DATABASE LANGUAGES

BACKGROUND

A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. In general, the user accesses and manipulates the database with a data definition language (DDL) to define database schemata. After the schemata are compiled and the database is populated with the data, the user uses a data manipulation language (DML) to retrieve, insert, delete, or modify the data stored in the database.

There are basically two types of DMLs. A low-level or *procedural* DML can be used to specify complex database operations in a concise manner. In this case the user has to know how to execute the operations in the right order. Otherwise, a high-level or *declarative* DML can be used: The user only specifies what the result is, leaving the decisions about how to execute the operations to the DBMS. Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user cannot specify the procedures to access the data, these languages may not be as efficient as procedural languages in terms of performance.

Low-level DML statements may be embedded in a general purpose programming language such as COBOL, Pascal, or C. These languages are also referred to as *record-at-time* DMLs because they retrieve and process each individual record from a set of records at a time. High-level DMLs can specify and retrieve many records in a single statement and hence are referred to as *set-at-time* DMLs. Whenever a DML, whether high level or low level, is embedded in a general purpose programming language, the latter is called the *host language,* and the DML is called the *data sublanguage.* On the other hand, a high-level DML used in a stand-alone, interactive manner is called a *query language.*

A major criterion used to classify a database language is the data model based on which the language is defined. Conventional data models employed in database languages include the relational, network, and hierarchical models. Among them, the relational model has been successfully used in most commercial database management systems. This is due to the fact that relational database languages can provide high-level query specifications and set-at-time retrievals, whereas network and hierarchical database languages can only support low-level query and record-at-time retrievals. A comparison among the three types of database languages is shown in Table 1. Later we will discuss some modern database languages developed based on the object-oriented, object relational, temporal, active, and deductive data models.

RELATIONAL DATA MODEL, RELATIONAL ALGEBRA, AND RELATIONAL CALCULUS

The relational data model was introduced by Codd (2,3). It provides the simplest and the most uniform structure among all the data models. A relational database consists of a collection of tables (relations). A table is a two-dimensional structure, where a row (*tuple*) represents a collection of related data values and a column (*attribute*) represents the role played by some domain in the table. A *super key* is a set of attributes that can uniquely identify the tuples in a relation. A major restriction of the relational data model is that each attribute has to have a *single* value: no multivalues or composite values can be carried by an attribute. A relation satisfying this requirement is said to be in the *first normal form.*

The relational data model comes with two DMLs: the relational algebra and the relational calculus, where the relational algebra is procedural and the relational calculus declarative. The basic operators in relational algebra are union, difference, selection, projection, and Cartesian product. The union, difference, and Cartesian product operations come directly from the mathematical set theory. The selection operation takes a relation and selects from the relation those tuples that satisfy some conditions. The projection operation chooses certain attributes from a relation. Finally, the join operation (which can be derived from the basic operations) combines two relations into one based on the common attributes. Different from the relational algebra, which is procedural in nature, a query in the relational calculus is expressed as $\{t|P(t)\}$, where t is a *tuple variable* that designates a typical tuple in the answer set and $P(t)$ is a set of predicates combined by logical connectives that qualify the attributes of t. It can be shown that the relational algebra and the relational calculus are identical in expressive power. In other words, any query that can be specified in the relational algebra can also be specified in the relational calculus, and vice versa. A relational language L is *relational complete* if we can express in L any query that can be expressed in the relational calculus. Therefore relational completeness is an important criterion for comparing the expressive power of relational languages. Most commercial query languages have a higher expressive power than that of the relational algebra or calculus due to the introduction of additional operations such as aggregate functions, grouping, and ordering.

STRUCTURED QUERY LANGUAGE (SQL)

Structured Query Language (SQL) is a declarative query language that was developed based on a combination of the rela-

Table 1. A Comparison of Relational, Network, and Hierarchical Database Languages

	Navigational	Set-at-a-Time	Query Specification	Query Optimization
Relational languages	No	Yes	Declarative	System
Network languages	Yes	No	Procedural	User
Hierarchical languages	Yes	No	Procedural	User

tional algebra and relational calculus (2). It was originally implemented in a relational DBMS called SYSTEM R developed by IBM. Over the years it has evolved to be the standard query language for commercial (relational) database management systems. SQL is considered a comprehensive language that supports data definition, data manipulation, and view definition.

Data Definition

The basic commands for data definition include *CREATE, ALTER,* and *DROP,* which defines the attributes of a relation, adds an attribute to a relation, and deletes a relation, respectively. The basic format of the *CREATE* command is

```
CREATE TABLE table name <attribute name>: <attribute
   type>[<constraints>]
```

where each attribute is given its name, a data type that defines its domain of values, and possibly some constraints. The data types are limited to system-defined data types such as numbers and character strings. Since SQL allows NULL (which means "unknown") to be an attribute value, the constraint "NOT NULL" can be specified on an attribute if NULL is not allowed for that attribute. A table defined by the *CREATE* command is called a *base table,* which is physically stored in the database. Base tables are different from *virtual tables* (*views*), which are not necessarily physically stored in the database. The following example shows how a University-personnel table can be created using the above command:

```
CREATE TABLE University-personnel
   <pname: char(10) NOT NULL, UNIQUE,
   residence: char(30),
   birth-date: date NOT NULL>
```

If the *University-personnel* table is no longer needed, we can delete the table with the following command:

```
DROP TABLE University-personnel
```

If we want to add an attribute to the relation, we can use the *ALTER* command. In this case all tuples of the relation will have NULL as the value of the new attribute. For example, we can add an attribute 'salary' with the following command:

```
ALTER TABLE University-personnel
ADD salary <integer>
```

Data Manipulation—Querying

A basic SQL query consists of three clauses:

```
SELECT   <attribute list>
FROM     <relation list>
WHERE    <condition>
```

where the *SELECT* clause identifies a set of attributes to be retrieved, the *FROM* clause specifies a list of tables to be used in executing the query, and the *WHERE* clause consists of a set of predicates that qualifies the tuples (of the relations involved) to be selected by the query in forming the final result. Therefore a query expressed in the form above has the following intuitive meaning: *Retrieve <attribute list> of those tuples satisfying <condition> from <relation list>*. The following are two example queries, assuming that the relations *University-personnel, Car-ownership, Membership* are defined as

University-personnel (pname, residence, birth-date, salary)
Car-ownership (pname, cname)
Membership (pname, society)

Query 1. Retrieve the names and residences of all university personnel who were born on June 25, 1970.

```
SELECT   pname, address
FROM     University-personnel
WHERE    birth-date = '6/25/75'
```

Query 2. Retrieve the names and residences of all university personnel who own a 'Taurus'.

```
SELECT   pname, residence
FROM     University-personnel,     Car-
         ownership
WHERE    (cname = 'Taurus') AND
         (University-personnel.pname
         = Car-ownership.pname)
```

Query 3. Retrieve the names and residences of all university personnel who are members of any society of which 'John' is a member.

```
SELECT   pname, residence
FROM     University-personnel, Membership
WHERE    (University-personnel.pname = Member-
            ship.pname) AND
         (society in (SELECT society
                      FROM Membership
                      WHERE pname = 'John'))
```

Note that Query 3 is a *nested query,* where the inner query returns a set (of values), and it is used as an operand in the outer query.

Several aggregate functions can be applied to collections of tuples in a query, where the collections are formed by a *GROUP BY* clause that groups the answers to the query according to some particular attribute(s) (i.e., each collection consists of answers that have the same value for the attribute(s) specified; in case no *GROUP BY* clause is used, all the answers to the query are considered to be in a single collection). The *COUNT* function returns the number of values associated with a particular attribute in a collection. The *SUM, AVG, MAX,* and *MIN* functions return the sum, average, maximum, and minimum value of a particular attribute in a collection, respectively. The following are two example queries, assuming that the relation *University-personnel* is defined as

University-personnel (pname, residence, birth-date, salary, dname)

Query 4. Find the average salary of all university personnel associated with the 'computer science' department.

SELECT AVG (salary)
FROM University-personnel
WHERE dname = 'computer science'

Query 5. For each department, retrieve the department name and the highest salary.

SELECT dname, MAX(salary)
FROM University-personnel
GROUP BY dname

Data Manipulation—Updates

In SQL three commands can be used to modify a database: DELETE, INSERT, and UPDATE. The DELETE command removes tuples from a table. It includes a WHERE clause to select the tuples to be deleted. Tuples are explicitly removed from only one table one at a time. The following example shows a query to delete those university personnel with birth-date '6/25/70':

DELETE University-personnel
WHERE birth-date = '6/25/70'

The INSERT command inserts one or more tuples into a table. The following example shows a query to insert a new person into a University-personnel table:

INSERT University-personnel
VALUES ('John', 'NULL', '6/25/70')

The UPDATE command modifies certain attribute values of some selected tuples. It includes a WHERE clause to select the tuples and a SET clause that specifies the attributes to be modified and their new values. The following example shows a query to increase by 10% the salary of those university personnel with birth-date later than '6/25/70':

UPDATE University-personnel
*SET salary := salary * 1.1*
WHERE birth-date > '6/25/70'

View Definition

A view is a table which is derived from other (base and/or virtual) tables. The command to define a view is as follows:

CREATE VIEW <table name>
AS <query statement>

The following example shows the definition of a view called 'Young-University-personnel' which are those university personnel born after June 25, 1970:

CREATE VIEW Young-University-personnel
AS SELECT pname, birth-date
FROM University-personnel
WHERE birth-date >'6/25/70'

OBJECT RELATIONAL DATABASE LANGUAGES

Although a relational database language such as SQL is useful, it has several critical limitations: First, only primitive data types such as alphanumerical values are allowed in a relation. Second, each attribute is allowed to carry only one value. Finally, a logical object with complex structure has to be decomposed and stored in several relations. These limitations make it difficult to model complex data such as multimedia, geographical, and engineering information in advanced applications.

The object-oriented data model has emerged to overcome these problems. The basic concepts in the object-oriented model includes encapsulation, object identity, inheritance, and complex objects:

Encapsulation refers to the ability to define a set of operations (*methods*) that can be applied to objects of a particular *class* (*object type*). Thus objects that share the same attributes and methods are grouped into a single class. All accesses to these objects have to be done via one of the associated methods. An object consists of an interface and an implementation; the implementation is private and may be changed without affecting the interface.

Object identity is the ability to identify each object independent of its attribute values. This is typically realized by an object identifier, which is generated by the system. Hence any attribute of an object can be updated without destroying its identity.

Inheritance is the ability to reuse the attributes and methods of an existing class. Object classes can be organized into a type hierarchy based on the **is-a** relationship between a superclass and its subclasses. A subclass can inherit the attributes and methods for its superclass.

Complex objects in the object-oriented model can be defined from previously defined objects in a nested or hierarchical manner.

An object relational data language extends a relational language such as SQL by incorporating the main concepts from the object-oriented model. Consequently, with an object relational language, we can retain the strengths of a relational language such as declarative specification and query optimization. A standard language for object relational systems, called SQL3, has been proposed. Following is a summary of its key features (6).

Class Definition

Conceptually an object can be viewed as a tuple in a relation, and a class can be viewed as a relation, except that an object encapsulates a set of attributes (which are objects as well) and methods into a single unit. For instance, we can define a class 'address' as follows:

CREATE CLASS address {
[*attributes*] *street: char(20),*
city: char(10),
state: char(2);
[*methods*] *change-address();*
}

In the above, the class 'address' consists of two parts: attributes and methods. Each object in the class 'address' contains the attributes 'street', 'city', and 'state'; and they share the same method 'change-address' defined in that class.

Complex Data Types

In the relational model the value of a tuple attribute has to be primitive as required by the first normal form. However, the object relational model extends the relational model so that the value of an attribute can be a complex object or a set/multiset/sequence of complex objects. (This is called a *nested relation.*) For example, we may define a class 'University-personnel' as follows:

CREATE CLASS University-personnel {
  /attributes/ name: char (10),
    residence: REF(address),
    birth-date: date;
  /methods/ compute-age();
}

In the above the declaration *'residence: REF(address)'* states that the value of the attribute 'residence' has to be the identifier of an 'address' object.

Class Hierarchy

Similar classes can share some attributes and methods. Suppose that we define two classes called 'graduate-student' and 'university-staff'. Since graduate students and university staff members are University-personnel, they can be defined naturally with inheritance as follows:

CREATE CLASS graduate-student {
  AS SUBCLASS OF University-personnel;
    student-id: char(10),
    advisor: REF(University-personnel);
}

CREATE CLASS university-staff {
  AS SUBCLASS OF University-personnel;
    /attributes/ years-of-experience: integer;
    /methods/ compute-salary();
}

The subclasses 'graduate-student' and 'university-staff' automatically inherit the attributes (i.e., name, residence, and birth-date) and methods (i.e., compute-age) defined in the superclass 'University-personnel'. In general, a superclass can have one or more subclasses. However, a subclass may have more than one superclass. In this case the subclass inherits the attributes and methods defined in all its superclasses. This is called *multiple inheritance.* For example, we may define a class 'research-assistant' that is a subclass of 'graduate-student' and 'university-staff':

CREATE CLASS research-assistant {
  AS SUBCLASS OF graduate-student;
  AS SUBCLASS OF university-staff;
    /methods/ compute-salary();
}

Operator Overloading

Operator overloading allows the same operator name to be bounded to two or more different implementations, depending on the type of objects to which the operator is applied. For example, the operator '+' can invoke different implementations when applied to operands of different types. In a traditional language, operator overloading is usually limited to system-defined operators. Object-oriented languages extend operator overloading to support user-defined operators, especially in conjunction with a class hierarchy. In this case a method defined in a subclass overwrites any one defined in its superclass. For example, if the way to compute salary for research-assistants is different from that of university-staff, then the class 'research-assistant' can inherit only the name of the method, namely *compute-salary,* from *University-staff.* In this case the subclass can implement its own 'compute-salary'.

OBJECT-ORIENTED DATABASE LANGUAGES

A main difference between a programming language and a database language is that the latter directly accesses and manipulates a database (called *persistent data* in many texts), whereas the objects in the former only last during program execution. In the past two major approaches have been proposed to implement database programming languages. The first is to embed a database language (e.g., SQL) in a conventional programming language; these language are called *embedded languages.* The other approach is to extend an existing programming language to support persistent data and database functionality. These languages are called *persistent programming languages* (6).

However, use of an embedded language leads to a major problem, namely *impedance mismatch.* In other words, conventional languages and database languages differ in their ways of describing data structures. The data type systems in most programming languages do not support database relations directly, thus requiring complex mappings from the programmer. In addition, since conventional programming languages do not understand database structures, it is not possible to check for type correctness.

In a persistent programming language, the above mismatch can be avoided: The query language is fully integrated with the host language, and both share the same type system. Objects can be created and stored in the database without any explicit type change. Also the code for data manipulation does not depend on whether the data it manipulates is short-lived or persistent. Despite of the above advantages, however, persistent programming languages have some drawbacks. Since a programming language accesses the database directly, it is relatively easy to make programming errors that damage the database. The complexity of such languages also makes high-level optimization (e.g., disk I/O reduction) difficult. Finally declarative querying is in general not supported (1).

Several persistent versions of object-oriented languages such as Smalltalk or C++ have been proposed. Unfortunately, there exists no standard for such languages. The object Database Management Group (ODMG, which is a consortium of object-oriented DBMS vendors) has attempted to develop a standard interface, called ODMG 93, for their products. The standard includes a common architecture and a definition for object-oriented DBMS, a common object model with an object definition language, and an object query language for C++ and Smalltalk. Following is a summary of the key features of ODMG 93.

Persistence of Objects

In an object-oriented programming language, objects are transient, since they only exist when a program is executed, and they disappear once the program terminates. In order to integrate such a language with a database, several approaches have been proposed. One simple approach is to divide object classes into persistent classes and transient classes. A persistent class is a class whose objects are stored in the database, and thus can be accessed and shared by multiple programs. However, this approach is not flexible because in many situations it is necesary to have both persistent and transient objects in the same class. One possible solution is to first create a persistent object, called a persistent root; other objects are persistent if they are referred to directly or indirectly from the persistent root. Here the term 'reference' means that an object is a member of a set-valued persistent object or a component of a complex object.

Object Identification

An object-oriented database system assigns a unique identity to each object stored in the database. The unique identity is typically implemented via a unique, system-generated *object identifier.* The value of an object identifier is not visible to the external user, but it is used internally by the system to identify each object uniquely. Several major requirements for object identification need to be considered. *Value independence* requires that an object does not lose its identity even if some attributes change their values over time. *Structure independence* requires that an object does not lose its identity even if some structures change over time. In a relational database system, a set of attributes (i.e., the key attributes) is used to identify the tuples in a relation; therefore value independence cannot be enforced. Another major property of an object identifier is that it is immutable; that is, the value of an object identifier for a particular object should not change. It is also desirable that each object identifier is used only once; which means that even if an object is deleted from the database, its object identifier should not be assigned to another object. These two properties imply that an object identifier does not depend on any attribute values or structures.

When a persistent object is created in a persistent object-oriented database language, it must be assigned a persistent object identifier. The only difference between a transient identifier and a persistent identifier is that the former is valid only when the program that creates it is executing; after the program terminates, the object is deleted and the identifier is meaningless. Additional requirements have been proposed for persistent object identifiers. *Location independence* requires that an object does not lose its identity even if the object moves between the memory and the secondary storage. Another requirement is that an identity persists from one program execution to another. Note that a disk pointer does not satisfy this property, since it may change if the structure of the file system is reorganized.

TEMPORAL DATABASE LANGUAGES

One major drawback of conventional databases is that they do not maintain the history of data. Because each update simply destroys the old data, a database represents only the current state of some domain rather than a history of that domain. The history aspect of databases is important for applications such as project management and equipment maintenance. In general, a temporal database must support time points, time intervals, and relationships involving time such as *before, after,* and *during.* Temporal data models also need to represent time-varying information and time-invariant information separately. The *temporal relational model* (7) extends the relational model based on the above considerations. In this model, a database is classified as two sets of relations *Rs* and *Rt*, where *Rs* is the set of *time-invariant relations* and *Rt* is the set of *time-varying relations.* Every time-variant relation must have two time-stamps (stored as attributes): *time-start* (*Ts*) and *time-end* (*Te*). An attribute value of a tuple is associated with *Ts* and *Te* if it is valid in [*Ts, Te*].

Temporal SQL (TSQL) is an extension of SQL with temporal constructs. TSQL allows both time-varying and time-invariant relations. Thus SQL, a subset of TSQL, is directly applicable to time-invariant relations. TSQL has the following major temporal constructs, which are illustrated with the following example relations:

University-staff (sname, salary, Ts, Te)
Car-ownership (sname, cname, Ts, Te)

where a tuple (*s, c, Ts, Te*) of the 'Car-ownership' relation states the fact that the staff *s* owns a car *c* from time *Ts* to *Te*; that is, the car *c* was owned by the staff *s* continuously during the interval [*Ts, Te*].

WHEN Clause

The WHEN clause is similar to the WHERE clause in SQL. It evaluates the associated temporal predicates by examining the relative chronological ordering of the time-stamps of the tuples involved. The available temporal predicates include predefined temporal comparison operators such as *BEFORE, DURING,* and *OVERLAP.* The binary operator *INTERVAL* is used to specify time intervals, namely [*Ts, Te*]. To qualify a single time-stamp, the unary operators *TIME-START* or *TIME-END* can be used.

The following query shows the use of an *OVERLAP* operator in the *WHEN* clause:

Query. Retrieve the salary of the university-staff with name 'John' when he owned 'Taurus'.

```
SELECT  University-staff.salary
FROM    University-staff, Car-ownership
WHERE   (University-staff.sname = Car-owner-
          ship.sname) AND
        (Car-ownership.sname = 'John') AND
        (Car-ownership.cname = 'Tauraus')
WHEN    University-staff.INTERVAL OVERLAP Car-
          ownership.INTERVAL
```

TIME-SLICE Clause

The TIME-SLICE clause specifies a time period or a point of time point. It selects only those tuples from the underlying relations that are valid for the specified time period or time point. The following query shows the use of a *TIME-SLICE* operator in the *WHEN* clause:

Query. Retrieve the changes of salary during the years 1983–1990 for all university-staff whose car was 'Taurus'.

```
SELECT   University-staff.sname, salary, University-
           staff.TIME-START
FROM     University-staff, Car-ownership
WHERE    (University-staff.sname = Car-owner-
           ship.sname) AND
         (Car-ownership.cname = 'Tauraus')
WHEN     University-staff.INTERVAL OVERLAP Car-
           ownership.INTERVAL
         TIME-SLICE year [1983, 1990]
```

Retrieval of Time-Stamps

To retrieve time points or intervals that satisfy certain conditions, the target list of time-stamps should be specified in the SELECT clause. This target list may include the unary operators *TIME-START* or *TIME-END*. If more than one relation is involved, then new time-stamp values are computed based on the tuples involved. TSQL allows an *INTER* operator to be applied in the target list. The *INTER* operator takes two time intervals and returns another interval which is their intersection, assuming that the two time intervals overlap.

The following query shows how to use an INTER operator to retrieve time-stamp values:

Query. List the salary and car history of all university-staff while their salaries were less than 35K.

```
SELECT   University-staff.sname, salary, Car-owner-
           ship.cname
         (University-staff INTER Car-owner-
           ship).TIME-START
         (University-staff INTER Car-owner-
           ship).TIME-END
FROM     University-staff, Car-ownership
WHERE    (University-staff.sname = Car-owner-
           ship.sname) AND
         (University-staff.salary < 35K)
WHEN     University-staff.INTERVAL OVERLAP Car-
           ownership.INTERVAL
```

ACTIVE DATABASE LANGUAGES

Conventional database systems are passive. In other words, data are created, retrieved, and deleted only in response to operations issued by the user or from the application programs. Proposals have been made to transform database systems to active. This means that the database system itself performs certain operations automatically in response to certain events or conditions that must be satisfied by every database state. Typically an *active database* supports (1) the specification and monitoring of general integrity constraints, (2) flexible timing of constraint verification, and (3) automatic execution of actions to repair a constraint violation without aborting a transaction.

A major construct in active database systems is the notion of event condition action (ECA) rules. An active database rule is triggered when its associated event occurs; in the meantime the rule's condition is checked and, if the condition is true, its action is executed. An event specifies what causes the rule to be triggered. Typically triggering events include data modifications (i.e., SQL INSERT, DELETE, or UPDATE), data retrievals (i.e., SELECT), and user-defined statements; the condition part of an ECA rule is a WHERE clause, and an action could be a data modification, data retrieval, or a call to a procedure in an application program. The following SQL-like statement illustrates the use of an ECA rule:

```
<EVENT>:       UPDATE University-staff
               SET Salary := Salary * 1.1
<CONDITION>:   Salary > 1000K
<ACTION>:      INSERT INTO Highly-Paid-University-
                 Staff
```

Several commercial (relational) database systems support some restricted form of active database rules, which are usually referred to as *triggers.* In SQL3, each trigger reacts to a specific data modification operation on a table. The general form of a trigger definition is as follows (8):

```
<SQL3 trigger> ::= CREATE TRIGGER <trigger name>
                   {BEFORE|AFTER|INSTEAD OF} <trig-
                     ger event>
                   ON <table name>
                   WHEN <condition>
                   <SQL procedure statements>
                   [FOR EACH {ROW|STATEMENT}]
<trigger event> ::= INSERT|DELETE|UPDATE
```

where *<trigger event>* is a monitored database operation, *<condition>* is an arbitrary SQL predicate, and *<action>* is a sequence of SQL procedural statements which are serially executed. As shown, a trigger may be executed BEFORE, AFTER, or INSTEAD OF the associated event, where the unit of data that can be processed by a trigger may be a tuple or a transaction. A trigger can execute FOR EACH ROW (i.e., each modified tuple) or FOR EACH STATEMENT (i.e., an entire SQL statement).

An *integrity constraint* can be considered as a special form of trigger whose action is to issue an error message when some conditions are violated. SQL-92 allows integrity constraints to be specified in some restricted forms. Table constraints are used to enforce permissible values on the domain of a particular attribute of a relation. Typical examples of such constraints are nonnull values (NOT NULL) and nonredundant values (UNIQUE). These constraints are defined as a part of the CREATE TABLE statement. A *referential integrity constraint* specifies that a tuple in one table (called the referencing table) referencing another table (called the referenced table) must reference an existing tuple in that table. They are specified in terms of a FOREIGN KEY clause in the referencing table, which states that if a delete or update on the referenced relation violates the constraint, then (instead of rejecting the operation) some action is taken to change the tuple in the referencing relation in order to repair the constraint violation. Consider the following example: If an update of a tuple in the referenced relation 'Department' violates the referential constraint, then the attribute 'dept-name' in the referencing tuple is also updated to the new value.

CREATE TABLE University-personnel
  person-name: char(9),
  dept-name: char(20)
  FOREIGN KEY (dept-name) REFERENCES Department
    ON DELETE CASCADE
    ON UPDATE CASCADE

An integrity constraint may also be an arbitrary user-defined SQL predicate. There are several ways to evaluate an integrity constraint. Immediate evaluation allows an integrity constraint to be checked after every SQL statement which may violate the constraint is executed. In deferred evaluation, constraint checking is not performed until a transaction commits. Usually system-defined constraints (i.e., table or referential constraints) are evaluated immediately, and general assertions are evaluated in the deferred mode.

DEDUCTIVE DATABASE LANGUAGES

A deductive database extends the relational data model to support deductive reasoning via a *deductive* (or *inference*) *mechanism* that can deduce new facts from the database rules. It consists of two main types of specifications: facts and rules. *Facts* are similar to tuples in a relation, and *rules* are similar to relational views. They specify virtual relations that are not actually stored but can be derived from facts. The main difference between rules and views is that rules may involve recursion, which cannot be defined in the relational model. In general, a rule is a conditional statement of the form *if <condition> then <deduced relation>*.

Integrating logical deduction with a database system requires the development of a rule language. DATALOG is a declarative query language that can be used to facilitate set-oriented database processing. It is based on the logic programming language PROLOG. The syntax of DATALOG is similar to that of PROLOG. However, a major difference between DATALOG and PROLOG is that a DATALOG program is defined in a purely declarative manner, unlike the more procedural semantics of PROLOG. Therefore DATALOG is a simplified version of PROLOG.

DATALOG Rules

An atom (or positive literal) has the form $P(t1, t2, \ldots, tn)$ where P is a predicate and $t1, t2, \ldots, tn$ are either variables or constants. Similarly a negative literal has the form *NOT* $P(t1, t2, \ldots, tn)$. A ground atom (or fact) is an atom containing only constants. A rule is presented as $P :- Q1, Q2, \ldots, Qn$, where P is an atom built with a relation predicate and Qi's are atoms built with any predicate. This form of a rule is called a *Horn clause,* where P is called the rule head (or conclusion) and $Q1, Q2, \ldots, Qn$ are called the rule body (or premises). Following are some examples of facts and rules:

Facts: *(1) Parent(Mary,Tom)*
(2) Parent(John,Mary)
(3) Parent(Mary,Ann)
Rules: *(1) Ancestor(x,y) :− Parent(x,y)*
(2) Ancestor(x,y) :− Parent(x,z), Ancestor(z,y)
(3) Sibling(x,y) :− Parent(z,x), Parent(z,y)

As shown in the examples, there are two predicates: *Parent* and *Ancestor.* An ancestor is defined via a set of facts, each of which means *'X is a parent of Y'.* These facts correspond to a set of tuples stored in the relation *'Parent'.* Rule 3 is an example of recursive rules, where one of the rule body predicates is the same as the rule head. A DATALOG program is a set of rules as exemplified.

A rule is instantiated by replacing each variable in the rule by some constant. A rule simply states that if all the body predicates are true, the head predicate is also true. Thus a rule provides us a way of deriving new facts that are instantiations of the head of the rule. These new facts are based on facts that already exist. In other words, the rule body specifies a number of premises such that if they are all true, we can deduce that the conclusion is also true. As an example, suppose that in rule 3, variable z was replaced by 'Mary', variable x by 'Tom', and variable by 'Ann'. Since the facts corresponding to *Parent(Mary, Tom)* and *Parent(Mary, Ann),* we can deduce a new fact *Sibling (Tom, Ann)* from rule 3.

In DATALOG, a query is specified by a predicate symbol with some variables; this means to deduce the different combinations of constant values that can make the predicate true. In the above example, a query *'Find all descendants of John*?' can be expressed as *Ancestor(John, x)* whose answer set is {*Mary, Tom, Ann*}.

In a deductive database, a *model* of a set of rules is defined to be a set of facts that makes those rules true. An interesting point of a DATALOG program is that the intersection of a set of models is also a model. Thus any DATALOG program has a unique *least model.* The procedure to compute a minimal model of a DATALOG starts with a set of given facts I. While the rule body can be instantiated with the facts in I, the fact corresponding to the instantiated rule head is generated and added to I. When no new elements can be added to I, this is the minimal model. For example, consider the following DATALOG program (3):

{*Parent(Sue,Pam), Parent(Pam,Jim), Ancestor(x,y) :− Parent(x,y)*}

We can compute the following:

I1 = {*Parent(Sue,Pam), Parent(Pam,Jim)*}
I2 = *I2 union* {*Ancestor(Sue,Pam)*}
I3 = *I3 union* {*Ancestor(Pam,Jim)*}

At this point, since no new fact can be generated, *I3* is the least model.

One problem associated with DATALOG is to guarantee the answer set is finite. A rule is called *safe* if it generates a finite set of facts. It is possible to specify rules that generate an infinite number of facts. Following is a typical example of unsafe rules:

High_Temperature(y) :− y > 100

In this example some unsafe situations can be identified. Specifically, a variable in the body predicate can have an infinite number of possible instantiations. It can be shown that it is not solvable to determine whether a set of rules is safe. However, a syntactic structure of safe rules has been proposed based on the notion of range restricted rules. A rule is *range restricted* if all variables of the rule's head appear in a nonnegated relational predicate in the rule body.

Extension of DATALOG

To increase the expressive power of DATALOG, several extensions of DATALOG have been proposed (3). DATA(fun) extends DATALOG with functions that may be invoked in the rule body. DATALOG(neg) extends DATALOG with the use of negative literals in the rule body. Thus we can generalize the basic rule definition as follows: *P :− Q1*, *Q2*, . . ., *Qn*, where *Qi*'s are positive or negative literals built with any predicate. The semantics of DATALOG(neg) is not easy to define because the program may not have a least model. For instance, the following program has two models: {*Bird(Tiger), Bat(Tiger)*} and {*Bird(Tiger), Has-Egg(Tiger)*}. The intersection of these models is not a model of this program:

{*Bird(Tiger), Has-Egg(x) :− Bird(x), NOT Bat(x)*}

One important extension of DATALOG(neg) is stratified DATALOG. A program is *stratified* if there is no recursion through negation. For instance, the following example is not stratified because it involves a recursion via negative literals (3):

{*P(x) :− NOT P(x), Q(x) :− NOT P(x), P(x) :− NOT Q(x)*}

Stratified programs have a least model that can be computed in an efficient way.

Recursion

Recursive rules are useful to express complex knowledge concisely. The concept of recursion in DATALOG is similar to that of general programming languages. A typical type of recusion is transitive closure such as ancestor-parent, supervisor-employee, or part-subpart relationship. A rule is *linearly recursive* if the recursive predicate appears only once in the rule body. Notice that rule 2 below is not linearly recursive. It is known that most application rules are linear recursive rules; algorithms have been developed to execute linear recursive rules efficiently.

(1) *Ancestor(x,y) :− Parent(x,z), Ancestor(z,y)*
(2) *Ancestor(x,y) :− Ancestor(x,z), Ancestor(z,y)*

Indeed there exist database queries that cannot be answered without using recursion. Consider a query "*Retrieve all supervisors of the employee John*". Although it is possible to retrieve John's supervisors at each level, we cannot know the maximum number of levels in advance. An alternative to recursion is to embed SQL into a general programming language and iterate on a nonrecursive query, which in effect implements the fixed point process. However, writing such queries is much more complicated than using recursive rules.

Given a DATALOG program, the :− symbol may be replaced by an equality symbol to form DATALOG equations. A *fixed point* of a set of equations with respect to a set of *relations* $R1$, $R2$, . . ., Rn is a solution for these relations that satisfies the given equation. A fixed point then forms a model of the rules. For a given set of equations it is possible to have two solution sets $S1 <= S2$, where each predicate in $S1$ is a subset of the corresponding predicate in $S2$. A solution $S0$ is called the least fixed point if $S0 <= S$ for any solution S satisfying those equations. Thus a least fixed point corresponds to a least model. We also note that the existence of fixed point always guarantees the termination of a program.

ODBC

Open Database Connectivity (ODBC) (4) is an application program interface for multiple database accesses. It is based on the call level interface (CLI) specifications and uses SQL as its database access language. ODBC is designed for maximum interoperability, that is, the ability of a single application to access heterogeneous databases with the same source code. The architecture of ODBC consists of three layers to provide transparency: an application program calls ODBC functions to submit SQL statements and retrieve the results; the Driver Manager processes ODBC function calls or pass them to a driver; and the driver processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application.

CONCLUSION

We have considered several modern database languages based on their underlying data models. Although SQL has been widely accepted as the standard query language, it requires additional features such as complex data types, temporal data, trigger, and deduction to support advanced applications. A comparison of the database languages discussed in this article is summarized in Table 2.

Table 2. Comparison of Database Languages

	Relational	Object Relational	Object-Oriented	Temporal Relational	Active	Deductive
Structure	Flat table	Nested table	Class	Table with time	Table (with trigger)	Rule
Query type	Declarative	Declarative	Procedural, declarative	Declarative	Declarative	Declarative
Language	SQL	SQL3	Persistent C++	TSQL	SQL3	DATALOG
Optimization	System	System	User	System	System	System

BIBLIOGRAPHY

1. M. P. Atkinson and O. P. Buneman, Types and persistence in database programming languages, *ACM Comput. Surveys,* **19** (2): 105–190, 1987.
2. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems,* Menlo Park, CA: Benjamin/Cummings, 1994.
3. G. Gardarin and P. Valduriez, *Relational Databases and Knowledge Bases* Reading, MA: Addison-Wesley, 1989.
4. *Microsoft ODBC 3.0: Programmer's Reference,* vol. 1, Redmond, WA: Microsoft Press, 1997.
5. K. Parsaye et al., *Intelligent Databases: Object-Oriented, Deductive Hypermedia Technologies,* New York: Wiley, 1989.
6. A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts,* New York: McGraw-Hill, 1996.
7. A. Tansel et al., *Temporal Databases,* Menlo Park, CA: Benjamin/Cummings, 1993.
8. J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules For Advanced Database Processing,* San Mateo, CA: Morgan Kaufmann, 1996.

UNG MO KIM
Sung Kyun Kwan University
PHILLIP C-Y SHEU
University of California

DATABASE MACHINES. See PARALLEL DATABASE SYSTEMS.

DATABASE MANAGEMENT SYSTEMS. See DATABASES; VERY LARGE DATABASES.