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.

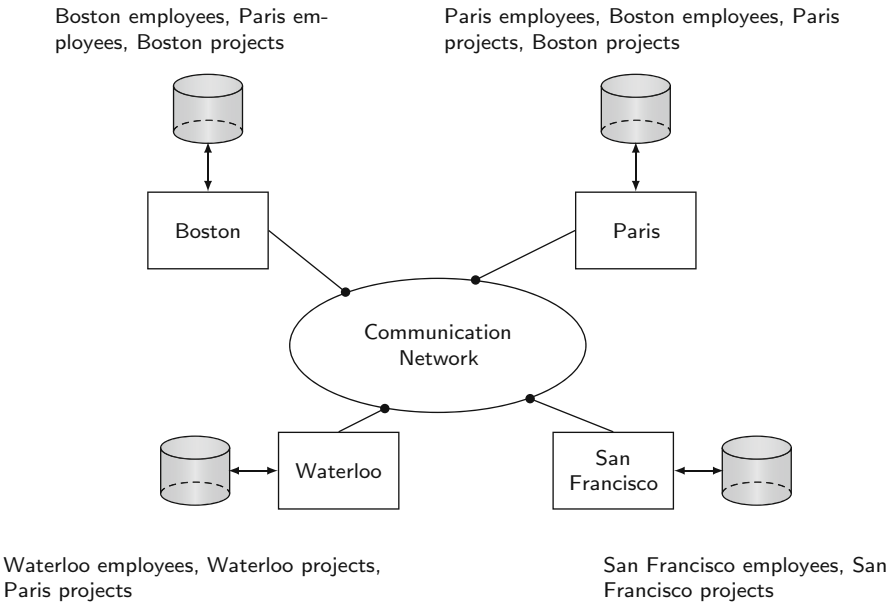

Fig. 1.5 Distributed database

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*

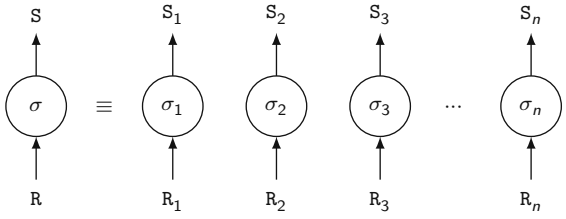

Fig. 1.6 Intraoperator parallelism. σ_i is instance i of the operator; n is the degree of parallelism

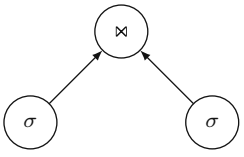

Fig. 1.7 Interoperator parallelism

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.