Databases

Alvaro A.A. Fernandes, Norman W. Paton
Department of Computer Science
University of Manchester

April 27, 2000

Contents

Glossary

Application A topic or subject for which information systems are needed, e.g., genomics.

Conceptual model A data model that describes application concepts at an abstract level. A conceptual model for an application may be amenable to implementation using different database management systems.

Concurrency control Mechanisms that ensure that each individual accessing the database can interact with the database as if they are the only user of the database, with guarantees as to the behavior of the system when many users seek to read from or write to the same data item at the same time.

Database A collection of data managed by a *database management system.*

Database management system A collection of services that together give comprehensive support to applications requiring storage of large amounts of data that is to be shared by many users

DBMS *see* **database management system**

Data model A collection of data types which are made available to application developers by a DBMS.

Data type A set of instances of an application concept with an associated set of legal operations in which instances of the concept can participate. Some data types are *primitive* (e.g., integer, string) insofar as they are supported by the DBMS directly (and, hence, are not application specific). Some are *application specific* (e.g., gene). Some data types are *atomic* (e.g., integer) and some are *complex* (e.g., address, comprising street name, state, etc.). Some data types are *scalar* (e.g., integer) and some are *bulk*, or *collection* (e.g., sets, lists).

Persistence The long term storage of an item of data in the form supported by a data model.

Referential integrity The requirement that when a reference is made to a data item, that data item must exist.

Secondary storage management The ability to store data on magnetic disks or other long-term storage devices, so that data outlives the program run in which it was created.

Security Mechanisms that guarantee that no users see or modify information that they are not entitled to see or modify.

A **database** is a collection of data managed by a *database management system* (DBMS). A DBMS provides facilities for describing the data that is to be stored, and is engineered to support the long term, reliable storage of large amounts of data [4]. DBMSs also provide query language and/or programming language interfaces for retrieving and manipulating database data. Many organizations are dependent in a significant way upon the reliability and efficiency of the DBMSs they deploy.

I Database Management Systems

A **database management system** (DBMS) provides a collection of services that together give comprehensive support to applications requiring storage of large amounts of data that is to be shared by many users. Among the most important services provided are:

Secondary storage management: The ability to store data on magnetic disks or other long-term storage devices, so that data outlives the program run in which it was created. Sophisticated secondary storage management systems are required to support effective storage and access to large amounts of data. Storage managers typically include facilities for providing rapid access to specific data items using *indexes*, minimize the number of distinct accesses required to the secondary store by *clustering* related data items together, and seek wherever possible to make effective use of *buffers* that retain recently accessed data items in the memory of the computer on which the database system or the application is running.

Concurrency control: Large databases are valuable to the organizations that hold them, and often have to be accessed or updated by multiple users at the same time. Uncontrolled updates to a shared store can lead to the store being corrupted as a result of programs interfering with each other in unanticipated ways. As a result, DBMSs incorporate mechanisms that ensure that each individual accessing the database can interact with the database as if they are the only user of the database, with guarantees as to the behavior of the system when many users seek to read from or write to the same data item at the same time.

Recovery: Many database applications are active 24 hours a day, often with many different programs making use of the database at the same time, where these programs are often running on different computers. As a result, it is inevitable that programs accessing a database will fail while using the database, and even that the computer running the DBMS can go down while it is in use. The DBMS must be as resilient as possible in the face of software or hardware failures, with a view to minimizing the chances of a database being corrupted or information lost. As a result, DBMSs include recovery facilities that often involve some temporary *replication* of data so that enough information is available to allow automatic recovery from different kinds of failure.

Security: Most large database systems contain data that should not be visible to or updateable by at least some of the users of the database. As a result, comprehensive mechanisms are required to guarantee that no users see information that they are not entitled to see, and more important still, that the users who are able to modify the contents of the database are carefully controlled – staff should not be able to modify their own salaries, etc. As a result, database systems provide mechanisms that allow the *administrator* of a database to control who can do what with the database.

DBMSs can thus be seen as having many responsibilities. Like a bank, it is good if a DBMS provides a wide range of facilities in an efficient way. However, in the same way as it is of overriding importance that a bank doesn't lose your money, it is crucial that a DBMS must keep track of the data that is entrusted to it, minimizing errors or failures. Thus the provision of a database management facility is essentially a serious matter, and substantial database failures would be very damaging to many organizations. However, the details of how the above services are supported is quite technical, and the remainder of this Chapter focuses on the facilities provided to users for describing, accessing and modifying database data. Details on how databases are implemented can be found in [9, 8].

II Data Models

In the context of database technology, by **data model** is meant a collection of data types which are made available to application developers by a DBMS. By **application** is meant a topic or subject for which information systems are needed, e.g., genomics. The purpose of a data model is to provide the formal framework for modeling application requirements regarding the persistent storage, and later retrieval and maintenance, of data about the entities in the conceptual model of the application and the relationships in which entities participate.

Each data type made available by the data model becomes a building block with which developers model application requirements for persistent data. The permissible operations associated with each data type become the building blocks with which developers model application requirements for interactions with persistent data. Data models, therefore, differ primarily in the collection of data types that they make available. The differences can relate to structural or behavioral characteristics. Structural characteristics determine what states the database may be in (or, equivalently, are legal under the data model). Behavioral characteriztics determine how database states can be scrutinized and what transitions between database states are possible under the data model.

A DBMS is said to implement a data model in the sense that:

1. it ensures that every state of every database managed by it only contains instances a data type that is well-defined under the implemented data model, and

2. it ensures that every retrieval of data and every state transition in every database managed by it is the result of applying an operation that is (and only involves types that are) well-defined under the implemented data model.

¿From the point of view of application developers then, the different structural or behavioral characteriztics associated with different data models give rise to the pragmatic requirement of ensuring that the data model chosen to model application requirements for persistent data do not unreasonably distort the conceptual view of the application that occurs most naturally to users. The main data models in widespread use at the time of writing meet this pragmatic requirement in different degrees for different kinds of application.

II.A The Relational Data Model

The relational data model makes available one single data type, referred to as a *relation*. Informally, a relation can be thought of as a table, with each row corresponding to an instance of the application concept modeled by that relation and each column corresponding to the value of a property that describes the instance. Rows are referred to as *tuples* and column headers as *attributes*. More formal definitions now follow.

Let a **domain** be a set of atomic values, where a value is said to be *atomic* if no further structure need be discerned in it. For example, integer values are atomic, and so, quite often, are strings. In practice, domains are specified by choosing one data type, from the range of primitive data types available in the software and hardware environment, whose values become elements of the domain being specified.

A **relation schema** $R(A_1, \ldots, A_n)$ describes a relation R of **degree** n by enumerating the n-*attributes* $A_1, \ldots, A_n$ that characterize R. An **attribute** names a role played by some domain D in describing the entity modeled by R. The domain D associated with an attribute A is called the **domain of** A and is denoted by $dom(A) = D$. A **relational database schema** is a set of relation schemas plus the following **relational integrity constraints**:

1. *Domain Constraint*: Every value of every attribute is atomic.

2. *Key Constraint*: Every relation has a designated attribute (or a concatenation thereof) which acts as the **primary key** for the relation in the sense that the value of its primary key identifies the tuple uniquely.

3. *Entity Integrity Constraint*: No primary key value can be the distinguished `NULL` value.

4. *Referential Integrity Constraint*: If a relation has a designated attribute (or a concatenation thereof) which acts as a **foreign key** with which the relation refers to another, then the value of the foreign key in the referring relation must be a primary key value in the referred relation.

For example, a very simple relational database storing nucleotide sequences might draw on the primitive data type `string` to specify all its domains. Two relation names might be `DNA_sequence` and `organism`. Their schemas can be represented as in Figure 1. In this simple example, a `DNA_sequence` has an identifying attribute `sequence_id`, one or more `accession_nos` by which the DNA_sequence is identified in other databases, the identifying attribute `protein_id` which the DNA_sequence codes for and the `organism_ids` of the organisms where the DNA_sequence has been identified. An `organism` has an identifying attribute `organism_id`, perhaps a `common_name` and the `up_node` of the organism in the adopted taxonomy.

DNA_sequence			
sequence_id	*accession_no*	*protein_id*	*organism_id*

organism		
organism_id	*common_name*	*up_node*

Figure 1: Relation Schemas for `DNA_sequence` and `organism`

A **relation instance** $r(R)$ of a relation schema $R(A_1, \ldots, A_n)$ is a subset of the Cartesian product of the domains of the attributes that characterize R. Thus, a relation instance of degree n is a set, each element of which is an n-tuple of the form $\langle v_1, \ldots, v_n \rangle$ such that either $v_i \in dom(A_i)$, $1 \leq i \leq n$, or else v_i is a distinguished `NULL` value. A **relational database state** is a set of relation instances. Legal relation instances for `DNA_sequence` and `organism` from Figure 1 might be as depicted in Figure 2.

DNA_sequence			
sequence_id	accession_no	protein_id	organism_id
TRBG361	X56734	AA40058	Trif. repens
BNBGL	X82577	CAA57913	Bras. napus
...	...	...	...

organism		
organism_id	common_name	up_node
Bras. napus	rape	Brassicaceae
Trif. repens	white clover	Papilionoideae
...	...	...

Figure 2: Relation Instances under the Schemas in Figure 1

Notice, in Figure 2, how each tuple in `DNA_sequence` is an element of the Cartesian product:

$$dom(\texttt{sequence\_id}) \times dom(\texttt{accession\_no}) \times dom(\texttt{protein\_id}) \times dom(\texttt{organism\_id})$$

and correspondingly in the case of `organism`. Notice also, in Figures 1 and 2, that all the relational integrity constraints are satisfied, as follows. The domain constraint is satisfied due to the fact that the domain of every attribute is the primitive data type `string`, whose instances have been assumed to be atomic. The designated primary keys for `DNA_sequence` and `organism` are respectively `sequence_id` and `organism_id`, and since they are intended as unique identifiers of the entities modeled, the key constraint is satisfied. The entity integrity constraint is satisfied assuming that tuples that are not shown also do not contain `NULL` as values for the primary keys. There is a designated foreign key, `organism_id`, from `DNA_sequence` to `organism`, and since the values of `organism_id` appear as primary key values in `organism`, the referential integrity constraint is satisfied.

The relational data model has several distinctive features as a consequence of its simple mathematical characterization:

- Since a relation instance is a set of tuples, no order can be assumed in which tuples occur when retrieved or otherwise operated on.
- Since there are no duplicate elements in a set, each tuple in a relation instance is unique. In other words, the concatenation of all the attributes is always a candidate key for the relation.
- Since values are atomic, the relational model cannot ascribe to an attribute a domain whose elements are collections (e.g., sets, or lists). For example, in Figure 2, if a `DNA_sequence` has more than one `accession_no`, this cannot be captured with sets of strings as attribute values.

The languages with which data in relation instances can be retrieved and manipulated (thereby effecting state transitions) are discussed in Section III.A.

The relational model was proposed in a form very close to that described in this section in [7]. An excellent formal treatment can be found in [1].

II.B Object-Oriented Data Models

Rather than a single data type, as is the case with the relational data model, an object-oriented data model makes available to application developers a collection of type constructors with which complex, application-specific types can be built. The immediate effect of this approach, in contrast with the relational case, is that more complex application data can be modeled more directly in the database. Instead of having to break down one application concept into many relations, the complex structure needed to model the concept is often directly represented by a combination of type constructors provided by the data model.

The most basic notion in object-oriented data models is that of an *object*, whose purpose is to model application entities. An **object** is characterized by having an *identity*, by being in a certain *state* (which conforms to a prescribed *structure*) and by being capable of interaction via a collection of *methods* (whose specifications characterize the *behavior* of the object).

The **identity** of an object is unique. It is assigned to the object by the DBMS when the object is created, and deleted when the object is destroyed. In contrast with a relational primary key, between these two events the identity of the object cannot be changed, neither by the DBMS nor by application systems. Another contrast is that the identity of an object is distinct and independent from both the state of the object and the methods available for interacting with it.

Objects that are structurally and behaviorally similar are organized into *classes*. A **class** has an associated **extent** which is a set of the identifiers of objects whose structural and behavioral similarity the class reflects. An **object-oriented schema** is a collection of classes plus integrity constraints.

In contrast to the relational case, there is no widespread agreement on which integrity constraints are an intrinsic part of an object-oriented model, but one of the most often used stipulates how many objects of each of the classes related by some relationship type can participate in relationship instances. For binary relationships (which are the most commonly used), there are three possibilities, viz., *one-to-one*, *one-to-many* and *many-to-many*.

Figure 3 depicts in diagrammatic form a small fragment of an object-oriented database schema with four classes, viz., `gene`, `allele`, `wild_type` and `mutant`. The latter two are subclasses of `allele` as indicated by the arrow head and explained below. Note also the relationship type from `gene` to `allele` named `has_alleles`. The annotations `1` and `*` specify a one-to-many constraint to the effect that an object of class `gene` may be related to many objects of class `allele`, but one allele is only related to one gene.

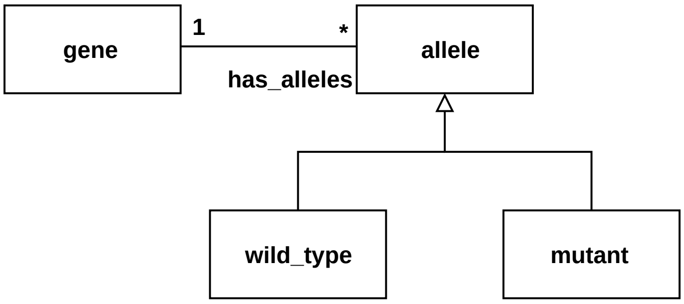

Figure 3: A Fragment of an Object-Oriented Schema Diagram

The **structure** of an object is characterized by a collection of *properties* and *relationships*. An object is in a certain **state** when all the properties and relationships defined by its structure are assigned legal values. Both **properties** and **relationships** resemble attributes in the relational model, with the following main differences:

1. the domain of a property need not be atomic insofar as the type of its values can be structures or collections by the application of the available **type constructors** (typically, `tuple_of`, `set_of`, `list_of` and `bag_of`);

2. the domain of a relationship is a set of object identities (or power set thereof).

Figure 4 shows how the structure of the `gene` class from Figure 3 might be specified in an object-oriented data model. Notice the use of type constructors in assigning domains to the property `citation` and the relationship `has_alleles`. Notice the non-atomic domain `codon_structure`. Notice also that the identity of individual `gene` objects is not dependent on any of the specifications in Figure 4: it is the DBMS that assigns identifiers, manages their use (e.g., in establishing a relationship between `gene` objects and `allele` objects) and annuls them if objects are destroyed. Notice, finally, that while an object-oriented database allows relationships to be expressed as simple properties (e.g., `protein_id` and `organism_id` establish relationships between `gene` and `protein` and `organism`, respectively), it is only by explicitly expressing them as relationships that an object-oriented DBMS would have the information needed to enforce the kind of referential integrity that relational systems enforce for attributes designated as foreign keys.

```
tuple_of codon_structure{
        initiation_codon     :  string;
        codon_sequence       :  list_of(string);
        termination_codon    :  string; }
class gene{
        properties
                standard_name :  string;
                sequence      :  codon_structure;
                function      :  string;
                citation      :  list_of(string);
                protein_id    :  protein;
                organism_id   :  organism;
        relationships
                has_alleles   :  set_of(allele);
        methods
                ... }
```

Figure 4: A Structure for the `gene` Class from Figure 3

The **behavior** of an object is characterized by a collection of *method specifications*. A **method specification** is an application-defined declaration of the name and parameter types for a method that can be invoked on the object. By **method invocation** on an object o with identity i is meant, roughly, that the method call refers to i (thereby identifying o as the object to which the method is addressed) and is meant to be executed in the context of the current state of o using o's assigned behavior (where that execution may itself invoke methods defined on objects other than o). A **method definition** is an executable specification of a function or procedure.

Typically, **accessor methods** (i.e., those that return the current value of a property or relationship) and **update methods** (i.e., those that modify to a given value the current value of a property or relationship) are automatically generated at schema compilation time. A popular approach to naming such methods is to prefix the property or relationship name by `get_` and `set_`, respectively. If type constructors are involved then more methods are made available, accordingly. For example, with reference to Figure 4, methods would be generated that implement (implicit) specifications such as `get_citation() -> list_of(string)`, `insert_citation(string)`. The former invoked on a `gene` object o would return a list of strings currently assigned as the value of the `citation` property of o, while the latter would cause a state transition in o by adding the string passed as argument to the list of strings currently assigned as the value of the `citation` property of o.

```
class gene{
        properties
                ...
        relationships
                ...
        methods
                specification
                        compute_G+C_density   ()           -> float;
                        retrieve_region_mRNA  (int, int)   -> list_of(mRNA);
                        ...
                definition
                        ...
        ... }
```

Figure 5: (Partial) Behavior of the `gene` Class from Figure 4

Figure 5 shows how behavior for the `gene` class from Figure 4 might be specified (but method definitions are omitted). Thus, a `gene` object will respond to a request for its density of guanine and cytosine pairs with a float value. Also, given a region inside it (as defined by a start and an end position in its nucleotide sequence) a `gene` object will respond to a request for mRNA found inside the region with a list of identifiers of `mRNA` objects (assuming a class `mRNA` to have been declared in the schema).

The requirement that one must always invoke methods to interact with objects gives rise to a distinctive feature of object-oriented approaches, viz., **encapsulation**, by which is meant that the state of an object is protected from direct manipulation or scrutiny. This is beneficial in the sense that it allows application developers to ignore the structural detail of the state of objects: all they need to know is the behavior of the object, i.e., which methods can be invoked on it. This, in turn, allows the detailed structure of objects to change without necessarily requiring changes in the application programs that use them.

The approach of organizing objects into classes gives rise to another distinctive feature of object-oriented approaches, viz., **inheritance**, by which is meant a subsumption relationship between the extents of two classes C and C'. Thus, a class C is said to **extend** (or **specialize**, or **inherit from**) a class C' if all objects in the extent of C have the same properties, relationships, and compatible method specifications as (but possibly more than) the objects in the extent of C'. In this case, C is said to be a **subclass** of C'. Thus, inheritance in object-oriented data models is both structural and behavioral (but the compatibility constraints for the latter are beyond the scope of this discussion). There are many motivations for inheritance mechanisms as described, including a stimulus to reuse, a more direct correspondence with how applications are conceptualized, etc.

In the presence of inheritance, not only can a method name be **overloaded** (i.e., occur in more than one method specification), but it can also be **overridden** (i.e., occur in more than one method definition). In this case, method invocations can be attended to by more than one method definition and the decision as to which one to use usually depends on the type of the object that the method was invoked on. This selection process which binds a method definition to a method invocation can only be carried out at run time, therefore this distinctive object-oriented feature is known as **late** (or **dynamic**) **binding**.

To summarize, an object-oriented data model has the following main distinctive features:

- Objects have not only structure but behavior as well.
- Objects have a unique identity that is independent of their state and behavior.
- Values can be drawn from complex domains resulting from the application of type constructors such as `tuple_of`, `set_of`, `list_of` and `bag_of`.
- The state of objects is encapsulated.
- Objects are organized into classes by structural and behavioral similarity.
- Structural and behavioral inheritance is supported.
- Overloading, overriding and late binding are supported.

Because an object-oriented model brings together structure and behavior, much of the functionality required by applications to interact with application objects can be modeled inside the database itself. Nevertheless, most object-oriented database systems provide application program interfaces that allow a programming language to retrieve and manipulate objects.

Unlike the relational model, object-oriented data models cannot all be traced back to a single major proposal. The closest thing to this notion is a manifesto (found, e.g., in [3]) signed by prominent members of the research community whose recommendations are covered by and large in the treatment given in this section. The ODMG industry consortium of object database vendors has proposed an object-oriented data model that is widely seen as the *de facto* standard [5].

II.C Object-Relational Data Models

While object-oriented data models typically relegate relations to the status of one amongst many types that can be built using type constructors (in this case, e.g., `tuple_of`), object-relational data models retain the central role relations have in the relational data model.

However, they relax the domain constraint of the relational data model (thereby allowing attribute values to be drawn from complex domains), they incorporate some of the distinctive features of the object-oriented model such as inheritance and assigned behavior with encapsulated states, and they allow for database-wide functionality to be specified by means of *rules* (commonly referred to as *triggers*, as described in Section IV.B) that react to specific interactions with application entities by carrying out some appropriate action.

As a consequence of the central role that relations retain in object-relational data models, one crucial difference with respect to the object-oriented case is that the role played by object identity is relaxed to an optional, rather than mandatory, feature. Thus, an object-relational DBMS stands in an evolutionary path regarding relational ones, whereas object-oriented ones represent a complete break with the relational approach. In this context, notice that while a `tuple_of` type constructor may allow a relation type to be supported, each tuple will have an identity, and attribute names will be explicitly needed to retrieve and interact with values.

Such differences at the data model level lead to pragmatic consequences of some significance at the level of the languages used to interact with application entities. In particular, while object-oriented data models naturally induce a navigational approach to accessing values, this leads to chains of reference of indefinite length that need to be traversed, or navigated.

In contrast, object-relational data models retain the associative approach introduced by the relational data model. Roughly speaking, this approach is based on viewing the sharing of values between attributes in different relations as inherently establishing associations between relation instances. These associations can then be exploited to access values across different relations without specifically and explicitly choosing one particular chain of references. These issues are exemplified in Sections III.B and III.C.

Since object-relational models are basically hybrids, their notions of schema and instance combine features of both relational and object-oriented schemas and instances. The object-relational approach to modeling the *gene* entity type is the same as the object-oriented one depicted in Figures 4 and 5 but for a few differences, e.g., relationships are usually not explicitly supported as such by object-relational data models.

As in the object-oriented case, object-relational data models were first advocated in a concerted manner by a manifesto [14] signed by prominent members of the research community. A book length treatment is available in [13]. Any undergraduate textbook on Database Systems (e.g., [4, 8]) can be used to complement the treatment of this and the section that follows.

III Database Languages

A DBMS must support one or more languages with which application developers can set up, populate, scrutinize and maintain data. In principle, different purposes are best served by different database languages, but in practice, DBMSs tend to use one language to cater for more than one purpose.

The kinds of **database languages** that must in principle be supported in DBMSs are:

1. **Data definition languages**: which are used to declare schemas (perhaps including application-specific integrity constraints).

2. **Data manipulation languages**: which are used to retrieve and manipulate the stored data.

Data manipulation languages can be further categorized as follows:

1. **Query languages**: which most often are high-level, declarative, and computationally incomplete (i.e., not capable of expressing certain computations, typically due to the lack of support for updates or for recursion or iteration).

2. **Procedural languages**: which most often are low-level, imperative, and computationally complete

Besides these, most DBMSs provide interfacing mechanisms with which developers can implement applications in a general-purpose language (referred to in this context as a **host language**) and use the latter to invoke whatever operations are supported over the stored data (because they are part of the behavior of either the data types made available by the data model or the application types declared in the application schema).

III.A Relational Database Languages

SQL is the ISO/ANSI standard for a relational database language. SQL is both a data definition and a data manipulation language. It is also both a query language and capable of expressing updates. However, SQL is not computationally complete, since it offers no support for either recursion or iteration. As a consequence, when it comes to the development of applications, SQL is often embedded in a host language, either one that is specific to the DBMS being used or a general purpose language for which a query language interface is provided.

Figure 6 shows how the schema illustrated in Figure 1 can be declared in SQL. Notice that SQL includes constructs to declare integrity constraints (e.g., a referential integrity on from `DNA_sequence` to `organism`) and even an action to be performed if it is violated (e.g., cascading deletions, by deleting every referring tuple when a referenced tuple is deleted).

```
CREATE SCHEMA genomics AUTHORIZATION DBA;
CREATE TABLE  genomics.DNA_sequence(
      sequence_id          VARCHAR(10)   NOT NULL
      accession_no         VARCHAR(10)   NOT NULL,
      protein_id           VARCHAR(10)   NOT NULL,
      organism_id          VARCHAR(10)   NOT NULL
      PRIMARY KEY (sequence_id),
      FOREIGN KEY (organism_id)
            REFERENCES genomics.organism(organism_id)
            ON DELETE CASCADE    ON UPDATE CASCADE );
CREATE TABLE  genomics.organism(
      organism_id          VARCHAR(10)   NOT NULL,
      common_name          VARCHAR(50),
      up_node              VARCHAR(30)
      PRIMARY KEY (organism_id) );
```

Figure 6: SQL Declarations for the Schemas in Figure 1

SQL can also express insertions, deletions and updates, as indicated in Figure 7. Note in Figure 7 that on insertion it is possible to omit null values by listing only the attributes for which values are being supplied. Note also that the order of insertion in Figure 7 matters, since referential integrity constraints would otherwise be violated. Finally, note that, because of cascading deletions, the final statement will also delete all tuples in `DNA_sequence` that refer to the primary key of the tuple being explicitly deleted in `organism`.

```
INSERT INTO genomics.organism(organism_id, up_node)
VALUES      ('Poly. tinctorium', 'Polygonaceae');

INSERT INTO genomics.DNA_sequence
VALUES      ('AB003089', 'AB003089', 'BAA78708', 'Poly. tinctorium');

DELETE FROM genomics.organism
WHERE       organism_id = 'Bras. napus';
```

Figure 7: Using SQL to Effect State Transitions in the Relation Instances from Figure 2

After the operations in Figure 7 the state depicted in Figure 2 will have changed to that shown in Figure 8.

DNA_sequence			
sequence_id	accession_no	protein_id	organism_id
TRBG361	X56734	AA40058	Trif. repens
AB003089	AB003089	BAA78708	Poly. tinctorium
...	...	...	...

organism		
organism_id	common_name	up_node
Trif. repens	white clover	Papilionoideae
Poly. tinctorium	NULL	Polygonaceae
...	...	...

Figure 8: Updated Relation Instances after the Commands in Figure 7

As a relational query language, SQL always returns results that are themselves relation instances. Thus, the basic constructs in SQL cooperate in specifying aspects of the schema as well as the instantiation of a query result. Roughly, the `SELECT` clause specifies the names of attributes to appear in the result, the `FROM` clause specifies the names of relations contributing data to the result, and the `WHERE` clause specifies, in terms of the relations (and their attributes) mentioned in the `FROM` clause, the conditions which each tuple in the result must satisfy. SQL queries tend to be structured around this combination of `SELECT`, `FROM` and `WHERE` clauses. Figure 9 shows example SQL queries.

Query `RQ1` in Figure 9 returns a unary table, each tuple of which records the `organism_id` of organisms that share the `common_name` of `'white_clover'`. Query `RQ2` returns a binary table relating each `common_name` found in the `organism` table with the `protein_id`s produced by their identified genes. Figure 10 shows the relations instances returned by `RQ1` and `RQ2`.

SQL also supports aggregations (e.g., `COUNT` and `AVG`, which, respectively, count the number of tuples in a result and compute the average value of a numeric attribute in the result), groupings, and sorting.

Embedding SQL into a host language is another approach to retrieving and manipulating data from relational databases. Vendors typically provide a host language for SQL of which the fragment in Figure 11 is an illustration. The fragment in Figure 11 uses a `CURSOR` construct to scan the `organism` relation instance for organisms with no common name. When one is found, rather than leaving the value

```
RQ1:   SELECT  organism_id
       FROM    organism
       WHERE   common_name = 'white clover';

RQ2:   SELECT  o.common_name, s.protein_id
       FROM    organism AS o, DNA_sequence AS s
       WHERE   o.organism_id = s.organism_id;
```

Figure 9: Using SQL to Query the Database State in Figure 8

RQ1
organism_id
Trif. repens

RQ2	
common_name	protein_id
white clover	AA40058
rape	CAA57913
...	...

Figure 10: Results of the SQL Queries in Figure 9

unassigned, the program uses the `UPDATE` construct to set the `common_name` attribute to the string `None`.

```
DECLARE  o_id            CHAR(10);
         o_common_name   CHAR(50);
         CURSOR organism_cursor
         IS     SELECT organism_id, common_name
                FROM organism;
BEGIN    OPEN   organism_cursor;
         LOOP   FETCH    organism_cursor
                INTO     o_id, o_common_name;
                EXIT WHEN organism_cursor%NOTFOUND;
                IF       o_common_name IS NULL
                THEN     UPDATE  organism
                         SET     common_name = 'None'
                         WHERE   organism_id = o_id;
         END LOOP;
         CLOSE  organism_cursor;
END;
```

Figure 11: Embedding SQL in a Host Language to Effect State Transitions

SQL is legally defined by ISO/ANSI standards which are available from those organizations. For comprehensive treatment, a good source is [11]. A detailed treatment of the relational algebra and calculi which underpin SQL can be found in [1].

III.B Object-Oriented Database Languages

In the object-oriented case, the separation between the languages used for data definition, querying and procedural manipulation is more explicit than in the relational case. This is because in object-oriented databases, the syntactic style of SQL is circumscribed largely to the query part of the data manipulation language.

Also, rather than make use of vendor-specific host languages, object-oriented DBMSs either provide interfaces to general-purpose languages or else the DBMS itself supports a persistent programming language strategy to application development (i.e., one in which a distinction between the data space of the program and the database is deliberately not made, which leads to applications which need not explicitly intervene to transfer data from persistent to transient store and back again).

The *de facto* standard for object-oriented databases is the proposal by the ODMG consortium of vendors. The ODMG standard languages are ODL, for definition, and OQL (which extends the query part of SQL), for querying. The standard also defines interfaces for a few widely-used general-purpose languages. Figure 4 could be declared in ODL as shown in Figure 12.

Note how ODL allows inverse relationships to be named, as a consequence of which referential integrity is enforced in both directions.

Two OQL queries over the `gene` class in Figure 12 are given in Figure 13. Query `OQ1` returns a set of complex values, i.e., `name-cited` pairs, where the first element is the `standard_name` of an instance of the `gene` class and the second is the list of strings stored as the value of the `citation` attribute for that instance. Query `OQ2` returns the `common_name` of `organisms` associated with `genes` that have `alleles`.

```
class gene{
    (extent       Genes)
    attribute     string         standard_name ;
    attribute     struct         codon_structure {
                                 string initiation_codon,
                                 list<string> codon_sequence,
                                 string termination_codon } sequence;
    attribute     string         function;
    attribute     list<string>   citation;
    attribute     protein        protein_id;
    attribute     organism       organism_id;
    relationship  set<alleles>   has_alleles
                  inverse        allele ::  is_allele_of;
           ... }
```

Figure 12: ODL to Specify the gene Class in Figure 4

Note the use of the COUNT aggregation function over a collection value. Note, finally, the denotation g.organism_id.common_name (known as a *path expression*). Path expressions allow a navigational style of access.

```
OQ1:   SELECT  struct(name   :  g.standard_name
                      cited  :  g.citation)
       FROM    g IN gene;

OQ2:   SELECT  g.organism_id.common_name
       FROM    g AS gene
       WHERE   COUNT(g.has_alleles) > 0;
```

Figure 13: Using OQL to Query the gene Class in Figure 12

For ODMG-compliant DBMSs, the authoritative reference on ODL and OQL is [5]. A more formal treatment of some of the issues arising in object-oriented languages can be found in [1].

III.C Object-Relational Database Languages

The proposed standard for object-relational database languages is SQL-99. Figure 14 shows how Figure 4 could be specified in SQL-99. Note the use of ROW TYPE to specify a complex domain, the use of REF to denote tuple identifiers and the use of type constructors such as SET and LIST. Note also that, unlike ODL (cf. Figure 12), in SQL-99 inverse relationships are not declared. Note, finally, how gene is modeled as including operations, as indicated by the keyword FUNCTION introducing the behavioral part of the specification of gene.

```
CREATE ROW TYPE  codon_structure(
                        initiation_codon     CHAR(3),
                        codon_sequence       LIST(CHAR(3)),
                        termination_codon    CHAR(3));
CREATE TABLE     codon_sequence       OF TYPE       codon_structure;

CREATE TYPE      gene(  standard_name VARCHAR(50),
                        sequence      REF (codon_sequence),
                        function      VARCHAR(30),
                        citation      LIST(VARCHAR(30)),
                        protein_id    REF(protein_tuple) REFERENCES (protein),
                        organism_id   REF(organism_tuple) REFERENCES (organism),
                        has_alleles   SET(REF(allele_tuple)) REFERENCES (allele),
                        FUNCTION ...
                        ... );
```

Figure 14: SQL-99 to Specify the gene Entity in Figure 3

Two SQL-99 queries over the gene type in Figure 14 are given in Figure 15. Query ORQ1 returns a binary table relating the standard_name of each gene with the common_name of organisms where the gene is found. Query ORQ2 returns the common_name of organisms associated with genes that have alleles. Note that in SQL-99 path expressions use the symbol -> to dereference identifiers and (not shown in Figure 15) the symbol .. to denote attributes in row types.

```
ORQ1:  SELECT  standard_name, SET(organism_id -> common_name)
       FROM    gene;

ORQ2:  SELECT  g.organism_id -> common_name
       FROM    gene AS g
       WHERE   (SELECT COUNT(a) FROM a IN g.has_alleles)) > 0;
```

Figure 15: Using SQL-99 to Query the gene Type in Figure 14

SQL-99 is legally defined by ISO/ANSI standards which are available from those organizations.

IV Advanced Models and Languages

The previous sections have described mainstream database models and languages, presenting facilities that are in widespread use today. This section introduces several extensions to the mainstream data models and languages with facilities that allow the database to capture certain application requirements more directly.

IV.A Deductive Databases

Deductive database systems can be seen as bringing together mainstream data models with logic programming languages for querying and analyzing database data. Although there has been research on the use of deductive databases with object-oriented data models, this section illustrates deductive databases in the relational setting, in particular making use of Datalog as a straightforward deductive database language [6].

A **deductive (relational) database** is a Datalog program. A **Datalog program** consists of an *extensional database* and an *intensional database*. An **extensional database** contains a set of facts of the form:

$$p(c_1, ..., c_m)$$

where p is a predicate symbol and $c_1, ..., c_m$ are constants. Each predicate symbol with a given arity in the extensional database can be seen as analogous to a relational table. For example, the table `organism` in Figure 2 can be represented in Datalog as:

```
organism('Bras. napus', 'rape', 'Brassicaceae')
organism('Trif.repens', 'white clover', 'Papilionoideae')
```

Traditionally, constant symbols start with a lower case letter, although quotes can be used to delimit other constants.

An **intensional database** contains a set of rules of the form:

$$p(x_1, ..., x_m) : -q_1(x_{11}, ..., x_{1k}), ..., q_j(x_{j1}, ..., x_{jp})$$

where p and q_i are predicate symbols, and all argument positions are occupied by variables or constants.

Some additional terminology is required before examples can be given of Datalog queries and rules. A **term** is either a constant or a variable – variables are traditionally written with an initial capital letter. An **atom** $p(t_1, ..., t_m)$ consists of a predicate symbol and a list of arguments, each of which is a term. A **literal** is an atom or a negated atom $\neg p(t_1, ..., t_m)$. A **Datalog query** is a conjunction of literals.

```
DQ1:   { OID | organism(OID, 'white clover', _) }

DR1:   species_protein(CommonName, ProteinId) :-
              organism(OID, CommonName, _),
              dna_Sequence(_, _, ProteinId, OID).

DQ2:   { ProteinId | species_protein('white clover', ProteinId) }
```

Figure 16: Using Datalog to Query the Database State in Figure 8

Figure 16 shows some queries and rules expressed in Datalog. The queries `DQ1` and `DQ2` are expressed using a set comprehension notation, in which the values to the left of the | are the result of the Datalog query to the right of the |.

Query `DQ1` retrieves the organism identifier of `'white clover'`. Informally, the query is evaluated by unifying the query literal with the facts stored in the extensional database. Query `DQ2` retrieves the identifiers of the proteins in `'white clover'`. However, the predicate `species_protein` representing the relationship between the common name of a species and the identifiers of its proteins is not part of the extensional database, but rather is a rule defined in the intensional database. This rule, `DR1` in Figure 16, is essentially equivalent to the SQL query `RQ2` in Figure 9, in that it retrieves pairs of species common names and protein identifiers that satisfy the requirement that the proteins are found in the species. In `DR1`, the requirement that a `dna_Sequence` be found in an `organism` is represented by the atoms for `dna_Sequence` and `organism` sharing the variable `OID`, i.e., the organism identifier of the `organism` must be the same as the organism identifier of the `dna_Sequence`.

Deductive databases can be seen as providing an alternative paradigm for querying and programming database applications. A significant amount of research has been conducted on the efficient evaluation of recursive queries in deductive languages [6], an area in which deductive languages have tended to be more powerful than other declarative query languages. Recursive queries are particularly useful for searching through tree or graph structures, for example in a database representing a road network or a circuit design. A further feature of deductive languages is that they are intrinsically declarative, so programs are amenable to evaluation in different ways – responsibility for finding efficient ways of evaluating deductive programs rests as much with the query optimizer of the DBMS as with the programmer.

Although there have been several commercial deductive database systems, these have yet to have a significant commercial impact. However some of the features of deductive languages, for example relating to recursive query processing, are beginning to be adopted by relational vendors, and are part of the SQL-99 standard.

IV.B Active Databases

Traditional DBMSs are **passive** in the sense that commands are executed by the database (e.g., query, update, delete) as and when requested by the user or application program. This is fine for many tasks, but where a DBMS simply executes commands without paying heed to their consequences, this places responsibilities on individual programmers that they may struggle to meet [12].

For example, imagine that it is the policy of an organization managing a repository of DNA sequence data that whenever a new sequence is provided with a given `sequence_id`, the previous sequence is recorded, along with the date when the change takes place and the name of the user making the update. This straightforward policy may, however, be difficult to enforce in practice, as there may be many different ways in which a new replacement can be provided – for example, through a program that accepts new information from a WWW interface, or through an interactive SQL interface. The policy should be enforced in a consistent manner, no matter how the sequence has come to be updated. In an **active database**, the database system is able to respond automatically to events of relevance to the database, such as the updating of a table, and thus the database can provide centralized support for policies such as that described above.

Active behavior is described using **event condition action** or **ECA** rules. These rules have three components:

event: a description of the happening to which the rule is to respond. In the example above, this could be the updating of the relation storing the sequence data.

condition: a check on the context within which the event has taken place. The condition is either a check on the parameters of the event or a query over the database.

action: a program block that indicates the reaction that the system should take to the event in a context where the condition is true.

Active rules will now be illustrated using the example described above, based around a table `sequence` with attributes `sequence_id` and `sequence`. A record will be kept of modified sequences in the table `modified_sequence`, with attributes `sequence_id`, `old_sequence`, `date_of_change`, `user` and `update_type`.

The active rule in Figure 17 is written using the SQL-99 syntax. In the SQL standard, active rules are known as **triggers**. The event the trigger is monitoring is `after update of sequence on sequence`, which is raised after any change to the `sequence` attribute of the table `sequence`. The tuple prior to the update taking place is referred to within the trigger as `oldseq`. The example is of a `row` trigger, which means that the trigger responds to updates to individual tuples, so if a single update statement modies 10 rows in the `sequence` table, the trigger will be executed 10 times. In this example there is no condition – the action is executed regardless of the state of the database, etc.

```
create trigger logUpdate
after update of sequence on sequence
referencing old as oldseq
for each row
begin
        insert into modified_sequence values (
                oldseq.sequence_id,
                oldseq.sequence,
                SYSDATE,
                USER,
                'delete');
end
```

Figure 17: Using an active rule to log changes to the `sequence` table.

Writing correct triggers is not altogether straightforward. For the example application, additional triggers would be required, for example, to log the deletion of an existing sequence entry, so that a modification of a sequence as a delete followed by an insert is not omitted from the log. Considerable skill is required in the development and maintenance of large rule bases. However, it is clear that active rules can be used for many different purposes – supporting business rules, enforcing integrity constraints, providing fine tuned authorization mechanisms, etc. As a result, the principal relational database products support active mechanisms, and the SQL-99 standard should encourage consistency across vendors in due course.

V Distribution

A conventional DBMS generally involves a single **server** communicating with a number of user **clients**, as illustrated in Figure 18. In such a model, the services provided by the database are principally supported by the central server, which includes a secondary storage manager, concurrency control facilities, etc, as described in Section I.

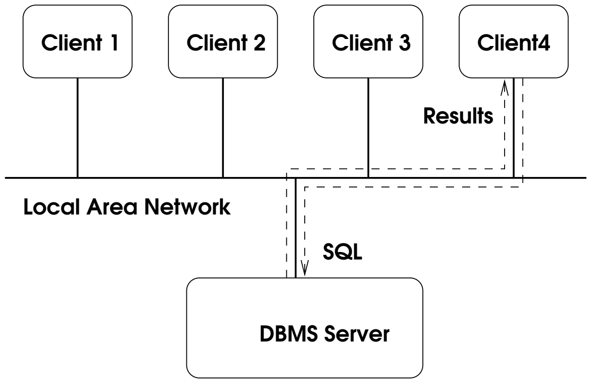

Figure 18: A client-server database architecture.

In relational database systems, clients generally communicate with the database by sending SQL query or update statements to the server as strings. The server then compiles and optimizes the SQL, and returns any result or error reports to the client.

Clearly there is a sense in which a client-server database is distributed, in that application programs run on many clients that are located in different parts of the network. However, in a client-server context there is a single server managing a single database described using a single data model. The remainder of this section introduces some of the issues raised by the relaxation of some of these restrictions.

V.A Distributed Databases

In a distributed database, data is stored in more than one place, as well as being accessed from multiple plases, as in the client-server model. A typical distributed DBMS architecture is illustrated in Figure 19, in which there are two servers, each of which is accessed by multiple clients.

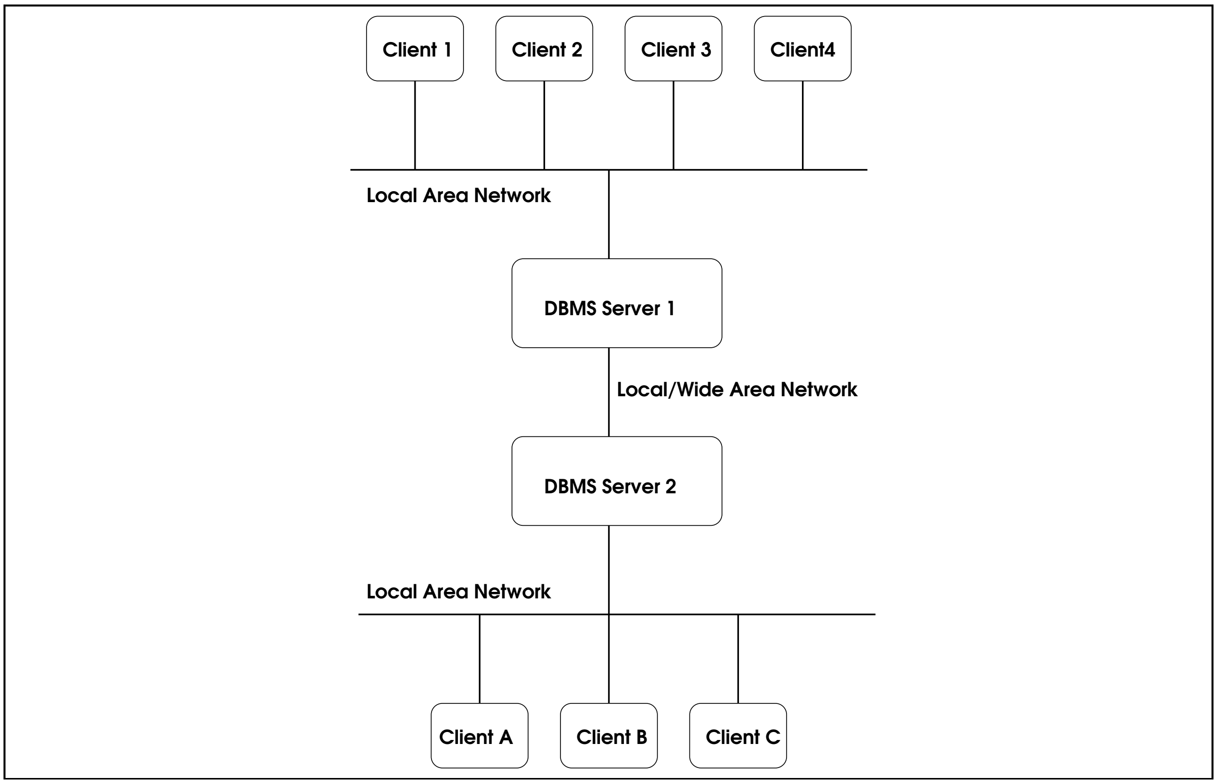

Figure 19: A distributed database architecture.

Distributed databases can come into being for different reasons. For example, a multinational organization may have distinct databases for each of its national subsidiaries, but some tasks may require access to more than one of these databases (e.g., an item that is out of stock in one country may be available in another). Another situation in which multiple databases may have to be used together is where organizations merge, and it becomes necessary to support some measure of interoperation across their independently developed information systems.

The context within which a distributed database comes into being significantly influences the complexity of the resulting system. The nature of a distributed database can be characterized by the following features:

Fragmentation: The fragmentation of data in a distributed database describes how individual data items are partitioned across the database servers. For example, in the genomic database of Figure 1, one form of fragmentation could lead to all the sequences from an `organism` being stored on the same server. This might happen, for example, if the sequencing of an organism is being carried out at a single sight. This form of fragmentation is known as **horizontal fragmentation**, as the tuples in the tables `organism` and `DNA_sequence` are partitioned on the basis of their `organism_id` values. In **vertical fragmentation**, some of the attributes of a table are placed on one server, and other attributes on another.

In many applications, it is difficult to arrange a straightforward fragmentation policy – for example, if distributed databases are being used where two organizations have recently merged, it is very unlikely that the existing databases of the merging organizations will have made use of similar fragmentation schemes.

Homogeneity: In a **homogeneous** distributed database system, all the servers support the same kind of database, e.g., where all the servers are relational, and in particular where all the servers have been provided by the same vendor. It is common, however, for more than one database paradigm to be represented in a distributed environment, giving rise to a **heterogeneous** distributed database.

Transparency: In Figure 19, a user at `Client 1` can issue a request that must accesses or update data from either of the servers in the distributed environment. The level of transparency supported indicates the extent to which users must be aware of the distributed environment in which they are operating [4]. Where there is **fragmentation transparency**, requests made by the user need not take account of where the data is located. Where there is **replication transparency**, requests made by the user need not take account of where data has been replicated in different servers, for example, to improve efficiency. Where there is **language transparency**, the user need not be concerned if different servers support different query languages. The greater the level of transparency supported, the more infrastructure is required to support transparent access. Distributed query processing and concurrency control present particular challenges to database systems developers [9].

V.B Data Warehouses

Traditional database systems have tended to be designed with an emphasis on **on line transaction processing** (OLTP), in which there are large numbers of short transactions running over the database. In our biological example, a typical transaction might add a new sequence entry into the database. Such transactions typically read quite modest amounts of data from the database, and update one or a few tuples.

However, the data stored in an organization's databases is often important for management tasks, such as decision support, for which typical transactions are very different from those used in OLTP. The phrase **on line analytical processing** (OLAP) has been used to characterize this different kind of access to databases, in which typical transactions read rather than update the database, and may perform complex *aggregation* operations on the database. As the data required for OLAP often turns out to be stored in several OLTP databases, it often proves to be necessary to replicate data in a system designed specifically to support OLAP – such a replicated store is known as a **data warehouse**.

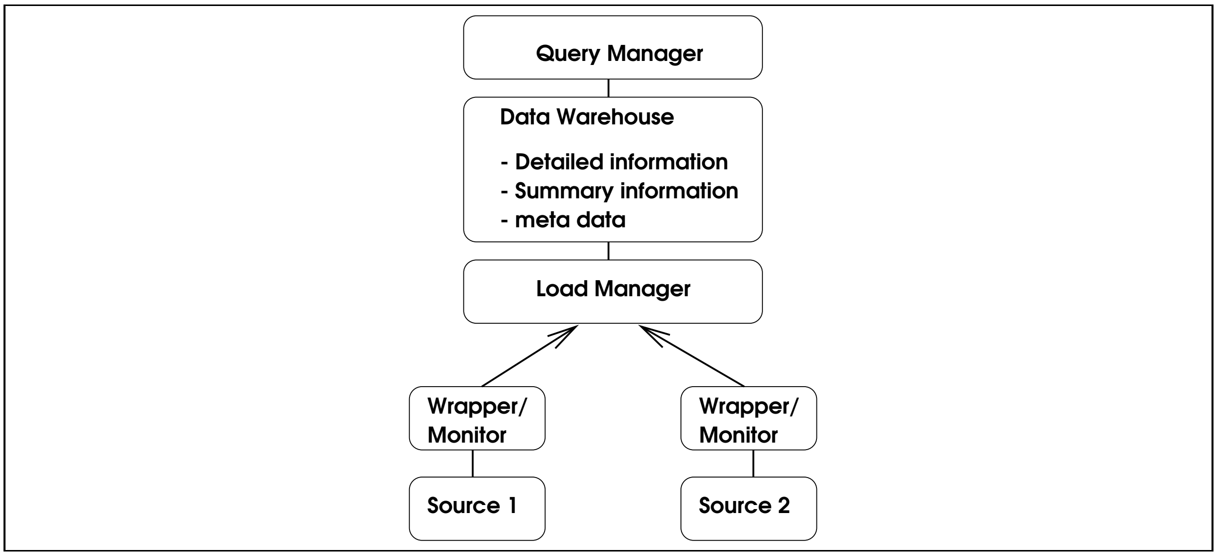

Figure 20: A data warehouse architecture.

A data warehouse architecture is illustrated in Figure 20 [2]. A data warehouse contains three principal categories of information: detailed information, for example, in the genomic context, this may be the raw sequence data or other data on the function of individual protein; summary information, for example, in the genome context, this may record the results of aggregations, such as the numbers of genes associated with a given function in each species; and meta data, which describes the information in the warehouse and where it came from.

An important prerequisite for the conducting of effective analyses is that the warehouse contains appropriate data, of adequate quality, which is sufficiently up-to-date. This means that substantial effort has to be put into loading the data warehouse in the first place, and then keeping the warehouse up-to-date as the sources change. In Figure 20, data essentially flows from the bottom of the figure to the top. Sources of data for the warehouse, which are often databases themselves, are commonly **wrapped** so that they are syntactically consistent, and **monitored** for changes that may be relevant to the warehouse (e.g., using active rules). The **load manager** must then merge together information from different sources, and discard information that doesn't satisfy relevant integrity constraints. The **query manager** is then responsible for providing comprehensive interactive analysis and presentation facilities for the data in the warehouse.

VI Conclusions

This chapter has provided an introduction to databases, and in particular to the models, languages and systems that allow large and complex data intensive applications to be developed in a systematic manner. DBMSs are now quite a mature technology, and almost every organization of any size uses at least one such system. The DBMS will probably be among the most sophisticated software systems deployed by an organization.

This is not to say, however, that research in database management is slowing down. Many data intensive applications do not use DBMSs, and developments continue, with a view to making database systems amenable for use in more and more different kinds of application. For example, spatial, temporal, multimedia and semi-structured data are difficult to represent effectively in many database systems, and many issues relating to distributed information management continue to challenge researchers.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] S. Anahory and D. Murray. *Data Warehousing in the Real World.* Addison-Wesley, 1997.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. B. Zdonik. The Object-Oriented Database System Manifesto. In *[10]*, pages 223–240, 1990.

[4] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems: Concepts, Languages and Architectures.* McGraw-Hill, 1999.

[5] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard: ODMG 3.0.* Morgan Kaufman, 2000. ISBN 1-55860-647-5.

[6] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[7] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970. Also in CACM 26(1) January 1983 p. 64–69.

[8] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, Reading, MA., USA, 3rd. edition, 2000. ISBN 0-201-54263-3.

[9] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 2000.

[10] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Deductive and Object-Oriented Databases (First International Conference DOOD'89, Kyoto)*, Amsterdam, The Netherlands, 1990. Elsevier Science Press (North-Holland). ISBN 0-444-88433-5.

[11] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufman, 1993. ISBN 1-55860-245-3.

[12] N. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 1(31):63–103, 1999.

[13] M. Stonebraker and P. Brown. *Object-Relational DBMS: Tracking the Next Great Wave.* Morgan Kaufman, 1999. ISBN 1-55860-452-9.

[14] M. Stonebraker, L. A. Rowe, B. G. Lindsay, J. Gray, M. J. Carey, M. L. Brodie, P. A. Bernstein, and D. Beech. Third-Generation Database System Manifesto - The Committee for Advanced DBMS Function. *SIGMOD Record*, 19(3):31–44, September 1990.