DATABASE PROCESSING

FUNCTIONAL LAYERS IN DATABASE SYSTEMS

The practical need for efficient organization, creation, manipulation, and maintenance of large collections of information, together with the recognition that data about the real world, which is manipulated by application programs, should be treated as an integrated resource independently of these programs, has led to the development of database management. In brief, a *database system* consists of a piece of software, the *database management system*, and some number of *databases*. The former is a special-purpose program stored in a computer's main memory and executed under the control of the operating system. Due to their size, which can easily reach terabyte (TB) or even petabyte (PB) ranges, databases are commonly stored in secondary memory, and the system then acts as an interface between users and their databases. It ensures that users can access data conveniently, efficiently, and under centralized control and that the data itself is resilient against hardware crashes and software errors and persists over long periods of time independent of the programs that access it.

We can view the functionality of a database management system as being organized in six different layers as shown in Fig. 1. The *language and API layer* manages the interfaces to the various classes of users, including the database administrator, casual users, and application programmers and application programs (e.g., Web servers). These interfaces may be menu-based, graphical, language-, or forms-based and provide a data definition language (DDL) and a data manipulation language (DML, e.g., structured query language [SQL]) as stand-alone languages or as languages embedded in host languages. Often, queries to a database are not explicitly specified by a user, but generated by a system based on user input; for example, Web servers often "talk" to a database system by sending off a query in a request and obtaining an answer in a reply. The *query processing and optimization processing layer* has to process the various forms of requests and queries that can be sent to a database. To this end, views used in a query need to be resolved (replaced by their definition), semantic integrity predicates are added if applicable, and access authorization is checked. Ad hoc queries are processed by an interpreter, and queries embedded in a host language program are compiled. Next, a query is decomposed into elementary database operations. Then the resulting sequence is optimized with the goal of avoiding executions with poor performance. An executable query or program ("access plan" or query execution plan) is passed to the *transaction management layer*, which is in charge of controlling concurrent accesses to a shared database ("concurrency control") and at the same time makes the system resilient against possible failures (through main-memory buffer management and through logging and recovery). At the *query execution layer*, individual queries are executed based on the execution plan created earlier, and subject to concurrency control and recovery mechanisms. Query execution will typically access stored data through the *access layer*, at which index structures for speeding up storage access are maintained. Finally, the *storage management layer* takes care of the physical data structures (files, pages, indexes) and performs disk accesses. In this context secondary storage is used for holding the database itself and also for keeping *logs* that allow restoring a consistent state after a crash ("recovery") and for keeping the *data dictionary* in which schema information is collected.

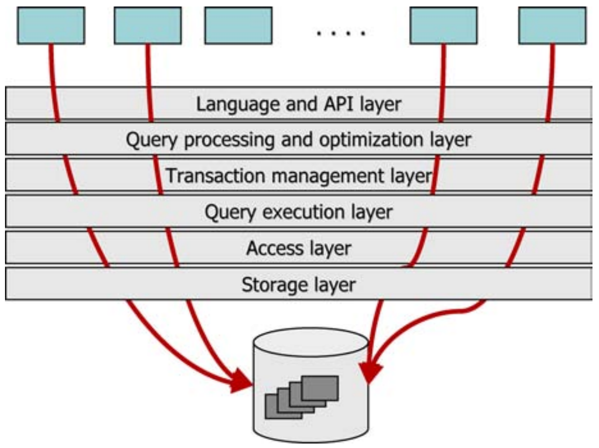

Figure 1. Functional DBMS layers.

RELATIONAL DATABASES

Relational database systems are based on a simple conceptual model introduced by Codd in (1) that allows for easy-to-use, yet powerful query languages. In particular, tables are accessed and manipulated via operators that process them as a whole. The most relevant of these will be described later. The model is based on the mathematical notion of a *relation* and organizes data in the form of *tables*. A table has *attributes* describing properties of data objects (as headline) and *tuples* holding data values (as other rows). In essence, a table is a *set of tuples*, whose tuple components can be identified by their associated attributes. This was originally restricted at least from the point of view of *types* in programming languages, because it allows only the application of a tuple *constructor* to attributes and given base domains, followed by the application of a set constructor; on the other hand, the simplicity of the relational model allows an elegant and in-depth formal treatment, which can be found, for example, in Maier (2) or Ullman (3). Codd's relational model has meanwhile undergone various extensions, in particular in the direction of object- as well as document orientation; a discussion of object-relational or XML-based systems is, however, beyond the scope of this paper; the reader is referred to Elmasri et al. (4) or Silberschatz et a. (5) for details. Most notably, the relational model has been combined with object-orientation into what is now known as *object-relational* databases, with powerful typing and querying facilities that smoothly integrate with modern programming languages.

A sample relational database is shown in Fig. 2. This database, which illustrates a banking application and will be used as our running example, comprises three tables

Account	AID	CustName	Balance
	110	Smith	1,324
	120	Jones	256
	130	Maier	22,345
	220	Paul	2,550
	240	Kline	86
	310	Adams	1,360
	320	Kelly	23,022
	333	Barbara	345

Branch	BID	AID	CityDistrict
	1	110	downtown
	1	120	downtown
	1	130	downtown
	2	220	beach area
	2	240	beach area
	3	310	lower east
	3	320	lower east
	3	330	lower east

Customer	Name	Address	District	YearOfBirth
	Adams	4680 Regents Rd.110	lower east	1955
	Barbara	240 Laurel St.	downtown	1963
	Jones	105 Boyer St.	downtown	1949
	Kelly	2360 Kings Ave.	lower east	1970
	Kline	240 Airport Blvd.	downtown	1961
	Maier	200 Laurel St.	downtown	1953
	Paul	2400 Mountain Rd.	northern valley	1965
	Smith	53 5th Ave.	downtown	1936

Figure 2. Sample relational database for a bank.

called *Account* (A), *Branch* (B), and *Customer* (C). Table A shows account balances for customer accounts. Individual accounts are identified by an ID. Table B indicates which accounts are kept at which branch of the bank and in which city district that branch, identified by a branch ID, is located. Finally, Table C holds customer information. Note that customers do not necessarily have their bank accounts in the city district where they live.

To be able to look at several aspects of the relational model in a formally precise manner, we next introduce some notation commonly used in this context. Let $X = \{A_1, \ldots, A_m\}$ be a set of *attributes*, and let each attribute $A \in X$ have a non-empty, finite domain dom(A) of atomic values (integers, strings, etc.). (Note that here attribute A is not to be confused with the table name we are using in the sample relational database shown in Fig. 1.) Let $\cup_{A \in X} \mathrm{dom}(A) =: \mathrm{dom}(X)$. A *tuple* over X is a mapping $\mu : X \to \mathrm{dom}(X)$ satisfying $\mu : (A) \in \mathrm{dom}(A)$ for each $A \in X$. For a given X, let Tup(X) denote the set of all tuples over X. A *relation* r over X is a finite set of tuples over X, that is, $r \subseteqq \mathrm{Tup}(X)$. The set of all relations over X is denoted by Rel(X).

A *relation schema* has the form $R = (X, \Sigma)$. It consists of a name (R), a set of attributes X, and a set Σ of local integrity constraints. It serves as a description of the set Sat(R) of all relations over X that satisfy Σ. Next let $\mathbf{R} = \{R_1, \ldots, R_k\}$ be a (finite) set of relation schemas, where $R_i = (X_i, \Sigma_i)$, $1 \le i \le k$, and $X_i \ne X_j$ for $i \ne j$. A (relational) *database* d (over $\mathbf{R}$) is a set of (base) relations, $d = \{r_1, \ldots, r_k\}$, such that $r_i \in \mathrm{Sat}(R_i)$ for $1 \le i \le k$. Let Dat($\mathbf{R}$) denote the set of all databases over $\mathbf{R}$.

Now let $\mathbf{R}$ be a set of relation schemas, and let $\Delta_{\boldsymbol{R}}$ be a set of global integrity constraints. A (relational) *database schema* is a named pair $\boldsymbol{D} = (\boldsymbol{R}, \Delta_{\boldsymbol{R}})$. It represents a description of the set of all *consistent* databases over $\mathbf{R}$ satisfying all local and global constraints. In our running example, a local constraint on Table A would say, for example, that the IDs used for identifying accounts have to be unique. A global constraint on the database would say that customer names in Table A must have a counterpart in Table C, that is, full information is recorded in Table C for each customer with an account. We do not pay any further attention to integrity constraints here since they are mostly used in the *design phase* of a database.

Operations on Relations

We next look at the operational part of the relational model, namely *algebraic operations* on relations. These operations directly provide the formal semantics of a relational data manipulation language known as *relational algebra*. Among their common features are that they all yield new (derived) relations from given ones and that they can be computed efficiently. We define only three algebraic operations on relations here: selection (S), projection (P), and natural join (J). These form the important subclass of *SPJ expressions* for which a host of interesting results is available in the literature; see Maier (2), Ullman (3) or Elmasri and Navathe (4) for details. The first two of these operations are unary, and the third is binary. *SPJ* expressions suffice to demonstrate important concepts in query processing and in query optimization.

Projections and Selections. The two unary operations we define next can be used to "cut" a table in a vertical (projection) or horizontal (selection) direction. Let $R = (X, \Sigma)$ be a relation schema, $R \in \mathrm{Rel}(X)$, $Y \subseteqq X$, and for a tuple μ over X, let $\mu[Y]$ denote its restriction onto Y. Then the *projection* of r onto Y is defined as follows:

$$\pi_Y(r) := \{\mu[Y] | \mu \text{ in } r\} \tag{1}$$

Next, the *selection* of r with respect to condition C is defined as

$$\sigma_C(r) := \{\mu \text{ in } r | \mu \text{ satisfies } C\} \tag{2}$$

Here, conditions may be either a term of the form $A\Theta a$ or $A\Theta B$, where $A, B \in X$, A, A and B have the same domain, $a \in \text{dom}(A)$, $\Theta \in \{\{<\}, \leq, >, \geq, =, \neq\}$, or several such terms connected by logical $\wedge$, and $\neg$. As an example, consider relation C from our running example. A projection of C onto attributes Name and YearOfBirth, that is, $\pi_{\text{Name, YearOfBirth}}(\text{C})$, will yield the following table:

Name	YearOfBirth
Adams	1955
Barbara	1963
Jones	1949
Kelly	1970
Kline	1961
Maier	1953
Paul	1965
Smith	1936

In the standard relational query language SQL this query is expressed as

```
select distinct Name, YearOfBirth
from C
```

where "distinct" is needed for removing duplicate tuples from the result. Similarly, a selection of all downtown accounts from Table B, that is, $\sigma_{\text{CityDistrict='downtown'}}(\text{B})$, results in the following table:

BID	AID	CityDistrict
1	110	downtown
1	120	downtown
1	130	downtown

In SQL this is written as

```
select*
from B
whereCityDistrict = 'downtown'
```

Notice that a number of simple rules apply to the selection and projection operations. Let r be a relation in Rel(X):

1. If $Z \subseteqq Y \subseteqq X$, then $\pi_Z[\pi_Y(r)] = \pi_Z(r)$;
2. if $Z, Y \subseteqq X$, and $Z \cap Y \neq 0$, then $\pi_Z[\pi_Y(r)] = \pi_{Z\cap Y}(r)$;
3. $\sigma_{C_1}[\sigma_{C_2}(r)] = \sigma_{C_2}[\sigma_{C_1}(r)]$ for any selection conditions C_1 and C_2;
4. if $A \in Y \subseteqq X$, then $\pi_Y[\sigma_{A\Theta a}(r)] = \sigma_{A\Theta a}[\pi_Y(r)]$, where Θ is an admissible comparison operator.

Natural Join. Next, we introduce the binary operation of the *natural join*. Intuitively, this type of join combines two relations into a new one, by looking for equal values for common attributes. The resulting relation has all attributes from both operands, where common ones are taken only once. Formally, we have the following. Let $r \in \text{Rel}(X)$, $s \in \text{Rel}(Y)$:

$$r \bowtie s := \{\mu \in \text{Tup}(X \cup Y) | \mu[X] \in r \wedge \mu[Y] \in s\} \qquad (3)$$

As an example, consider A ⋈ B in our sample database from Fig. 2, which is formed over the (only) common attribute AID. The result is the following relation:

AID	CustName	Balance	BID	CityDistrict
110	Smith	1,324	1	downtown
120	Jones	256	1	downtown
130	Maier	22,345	1	downtown
220	Paul	2,550	2	beach area
240	Kline	86	2	beach area
310	Adams	1,360	3	lower east
320	Kelly	23,022	3	lower east
330	Barbara	345	3	lower east

In SQL, this result is obtained from the following expression:

```
select A.AID, Cust, Balance, BID, CityDistrict
from A, B
where A.AID = B.AID
```

In other words, an SQL formulation requires a specification of the attributes of the result relation (in the select clause) and also that of a join condition (in the where clause). Variations of these rules brought along by modern SQL implementation, which result in simpler ways to state joins, are neglected here. Notice that by using explicit join conditions, it is possible to formulate more general joins, such as a join of Tables A and C over customer names (using the join condition "A.CustName = C.Name") or a join of B and C over city districts. It is even possible to specify join conditions with comparison operators other than equality. Without the possibility of equating attributes CustName and Name, a join of A and C would formally result in a Cartesian product of the operands, because all their attributes have pairwise distinct names. In general, a natural join degenerates to a Cartesian product (intersection) if the sets of attributes of the operands are disjoint (identical), respectively.

The operation of a natural join has additional properties. For example, we easily see that it is commutative and associative. In other words, the expressions (A ⋈ B) C, A ⋈ (B ⋈ C), (B ⋈ A) ⋈ C, (C ⋈ B) ⋈ A, C ⋈ (B ⋈ A) or A ⋈ (C ⋈ B) and so on all yield the same result (up to the ordering of attributes, which is considered immaterial), independent of the current contents of the operand tables. Additional rules for algebraic operations state under which conditions selection and projection distribute over joins. We list some of these rules next. Again let $r \in \text{Rel}(X)$, $s \in \text{Rel}(Y)$, and $t \in \text{Rel}(Z)$:

1. $\sigma_C(r \bowtie s) = \sigma_C(r) \bowtie s$ if the attributes mentioned in C are a subset of X;
2. $\sigma_C(r \bowtie s) = \sigma_C(r) \bowtie \sigma_C(s)$ if the attributes mentioned in C are a subset of X and Y;
3. if $V \subseteqq X \cup Y$, then $\pi_V(r \bowtie s) = \pi_V[\pi_W(r) \bowtie \pi_U(s)]$, where $W = X \cap (V \cup Y)$, $U = Y \cap (V \cup X)$.

For more on this topic, see Maier (2) or Silberschatz et al. (5). The typical way to prove such equalities (of sets of tuples) is to show set containment in both directions (i.e., "left-hand side $\subseteqq$ right-hand side" and vice versa), which is basically derived from the definitions of the operations involved.

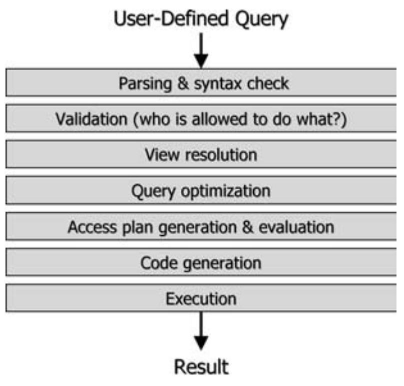

Figure 3. Steps in query processing.

Query Processing

Although we are not going to define the query language of relational algebra formally here, we mention that database queries, as shown by the previous SQL expressions, are always formulated in terms of the underlying schema. On the other hand, their evaluation takes place on the current contents of the respective database, and to this end, it is important that query evaluation can be done efficiently. Efficiency is determined by two aspects. First, the individual operations included in a query language must be efficiently implemented. We will see later on that this is particularly true for the operations introduced so far. Second, even if the operations used in a query are locally efficient, the overall query could be improvable, so that globally high efficiency can also be guaranteed. This latter aspect, called query *optimization*, is discussed next.

Figure 3surveys query processing in its entirety and the role of query optimization in it. A user-defined query is first parsed and checked for correct syntax. Next, validation ensures that the user who issued the query has the appropriate access rights, that attributes mentioned in the query indeed occur in the relations the user has specified, or that selection conditions use applicable comparisons. If view names occur as abbreviation for subqueries in the query, these are resolved, i.e., replaced by their defining expressions. Then the query is subject to optimization. Depending on the storage structures used at the internal level and the implementation available for the language operators, different access plans may be generated and evaluated w.r.t. execution costs, one of which is finally transformed into executable code and executed on the current state of the database.

Query Optimization. Consider the following query to our sample database: select the account balances of customers who were born before 1950 and do their banking downtown. Intuitively, this query needs to join together information from all three relations in our database:

1. account balances are found in Table A;
2. branch districts are stored in Table B; and
3. birth years of customers are available from Table C.

A corresponding SQL formulation is as follows:

```
select Balance
from C, B, A
where A.AID = B.AID and A.Cust = C.Name
and YearOfBirth < 1950
and CityDistrict = 'downtown'
```

An SQL query processor evaluates this expression by forming first the product of the operands mentioned in the from clause, then applying the conditions given in the where clause, and finally projecting onto the attributes mentioned in the select clause. Thus, it evaluates the following algebraic expression (which assumes that A.CustName and C.Name are equated):

$$\pi_{Balance}(\sigma_{YearOfBirth \leq 1950 \wedge CityDistrict='downtown'}(C \bowtie B \bowtie A))$$

Notice that the inner join produces an intermediate result that is much larger than the final result. Indeed, the join of A, B, and C contains 8 tuples, whereas the final result consists only of the following two tuples:

Balance
1,324
256

(Only customers Smith and Jones qualify.) Fortunately, there is a straightforward way to do better in terms of query evaluation, namely, to apply the rules of relational algebra so that selections and projections are applied *as early as possible*. For example, because we are interested only in customers born before 1950, it would be reasonable to apply this selection condition to Table C *before* it is joined with another table. Similarly, the section on CityDistrict could be applied before any join. The resulting expression now reads as follows:

$$\pi_{Balance}(\sigma_{YearOfBirth < 1950}(C) \bowtie_{CityDistrict='downtown'}(B) \bowtie A)$$

Clearly, we can still do better by projecting out as many attributes as possible based on what is relevant to the query result or needed for proper join computations. The important point is that already the previous expression would never yield an intermediate result that contains more tuples than the final result, simply because those customers born before 1950 are selected right away.

The aspect illustrated in this example is one facet of the wide field of *query optimization*, which, when performed at the schema level, attempts to reduce the evaluation or response time *without* knowing the internal data structures of the relations involved. Clearly, another such facet is *implementation-dependent* query optimization, which is at least equally important for system implementations. Again, the goal is to avoid bad execution time, now, by characterizing them via *cost functions* based on the "selectivity" of operations, the availability of indexes, or the implementation chosen for the operations. The interested reader should consult Yao (6), Elmasri and Navathe (4), Silberschatz et al. (5), or Freytag et al. (7).

Internal Join Processing. Next we look at two ways to implement the join operator because this gives an impression of what can be done beyond the type of optimization discussed previously. To this end, it is first reasonable to clarify the role of an algebra and its operators within the "landscape" of languages used to query databases. As shown in

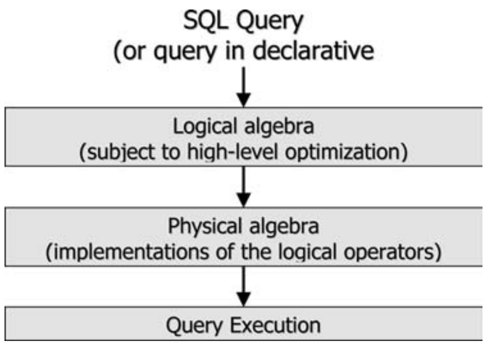

Figure 4. Role of query algebras.

Fig. 4, at the user level there is a typical SQL or another form of a declarative language (e.g., a rule-based language such as Datalog). Relational algebra can be seen as an "assembly language" for relational databases, because typically it does not pop up at the user interface. Instead, high-level queries are transformed by the system into an expression from the algebra to apply optimization techniques and to perform an efficient evaluation. However, although optimization is done to a *logical algebra* that corresponds to what we have described previously, implementation and execution is done in terms of a *physical algebra* that corresponds directly to system processes run on the bare machine.

We explain the difference between the two types of algebras by way of the join example. As seen earlier, the natural join has a formally precise definition, which on the one hand, tells what to expect from applying this operation to given tables, but which on the other hand, does not state how to implement the operation efficiently. The latter is treated in the context of physical join operators, of which we mention the *nested-loop* join and the *sort-merge* join.

The nested-loop join is immediately based on the join definition: A join of relations r and s is computed by fixing one relation, say r, as the "outer" relation, the other as the "inner", and searching for each new tuple of the outer relation to determine whether there exists a tuple in the inner relation with equal values for common attributes. We can write this as follows, where $Z := X \cap Y$:

```
initialize Result to the empty set;
for i: = 1 to |r| do
for j: = 1 to |s| do
if Tuple i of r matches Tuple j of s on Z
then add join of the 2 tuples to Result;
```

We mention that an implementation of this procedure needs to read both operands from secondary memory into the *database buffer* in the main memory to perform the equality tests. Correspondingly, the result relation is constructed in the main memory and eventually written back to the disk in case it is to be stored for further processing. This reveals an important observation we will study in more detail later when we talk about transaction processing: *read and write operations* are at the interface between the database and its portion that is currently being processed in the main-memory buffer. As seen later, these operations, if issued by distinct processes, require careful synchronization, so as not to corrupt the database or the results to be returned to the user.

Now it is easy to calculate the number of accesses a nested-loop join must perform on relations r and s of size n and m, respectively. The number of tuples to be handled is roughly $O(nm)$, because every tuple of s is compared for each tuple of r. If, for example, $|r| = 10^5$ and $|s| = 5 * 10^4$, this will result in $5*10^9$ tuple accesses. Real-world systems will normally improve on this by accessing relations in *blocks* that hold more than one tuple at a time. For example, if a block holds 10^2 tuples, the number of accesses drops to approximately 10^7.

The *sort-merge join* is a different method for computing a join. As the name suggests, this method proceeds by first sorting both operands, for example, according to increasing values for their common attributes. In a second phase, both are then scanned sequentially, and tuples satisfying the join condition are joined and added to the result. We can write this as follows, where as before Z denotes the common attributes, comparison is done in lexicographical order, index i ranges over relation r, and index j ranges over relation s:

```
sort r according to increasing values of common at-
tributes;
sort s correspondingly;
initialize Result to the empty set;
i: = 1;
j: = 1;
repeat
while Tuple j of s does not match Tuple i of r on Z do
while j < |s| do j: = j+1;
if Tuple i of r matches Tuple j of s on Z
then add join of the 2 tuples to Result;
i: = i+1
until i > |r|;
```

If r is of size n and s of size m, sorting requires the additional effort of $O(n \log n)$, respectively, $O(m \log m)$, but from then on the time complexity reduces to $O(n + m)$, which is proportional to the size of the bigger relation. Again, the number of accesses can be reduced even further by utilizing the fact that more than one tuple often fits into one block. We mention that similar considerations apply to selection and projection implementations. As is easily verified, writing physical operators for them is even easier than for the natural join.

Considerations like the previous are crucial for database system implementation. Therefore we refer the interested reader to Graefe (8), O'Neil and O'Neil (9), and Ramakrishnan and Gehrke (10). Issues like block sizes for appropriate units of transfer between secondary and primary memory, buffer size in main memory, or storage structures to manage secondary memory are subject to physical database design, but also subject to database *tuning;* see Shasha and Bonnet (11) for details.

TRANSACTION MANAGEMENT

Now we move to the system layer of a database system, at which translated user requests finally are executed. We have already described this for single queries. In particular, we have mentioned that read and write operations occur at the interface between a database (on disk) and the portion (in buffer) which is accessed to answer user requests. In general, a database system must perform queries, as described previously, but also update operations which insert, delete, or modify tuples, and more generally it may have to run application programs in which read and write operations to the database occur frequently.

If several such processes are interleaved arbitrarily, incorrect results may be produced, as seen from the following two examples, known as the *lost update* and the *inconsistent read* problem, respectively. First, consider two system processes P_1 and P_2 which are concurrently executed as follows:

P_1	Time	P_2
read(x)	1	
	2	read(x)
update(x)	3	
	4	update(x)
write(x)	5	
	6	write(x)

↑
update "lost"

Suppose that x is a numerical data object having a value of 10 at Time 1. Both P_1 and P_2 read this value. Assume that P_1 adds 1, whereas P_2 adds 2. So in the end x should have a value of 13. However, because P_2 updates the *original* value of x, the final value is 12, which is incorrect. Indeed, if P_1 writes its new value back into the database before P_2 does, the former update is lost.

Second, consider three numerical objects x, y, z with current values$x = 40$, $y = 50$, $z = 30$, that is, $x + y + z = 120$. This could arise in banking, where the objects represent account balances. For transfers between accounts, their sum obviously remains constant. Process 1 following computes the current sum, and process 2 transfers a value of 10 from z to x as follows:

P_1	Time	P_2
sum: = 0	1	
read(x)	2	
read(y)	3	
sum: = sum +x	4	
sum: = sum +y	5	
	6	read(z)
	7	$z: = z - 10$
	8	write(z)
	9	read(x)
	10	$x: = x - 10$
	11	write(x)
read(z)	12	
sum: = sum +z	13	

Clearly, process 1 returns 110 as a result, which is wrong. However, this error cannot be recognized by a user.

To allow users shared access to a common database, database systems know the concept of a *transaction*, which goes back to the work of Gray (12, 13). The basic idea is to consider a given program that wants to operate on a database as a logical unit and to process it as if the database were at its exclusive disposal. Now we describe what needs to be done, in particular from a conceptual point of view, to make this work.

The ACID Principle

If a database system allows multiple users shared access to a database, various conflicting goals have to be met in general: good throughput, shielding one user program from the others, avoidance of data losses or corruption, etc. To meet these goals, each individual user program is treated as a transaction by the system and processed so that the following properties are fulfilled:

- *Atomicity:* To the issuing user, it always appears that the user's transaction is executed *either completely or not at all*. Thus, the effects of the transaction on the database become visible to other transactions only if it terminates successfully and no errors have occurred in the meantime.
- *Consistency:* All integrity constraints of the database are maintained by each transaction, that is, a transaction always maps a consistent database state to another such state.
- *Isolation:* Each individual transaction is isolated from all others. Thus, each transaction is guaranteed to see only consistent data from the database.
- *Durability:* If a transaction has terminated normally, its effects on the database are guaranteed to survive subsequent failures.

These properties are collectively known as the *ACID principle*. To achieve them, the transaction processing component of a database management system has a *concurrency control* and a *recovery* component. In brief, the goal of concurrency control is to synchronize concurrent accesses to a shared database, and that of recovery is to restore a consistent state after a failure and to provide a guarantee that transaction results are durable.

Read-Write Transactions, Schedules, and Histories

To design concurrency control and recovery mechanisms, it is necessary to come up with a suitable *model* of transactions and their *executions*, to establish a notion of *correctness* of executions, and to devise *protocols* which achieve that. For simplicity, we stick to the model of *read-write transactions* here. As we have indicated when discussing the issue of query processing, this model is in some sense adequate, although it obviously abstracts from a number of (semantic) issues. The reader is also referred to Weikum and Vossen (14) for more motivation, but also for a more sophisticated transaction model.

If several transactions or read–write programs are run sequentially or one after the other, synchronization problems generally do not arise. Indeed, if each transaction preserves the consistency of the database on which it operates, the same is true for any sequential, or *serial*, execution of multiple transactions. Therefore, it makes sense to relate the correctness criterion for concurrent executions to serial executions, commonly captured by the term *serializability*. In other words, an execution of multiple transactions, also called a *schedule* for those transactions, is considered correct if it is "serializable," or equivalent to some serial execution of the same transactions.

Transactions and their executions in the read–write model of computation are described formally as follows: The underlying database is considered a countably infinite set $D = \{x, y, z, \ldots\}$ of objects, which are assumed to be pages or blocks that are read or written in one step and that are atomically transferred back and forth between primary and secondary memory. A single *transaction* has the form $t = p_1 \ldots p_n$, where each p_i has the form $r(x)$ ("read x") or $w(x)$ ("write x") for some $x \in D$. In the presence of several transactions we use indices to distinguish them. We assume that each transaction reads or writes every database object on which it operates at most once and that reading of an object is done before writing when both operations are desired.

A *history* for transactions $t_1, \ldots, t_n$ is an ordering of all operations of these transactions, which respects the order of operations specified by the transactions (formally called the "shuffle product" of the given transactions), and additionally contains a pseudostep for each transaction following its last operation that states whether this transaction finally *commits* (i.e., ends successfully) or *aborts* (i.e., is canceled prior to successful termination). If t_i appears in the schedule, a commit [abort] is indicated by c_i [a_i], respectively. In other words, a history comprises an indication of how each of its transactions terminates. Following the ACID principle, committed transactions have been run completely, preserve the consistency of the database, have not seen dirty data, and are durable. On the other hand, aborted transactions have no impact on the database, and the system must ensure they are undone completely. A history is *serial* if for any two transactions t_i, t_j appearing in it either all of t_i precedes all of t_j or vice versa.

Note that histories are rare in practice, because transaction processing and execution normally occur highly dynamically, that is, transactions come and go unpredictably. To capture this dynamic situation, we need the following: A *schedule* is a prefix of a history. We are interested mostly in schedules in what follows because this is what an execution protocol has to create dynamically. On the other hand, schedule correctness refers back to histories, because serial schedules by definition are complete, and serializability means equivalence to seriality.

Our notion of correctness is based on conflicts between transactions that access common data objects. Two steps from distinct transactions are *in conflict* in a given schedule, if they operate on the same database object and at least one of them is a write operation.

Now we are ready to write schedules for the lost update and the inconsistent read problem, called L and P, respectively:

$$L = r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2 \tag{4}$$

$$P = r_1(x)r_1(y)r_2(z)w_2(z)r_2(x)w_2(x)c_2r_1(z)c_1 \tag{5}$$

The following is an example of a history for four transactions, in which t_0, t_2 and t_3 are committed and t_1 is aborted:

$$S = w_0(x)r_1(x)w_0(z)r_2(x)w_0(y)c_0r_3(z)w_3(z)w_2(y)c_2w_1(x)w_3(y)a_1c_3 \tag{6}$$

If T is a subset of the set of all transactions in a schedule s, the *projection* of s onto T is obtained by erasing from s all steps from transactions not in T. For example, the projection of schedule S [Eq. (8)] onto its committed transactions is the schedule

$$w_0(x)w_0(z)r_2(x)w_0(y)c_0r_3(z)w_3(z)w_2(y)c_2w_3(y)c_3$$

A serial schedule for the original four transactions is given by $S' = t_0t_2t_1t_3$. Note that generally there always exist $n!$ serial schedules for n transactions.

An important observation at this point is that transactions, schedules, and histories are *purely syntactic* objects, which describe only the sequencing of data accesses performed by a database program, how these are interleaved, and what eventually happens to each transaction. A common assumption in traditional concurrency control theory is that the *semantics* of transactions are not known. On the other hand, a *pseudosemantics* can be associated with a given transaction as follows. It is assumed that the (new) value of an object x written by some step $w(x)$ of a given transaction t depends on *all* values of objects previously read by t. The value of x read by some step $r(x)$ of t depends on the last $w(x)$ that occurred before $r(x)$ in t or on the "initial" value of x if no such $w(x)$ exists. This can be extended to schedules and histories in the obvious way, with the additional condition that transactions aborted in a schedule or history are ignored. For example, in the schedule S [Eq. (8)], t_1 reads x and z from t_0, but the value produced by $w_1(x)$ does not appear in the database.

As mentioned already, the distinction between a history and a schedule captures a dynamic situation in which transactions arrive at a scheduling device step-by-step, and the device has to decide on the spot whether or not to execute a given step. For various reasons it might happen that at some point the device discovers that a transaction cannot be completed successfully, so that it has to output an abort operation for this transaction. We do not consider how aborts (and also commits) are processed internally. To this end, we refer the reader to Weikum and Vossen (14) as well as to Gray and Reuter (15). Next we turn to the issue of schedule *correctness*.

Conflict Serializability

Because serializability relates to serial executions as a correctness notion, next we introduce a corresponding notion of equivalence for schedules. We mention that essentially every notion of serializability described in the literature is obtained in this way, including *final-state* and *view serializability;* see Papadimitriou (16). The notion we are about to introduce here enjoys a number of interesting properties: Unlike final-state or view serializability, which have an NP-complete decision problem, it can be tested in time linear in the number of given transactions, and it allows designing simple protocols that can be implemented economically.

The *conflict relation* conf(s) of a schedule s consists of all pairs of steps $[a, b]$ from distinct, unaborted transactions which are in conflict in s and for which a occurs before b. If s and s' are two schedules for the same set of transactions, s and s' are *conflict equivalent*, denoted $s \approx_c s'$, if conf(s) = conf(s'). Finally, a history s is *conflict serializable* if there exists a serial schedule s' for the same set of transactions such that $s \approx_c s'$. Let CSR denote the class of all (complete

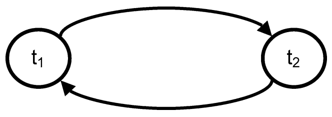

Figure 5. Cyclic conflict graph for schedules L and P.

and) conflict serializable histories.

Let us investigate the sample schedules, shown earlier, in the light of this correctness notion: For schedule L shown in Eq. (6),

$$\text{conf}(L) = \{[r_1(x), w_2(x)], [r_2(x), w_1(x)], [w_1(x), w_2(x)]\} \quad (7)$$

Now the only possible serial schedules are t_1t_2, whose conflict relation would avoid the second pair of conflicting operations, and t_2t_1, whose conflict relation would comprise only the second pair. Thus, schedule L cannot be conflict serializable. Next, for schedule P shown in Eq. (7),

$$\text{conf}(P) = \{[r_1(x), w_2(x)], [w_2(z), r_1(z)]\} \quad (8)$$

which again cannot be obtained from t_1t_2 or from t_2t_1. Thus, $P \notin$ CSR. Finally, for schedule S shown in Eq. (8),

$$\text{conf}(S) = con\ f(t_0t_2t_3) \quad (9)$$

Because the latter schedule, which ignores the aborted t_1, is serial, $S \notin$ CSR.

Thus we can state two important facts. First, the situations of lost update and inconsistent reads, identified above as unwanted, are "filtered out" by the correctness criterion of conflict serializability. Second, as also seen from these examples, conflict equivalence for two given schedules is easy to test: Compute their conflict relations, and check them for equality. Testing conflict serializability for a given schedule is, however, more complicated because in principle we would have to compute the conflict relation for *every* serial schedule over the given transactions and compare that to the conflict relation of the schedule in question. Fortunately, there is an easy test to determine whether a history S is in CSR: First, construct the *conflict graph* $G(S) = (V, E)$ of S, whose set V of nodes consists of those transactions from S which are not aborted and which contains an edge of the form (t_i, t_j) in E if some step from t_i is in conflict with a subsequent step from t_j. Second, test this graph for acyclicity. Then, it can be shown that for every schedule S, $S \in$ CSR iff $G(S)$ is acyclic. Because testing a directed graph for acyclicity is polynomial in the number of nodes, therefore membership of a schedule in class CSR is computationally easy to test.

To complete our example, Fig. 5shows the conflict graph of both schedules L and P, which contains two transactions involved in a cyclic conflict. Figure 6shows the conflict graph of schedule Sabove, which is acyclic.

Finally we mention that *correctness* of schedules generally involves a second issue, that of fault tolerance or resiliency against failures. To this end, notions like *recoverability* or *strictness* have been proposed, see Bernstein et al. (17) or Weikum and Vossen (14), and synchronization procedures, to be discussed next, normally have to ensure that their output is both serializable and recoverable.

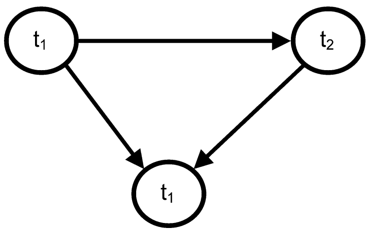

Figure 6. Acyclic conflict graph for schedule S.

Concurrency Control Protocols

Our next interest is in *protocols*, that is, algorithmic approaches for creating (correct) schedules dynamically. In essence, such protocols take several transactions (or an *arbitrary* schedule for them) as input and generate a *correct* schedule from these as output, as indicated in Fig. 7. Strictly speaking, only output schedules contain abort or commit operations for their transactions, but input schedules never do. This is the reason why we distinguish *schedules* from *histories:* A history describes the output produced by a scheduler and hence the complete sequence of operations that has been executed over time, whereas in a schedule only the data operations (reads and writes) matter.

Concurrency control protocols developed for system implementation can generally be divided into two major classes:

1. *Pessimistic protocols* are based on the assumption that conflicts between concurrent transactions are likely, so that provisions need to be taken to handle them. Known protocols in this class include *two-phase locking, time-stamp ordering* and *serialization graph testing*.
2. *Optimistic protocols* are based on the opposite assumption that conflicts are rare. As a consequence, it is possible to schedule operations vastly arbitrarily and just make sure from time to time that the schedule generated is correct. Protocols based on this idea are known as *certifiers* or as *validation protocols*.

Detailed descriptions of the protocols just mentioned and of many of their variations can be found, for example, in Weikum and Vossen (14). Here we sketch the idea behind locking schedulers only because these are most widely used in commercial systems. The basic idea underlying any locking scheduler is to require that accesses to database objects by distinct transactions are executed *mutually exclusively*. In particular, a transaction cannot modify (write) an object as long as another transaction is still operating on it (reading or writing it). Notice that this corresponds to the notion of conflict between data operations, as introduced earlier. The central paradigm to implement this idea is the use of *locks*, which are set by the scheduler on behalf of a transaction before the latter reads or writes and which are removed after the access has been executed. Two types of lock operations suffice for read and write operations: If a transaction wants to read [write] an object, it requests a read lock [write lock], respectively. A read lock indicates

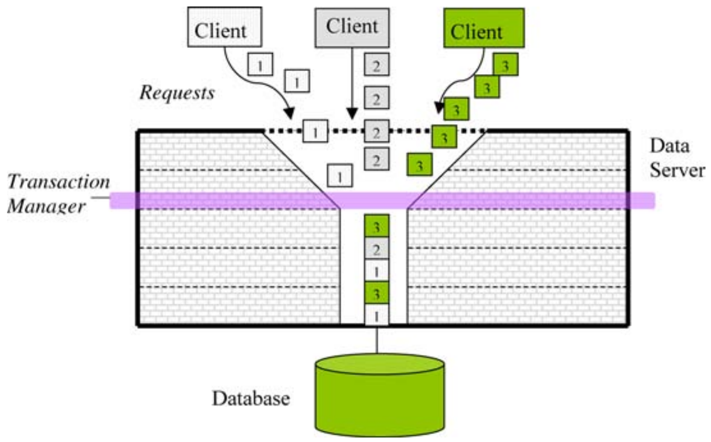

Figure 7. Scheduling situation.

to other transactions, which want to write, that the object in question is currently available for reading only. A write lock indicates that the object is currently not available. Two locks from distinct transactions are in *conflict* if both refer to the same object and (at least) one of them is exclusive. In this case, only one of the requests can be granted because the requests are *incompatible*. A scheduler operates according to a *locking protocol* if in every schedule generated by it all simultaneously held locks are compatible. It operates according to a *two-phase protocol* (*2PL*) if additionally no transaction sets a new lock after it has released one. The most popular variant of 2PL is to hold all locks of a transaction until this transaction terminates (*strict* 2PL). A straightforward motivation for its use is that a time at which a scheduler can be sure that a transaction will not request any further locks is the end of the transaction.

It is easy to verify that 2PL is *correct*, that is, that every schedule generated by 2PL is conflict serializable. In addition, its strict variant even generates strict schedules. Although this protocol is easy to implement, outperforms other protocols, and is easily generalized to distributed systems, it also has its shortcomings. For example, it is not free of *deadlocks*, so that additional means must be taken to discover and resolve these. We refer the reader to Gray and Reuter (15) for implementation issues.

We indicate the way 2PL works by way of our sample schedules. First, let us consider input schedule L shown in Eq. (6), whose sequence of data operations is $r_1(x)r_2(x)w_1(x)w_2(x)$: 2PL creates the output $rl_1(x)r_1(x)rl_2(x)r_2(x)$, where w_l stands for “write lock,” r_l for “read lock,” and u_l for “(read or write) unlock.” Now the scheduler must stop, because $w_1(x)$ needs a write lock on x, incompatible with the existing read lock from t_2, and $w_2(x)$ needs another write lock on x, incompatible with the existing lock from t_1. Thus, we observe a deadlock situation which can broken only by aborting one of the two transactions and restarting it at some later time.

Next, let us look at input schedule P from Eq. (7). Now a 2PL protocol starts out as $r_{l1}(x)r_1(x)rl_1(y)r_1(y)wl_2(z)r_2(z)w_2(z)$. At that point, t_2 requests a write lock on x for doing $r_2(x)w_2(x)$, which would not be granted. Moreover, $r_1(z)$ requires a read lock on z, incompatible with the existing z lock from t_2. So again, one of the transactions must be aborted, and this particular schedule is avoided.

Finally, let us consider schedule S from Eq. (8), for which 2PL could generate the following output:

$$wl_0(x)w_0(x)wl_0(z)w_0(z)wl_0(y)w_0(y)ul_0(x,z,y,)c_0rl_1(x)r_1(x)$$
$$rl_1(z)r_1(z)rl_2(x)r_2(x)wl_2(y)w_2(y)ul_2(x,y)c_2rl_3(z)r_3(z)wl_1(x)$$
$$w_1(x)a_1ul_1(z,x)wl_3(z)w_3(z)wl_3(y)w_3(y)ul_3(z,y)c_3$$

Even in this case, the order of operations in the schedule (not within individual transactions!) has been modified slightly, but only relative to allowed commutations of operations. Clearly, there could be other ways a 2PL scheduler handles this input, in particular if the output is additionally required to be strict.

In system implementations, a concurrency control protocol such as 2PL is commonly complemented with an appropriate *recovery protocol* that takes care of transaction aborts (by *undoing* or *redoing* the respective operations), by keeping a *log* in which a record of activities is kept, and by handling system restarts after crashes. Logs typically keep track of each and every operation done to a page (through sequence numbers) as well as done on behalf of a transaction; they are processed during a recovery operation in order to bring the database back into a stable state. To this end, it is crucial that a log is kept in a safe place, i.e., on disk, or that it is at least copied to disk in regular intervals. Various recovery techniques are described by Weikum and Vossen (14) or Gray and Reuter (15).

DISTRIBUTED DATABASE SERVERS

As recognized a long time ago, organizations are frequently decentralized and hence require databases at multiple sites. For example, a nationwide bank has branches all over its country and wants to keep customer data local, so that the data is available where it is actually used. In addition, decentralization increases the availability of a system in the presence of failures. As a result, *distributed database systems* began to emerge during the 1980s, and nowadays all major database system vendors are commercializing distributed technology. In brief, a *distributed database* is a collection of multiple, logically interrelated databases distributed over a computer network, as illustrated in Fig. 8. A *distributed database management system* is the software

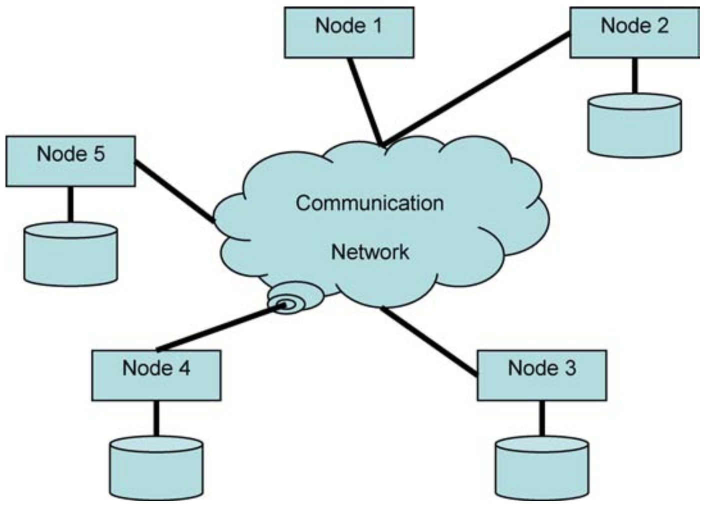

Figure 8. Distributed Database Environment.

that permits the management of a distributed database and makes the data distribution transparent to its users. The latter means that a distributed system should look to its users as if it were nondistributed. This objective has a number of consequences and creates many new challenges for implementors. Among the core requirements for a distributed database system are the following: Each site in the system should be locally autonomous and should not depend on a central master site. Users do not need to know at which site data is physically stored (*location transparency*), how data sets are internally fragmented (*fragmentation transparency*) or replicated at distinct sites (*replication transparency*), or how queries or transactions that access data at multiple sites are executed (*processing transparency*). Özsu and Valuriez (18) are a good source on the subject.

We will look at some implications that database distribution has on the underlying processing concepts; further information can be found in Öszu and Valduriez (18). In particular, we again consider query processing at multiple sites and transaction processing in distributed databases.

Query Processing in Distributed Databases

Query processing is somewhat trickier in a distributed database, because now it may be much more complicated to determine an efficient evaluation strategy. Consider our sample bank database once more. It is easy to imagine that the bank running this database, having branches in various city districts, wants to keep data local. Therefore, relation B would be *horizontally fragmented* into the following three relations:

B_1	BID	AID	CityDistrict
	1	110	downtown
	1	120	Downtown
	1	130	Downtown

B_2	BID	AID	CityDistrict
	2	220	beach area
	2	240	beach area

B_3	BID	AID	CityDistrict
	3	310	lower east
	3	320	lower east
	3	330	lower east

Because now each branch keeps its account information separate, it would even suffice to store a projection onto attributes BID and CityDistrict at each site. Correspondingly, relation A could be distributed to the same sites as the following fragments:

A_1	AID	CustName	Balance
	110	Smith	1,324
	120	Jones	256
	130	Maier	22,345

A_2	AID	CustName	Balance
	220	Paul	2,550
	240	Kline	86

A_3	AID	CustName	Balance
	310	Adams	1,360
	320	Kelly	23,022
	330	Barbara	345

Now consider again the query which we have previously optimized in Eq. (5). Let us assume that the relations and their fragments are distributed as follows: Site 1 holds A_1, B_1, and C, Site 2 keeps A_2 and B_2, and Site 3 maintains A_3 and B_3. Because the query asks for a selection on C (customers born before 1950) and one on B_1 (the downtown branch), the first join should be executed at Site 1. For the second join, however, there are various options, including the following:

1. Ship the result of the first join to Site 2 and the result computed there to Site 3;
2. ship A_2 and A_3 to Site 1, and compute the final result at this site;
3. ship A_2 to Site 3, compute a union with A_3, and join with the intermediate result from Site 1.

This situation is typical for query processing in distributed databases, where often a variety of options exist for shipping data from one site to another to speed up processing. Thus, query optimization in distributed databases is made more complicated by the fact that now even transfer costs have to be taken into account. One technique developed in this context is the use of *semijoins.* Referring back to the terminology introduced for database relations, let $r \in \text{Rel}(X)$, $s \in \text{Rel}(Y)$. The *semijoin* of r with s is defined as $r| \times s := \pi_x(r \bowtie s)$. The following rules are easily verified for this operation:

1. $r \ltimes s = r \bowtie \pi_{X \cap Y}(s)$
2. $r \bowtie s = (r \ltimes s) \bowtie s$

An immediate exploitation of these rules in a computation of a join of relations r and s stored at distinct Sites 1 and 2, respectively, is as follows:

1. compute $s' := \pi_{X \cap Y}(s)$ in Site 2;
2. ship s' to r, and compute $r' := r \bowtie s'$ in Site 1;
3. ship r' to s, and compute $s'' := r' \bowtie s$ in Site 2;

The following shows why this strategy is correct:

$$s'' = r' \bowtie s = (r \bowtie s') \bowtie s = (r \bowtie \pi_{X \cap Y}(s)) \bowtie s = (r| \times s) \bowtie s = r \bowtie s$$

Another way of avoiding data transfers between sites is to keep data *replicated* at various sites, that is, to keep copies of certain data to avoid unnecessary transfers. Although this appears attractive at first glance, it bears the additional complication of keeping the replicas identical, that is, to propagate updates made to one copy to all others consistently, even in the presence of network failures, a problem that we will briefly look at below in connection with data sharing systems.

Transaction Processing in Distributed Databases

From a logical point of view, transaction processing in distributed databases is a vast generalization of that in centralized databases, which, depending on the protocol used, introduces only minor additional complications. For example, if all sites participating in a distributed database run the same database software (i.e., if the system is *homogeneous*), then each site can run the strict 2PL protocol independently. Thus, transactions can access data at the respective sites and acquire and release locks as needed, as long as the 2PL property is locally maintained. The only critical situation arises when a transaction finishes, because it has to commit in *every* site where it was active or it has to abort, but not a combination of both. To this end, commercial systems apply the *Two-Phase Commit* (*2PC*) protocol, which guarantees consistent termination for distributed environments. Other problems specific to transactions over distributed data may involve global deadlock detection, difficult in the absence of a central monitor which would always have complete information, or the computation of global clock values needed for intersite synchronization purposes.

For efficiency, central monitors for whatever purposes are helpful in distributed scenarios, but at the same time they are undesirable from a logical point of view because their presence contradicts the requirement of local autonomy. In this situation, a compromise can be seen in *client-server* architectures as commonly used in database systems. In such an architecture, some sites act as clients which send processing requests to other sites known to be able to service them and to return replies as a result. A request typically is a query or a transaction's read or write operation. A general client-server database system model is described by Weikum and Vossen (14); see also Ramakrishnan and Gehrke (10) or Silberschatz et al. (5).

We conclude by briefly looking at a prominent form of distributed server today, which exhibits a considerable amount of *parallelism*. Recall that some form of parallelism is already found in standard transaction processing, because multiple transactions are frequently run concurrently. Another form of (true) parallelism, also transparent to applications, occurs in *data-sharing clusters*. Here, data is distributed over the available disks, and queries as well as transactions are decomposed so that they can be executed at multiple processors simultaneously, and the entire system comprises servers for data-intensive applications with very high throughput and very high availability guarantees. A cluster is a small number, typically between 2 and 8, of machines, each of which runs its own copy of the operating system, database system, etc., and could be a shared-memory multiprocessor. The key characteristic is that each server has its own "private" memory; there is no shared memory across servers. When a server fails, the other servers of the cluster continue operating and may take over the load of the failed server ("fail-over").

In most implementations, a transaction is executed entirely on a single server. When a transaction accesses a page, this page is brought into the memory of the corresponding server, either from the shared disks on which the data resides permanently or from the memory of another server which happened to have that page in its cache. For consistency reasons, a *cache coherency protocol* needs to be employed, so that if a page resides in more than one cache for an extended time period and is modified in one of these caches, the other servers are notified. The main invariant that each page-oriented coherency control protocol needs to ensure that (1) multiple caches can hold up-to-date versions of a page simultaneously as long as the page is only read, and (2) once a page has been modified in one of the caches, this cache is the only one that is allowed to hold a copy of the page. A protocol guaranteeing this is the *callback locking protocol*insisting on "calling back" pages for update, whose details can be found in Weikum and Vossen (14).

The data-sharing cluster architecture is illustrated in Fig. 9. The headers of the various pages will in reality contain log sequence numbers to indicate when they were last modified. Note that pages p and q reside in two different caches, and the sequence numbers of the two copies must be identical by the above coherency requirement. The cached version of page p will here be more recent than the one in the stable database on disk, as indicated by their sequence numbers.

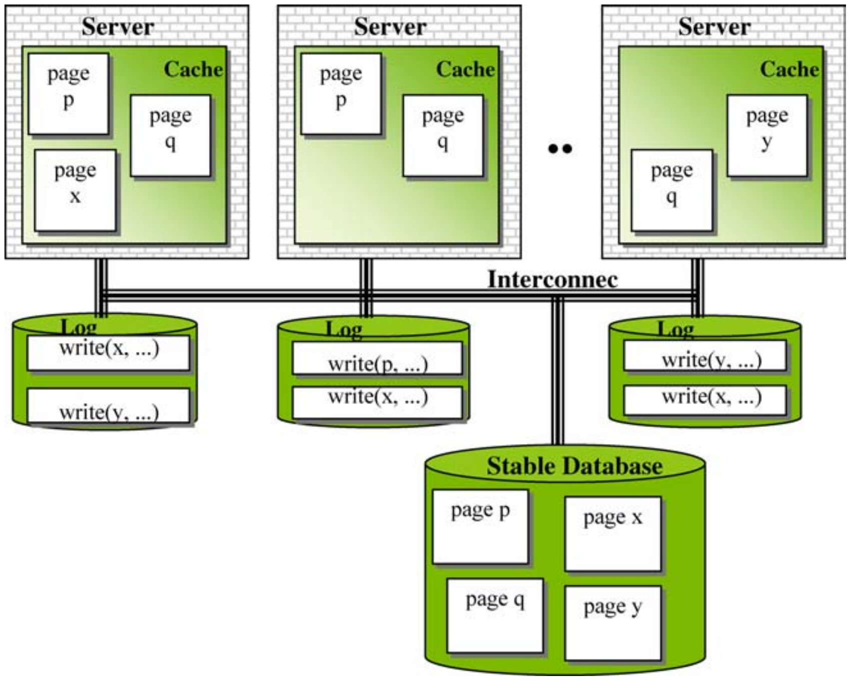

Figure 9. Data-Sharing Cluster.

FUTURE CHALLENGES

We conclude our brief summary of database processing by mentioning selected current and future challenges. Essentially, these fall into two categories: database administration and XML processing.

The database system internals we have sketched above result in a number of "knobs" or parameters which a database administrator can and typically must monitor and influence. For example, the query optimizer of a database system can be configured so that emphasis is given to space or time optimization; it can be switched on or off depending on the query application at hand (e.g., a query issued by a Web server several thousand times a day might be optimized "by hand" instead of leaving this to an optimizer program). Similarly, a transaction processor can be configured w.r.t. to the number of transactions it may run currently or the number of locks that can be held simultaneously. Very often, some form of *feedback loop control* (14) is employed, which basically monitors some system parameters(s) on a continuous basis and adapts the system settings appropriately. If, say, the concurrency control system spends too much time on lock management and cannot complete any further transactions, some need to be aborted (and no new ones accepted) in order to bring down the amount of parallelism, until the system has recovered enough to accept additional transactions. Activities like these fall under the general category of *database tuning* (11). The point is that many of these tasks can be automated in such a way that the system can take care of them *without* the intervention or help of a database administrator. As a consequence, one of the goals for DBMS system development nowadays is to make systems *self-administering* and to equip them with *auto-tuning* facilities; see Chaudhuri and Weikum (19) for more on this topic. The importance of this cannot be underestimated, given the fact that database systems are more and more embedded into other systems and then have to operate without the supervision of an administrator.

While the relational model of data has dominated the world of database applications for several decades now, it was discovered already in the early 1980s that the model is not expressive enough for a number of applications. For example, a complex artifact such as a car could be broken down into a relational representation, but it would be way more appropriate to use a database model at hand that can support some form of "complex objects". This has led to a number of extensions of the relational model and ultimately to the "marriage" of relational databases and the programming paradigm of object-orientation into what is now known as *object-relational* systems (20). Moreover, a wide-spread database application nowadays is *integration*, i.e., the task of bringing together and unifying data from a variety of sources into a consistent data collection. To this end, XML, the *Extensible Markup Language*, has found its way into database systems. As a brief example, Fig 10shows the representation of book information (with ISBN, author, title publisher etc.) in an arbitrarily chosen XML format. Notice that the partial document shown in Fig. 10is structured by so-called *tags* (e.g., "BOOK", "ISBN"), that tags observe a strict nesting (e.g., LASTNAME inside PERSON inside AUTHOR), and that the document in total is ordered, i.e., moving elements around would formally result in a new document.

XML is nowadays supported by every major database management system, either natively or as an extension to the features already available in the respective system (e.g., as "Extender" or "Blade"). Clearly, supporting XML requires a number of modifications to a previously relational system, in particular when it comes to database processing. Indeed, relational algebra is obviously not directly applicable anymore, so new features are needed for specifying

```
<BOOK category="fiction" language="de">
        <ISBN>342333052X</ISBN>
        <AUTHOR>
                <PERSON>
                        <LASTNAME>Singh</LASTNAME>
                        <FIRSTNAME>Simon</FIRSTNAME>
                </PERSON>
        </AUTHOR>
        <TITLE>Fermat‘s Last Theorem</TITLE>
        <PUBLISHER>DTV</PUBLISHER>
        <LOCATION>Munich</LOCATION>
        <EDITION>1</EDITION>
        <YEAR>2000</YEAR>
</BOOK>
```

Figure 10. A sample XML document.

and executing queries. To the end, it is important to observe that XML documents can be perceived as *trees*, so that query capabilities can be designed around the notion of a tree. This has led to the development of XPath as a language for navigating through XML document trees and for selecting nodes from such a tree. XPath poses a number of challenges to query processing, see Gottlob et al. (21) for an introduction. On the other hand, XPath is one of the foundations of XQuery, the next-generation database query language that has been designed for XML database systems; see Melton and Buxton (22) for the state-of-the-art in this respect.

BIBLIOGRAPHY

1. E. F. Codd A relational model of data for large shared data banks, *Commun. ACM*, **13**: 377–387, 1970.
2. D. Maier *The Theory of Relational Databases*, Rockville, MD: Computer Science Press, 1983.
3. J. D. Ullman *Principles of Database and Knowledge-Base Systems*, Rockville, MD: Computer Science Press,1988/9, Vols. I and II.
4. R. Elmasri S. B. Navathe *Fundamentals of Database Systems*, 5th ed., Boston, MA: Pearson Addison-Wesley, 2006.
5. A. Silberschatz, H.F. Korth, S. Sudarshan *Database System Concepts*, 5th ed., New York: McGraw-Hill, 2006
6. S. B. Yao Optimization of query evaluation algorithms, *ACM Trans. Database Syst.*, **4**: 133–155, 1979.
7. J. C. FreytagD. MaierG. Vossen (eds.) *Query Processing for Advanced Database Systems*, San Francisco: Morgan Kaufmann, 1994.
8. G. Graefe Query evaluation techniques for large databases, *ACM Computing Surveys*, **25**: 73–170, 1993.
9. P. E. O’Neil and P.E. O’Neil *Database: Principles, Programming, Performance*, 2nd ed., San Francisco: Morgan Kaufmann, 2000.
10. R. Ramakrishnan and J. Gehrke *Database Management Systems*, 3rd ed. New York: WCB/McGraw-Hill, 2003.
11. D. Shasha and Ph. Bonnet *Database Tuning – Principles, Experiments, and Troubleshooting Techniques*, San Francisco: Morgan Kaufmann Publishers, 2003.
12. J. Gray Notes on data base operating systems,inR. Bayer, M. R. Graham, andG. Seegmüller, (eds.), *Operating Systems—An Advanced Course*, Berlin: Springer Verlag, 1978, LNCS 60,pp. 393–481.
13. J. Gray The transaction concept: Virtues and limitations,in *Proc. 7th Int. Conf. Very Large Data Bases*, San Francisco, CA: Morgan Kaufmann, 1981, pp. 144–154.
14. G. Weikum and G. Vossen *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*; San Francisco: Morgan-Kaufmann Publishers; 2002
15. J. Gray A. Reuter *Transaction Processing: Concepts and Techniques*, San Francisco: Morgan Kaufmann, 1993.
16. C. H. Papadimitriou *The Theory of Database Concurrency Control*, Rockville, MD: Computer Science Press, 1986.
17. P. A. Bernstein V. Hadzilacos N. Goodman *Concurrency Control and Recovery in Database Systems*, Reading, MA: Addison-Wesley, 1987.
18. M. T. Özsu P. Valduriez *Principles of Distributed Database Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1991.
19. S. Chaudhuri, G. Weikum: Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System, in *Proc. 26th Int. Conf. Very Large Data Bases*, San Francsico, Morgan Kaufmann Publishers, 2000, pp. 1–10.
20. S. W. Dietrich, S. D. Urban: *An Advanced Course in Database Systems: Beyond Relational Databases*, Englewood Cliffs, NJ: Prentice-Hall, 2005.
21. G. Gottlob, Ch. Koch, R. Pichler: XPath Query Evaluation: Improving Time and Space Efficiency,in *Proc. 19th Int. Conf. Data Engineering*, IEEE Computer Society, 2003, pp. 379–390.
22. J. Melton, St. Buxton, *Querying XML – Xquery, XPath, and SQL / XML in Context*, San Francisco: Morgan Kaufmann Publishers, 2006

GOTTFRIED VOSSEN
European Research Center for Information Systems (ERCIS) University of Münster, Germany