and an SC and between an SC and a data processor. In Sect. 5.4 we consider the execution strategies for the commit and abort commands in a distributed environment, in addition to the recovery algorithms that need to be implemented for the recovery manager. In Chap. 6, we extend this discussion to the case of replicated databases. We should point out that the computational model that we described here is not unique. Other models have been proposed such as, for example, using a private workspace for each transaction.

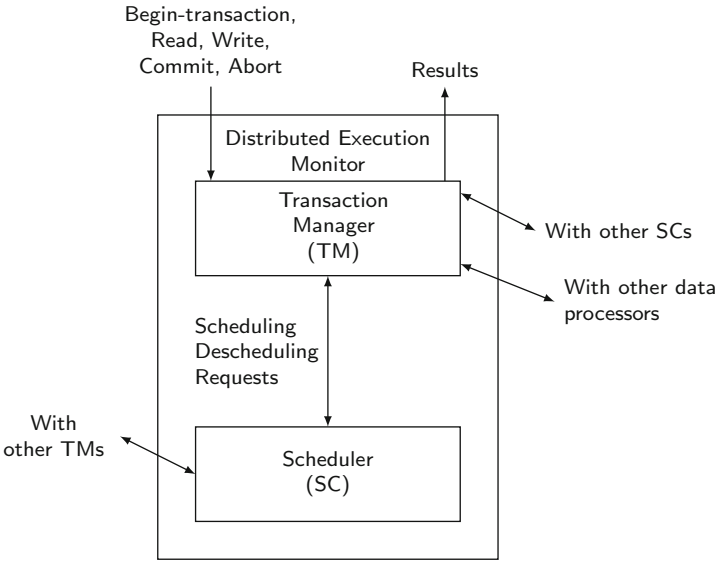

Fig. 5.2 Detailed model of the distributed execution monitor