Database Fundamentals

Byunggu Yu, PhD

Computer Science and Information Technology
University of the District of Columbia
Bldg. 42, Suite 112
4200 Connecticut Ave. NW
Washington, DC 20008
Phone: (202) 274-6289
byu@udc.edu

This article is primarily targeting people who have some knowledge in Computer Science or Information Technology or those who meet the prerequisites for a senior database course. Also targeted are those who have some database experiences but need to refresh/expand their database knowledge or those who need concise references.

Original submission on March 22, 2008
First Revision on Aug 10, 2008
Second Revision on December 5, 2008
Final Revision on Feb 10, 2009

Outline

Key Words: databases, database design, database system, spatiotemporal databases

Database Fundamentals

Abstract

From personal email clients to enterprise and government information systems, modern computer applications deal with increasingly large amount of data. Accordingly, there is an ever increasing demand for efficient and reliable technology for managing large collections of structured data (databases). This article provides readers with a comprehensive overview of modern database technology. Practical database development methods are presented along with necessary details of related concepts and theory. The following topics are covered: definition of modern database systems, standard database models, database design and development, performance and scalability issues, and emerging application concepts for managing continuously moving or changing data.

1. Introduction

In today's computing world, more and more applications -- ranging from personal programs, such as personal file managers and email clients, to enterprise and government server applications -- deal with increasingly large data.

A *database* is a collection of structured and organized data. Examples include a set of messages in a certain form (e.g., an email form of sender, recipient, subject, and message body fields), a set of student records, and a set of medical records with associated patient and insurance data, to name a few. To efficiently and reliably manage large databases, carefully designed computer application software is necessary. This software is called a *database management system* (DBMS).

A *database system* is a complete computer application system consisting of databases, a database management system (DBMS), application programs interacting with the DBMS, and human users interacting with the application programs. For efficiency and reliability, modern databases are systematically structured and organized in standard database models.

Modern database technology has been developed through decades of rapid evolution -- electronic databases have been in use since the advent of electronic computer systems equipped with some storage for programs and data.

Modern general-purpose computers (beginning with DEC's PDP series and IBM 360) evolved during the mid-1960s through the early-1980s. This is an important era of modern electronic computing: computer systems began to be widely distributed with more storage space and computing capabilities, supporting both data-intensive and computation-intensive applications. Accordingly, important modern computing concepts, such as general-purpose operating systems (UNIX), multiprogramming, random-access storage, spooling, and timesharing, were developed through this period. These key concepts and technologies provided a strong basis for the current generation of multitasking computers.

As the computers grew in popularity and capability, more data-intensive applications (or database applications) required reliable, robust, and secure mechanisms for managing and sharing large collections of complicated data. Early custom and ad hoc database management approaches were found to be easily defective due to several difficulties.

For example, suppose that one developed a student database application originally for an academic department at a university. If the program manages student data, such as names, majors, identification numbers, addresses, advisors, telephone numbers, and email addresses, in custom-made files, it will certainly be challenging for any others developing another student database application for the university's registrar's office to access the data (*data sharing*) and to manage the registrar's own portion of the data in such shared files in a protected and secure manner (*data protection and security*). Should the two applications end up with two different custom-made data sets (files), the department's students must be stored in both data sets (*data redundancy*). Then any changes on the student side must immediately be reflected on both databases, otherwise, this data redundancy results in *data inconsistency* (e.g., two different email addresses of a student: one is up-to-date and the other is old and not valid).

These challenges gave birth to modern database standards.

This article provides readers with a comprehensive overview of modern database technology. Standard database development methods are presented along with necessary details of related concepts and theory.

The organization of the rest of this article is as follows: Section 2 gives a brief definition of modern database systems; Section 3 introduces major standard models for database expression and sharing; Section 4 provides a good database development practice, addressing data redundancy, inconsistency, protection, and security; Section 5 gives a review on the ubiquitous SQL, a practical standard database language for database implementation and use; Sections 6 and 7 address the performance and scalability issues; Section 8 provides some current and future trends in the area of database research and practice.

2. What is a Database System?

A *database system* is a computer application system consisting of one or more databases (collections of structured and organized data), a database management system (DBMS) managing the databases, one or more database application programs interacting with the DBMS to access the databases, and human users or other applications that interact with the database application programs. (Physical database storage is outside the scope of this article. Interested readers are referred to [Kim, Yu, and Chang, 2008].)

All database application programs must indirectly access the databases only through the DBMS (by calling the data access and management functions of the DBMS). This is for ensuring performance optimization, protection, and security of the system and its data.

Typically, database system users are categorized into end users, application developers/programmers, database management system developers/programmers, and database administrators. Modern DBMSs recognize such different user types in order to support different roles of the user groups for enhanced protection and security.

Figure 1 shows a modern database system. An instance of a database system consists of the databases as well as the runtime instances (processes; programs in execution) of the DBMS and application programs.

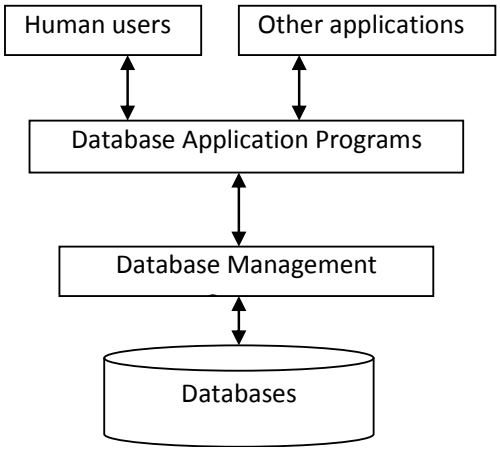


Figure 1. Database system architecture

A particular method used for structuring or organizing data within a database is called a database model. A database model specifies (1) a set of structure types, (2) the associated operations, (3) a set of rules according to which consistent database states and changes are defined and maintained. A database management system (DBMS) is built on a particular database model. Popular database models and recently reviving legacy database models (through object-oriented concepts) are introduced in Section 3.

3. Database Models

During the earliest period of electronic computing and programming, ad hoc style custom-made databases were created and managed by individual application programs. Standard database models and general purpose database management systems were nonexistent.

In the 1960s, the capability and popularity of electronic computers increased along with more applications dealing with large amount of data, demanding a general purpose database model that can provide a basis for data sharing and consistency control mechanisms as exemplified in Section 1.

To meet this growing demand for a standard database model, Charles Bachman and his colleagues formed Database Task Group within CODASYL (Conference on Database Systems Languages) to develop a database model, based on Bachman's IDS (Integrated Data Store), an early database engine developed at General Electric in the early 1960s. In 1969, CODASYL developed the first version. In 1971, CODASYL delivered a second publication, which became the model basis for some early DBMSs, such as IDMS. This modeling effort continued into the 1980s. This database model is commonly called the "Network Model" [CODASYL, 1971] and databases based on this model are called network databases. The Network Model is explained in Section 3.1.

Although the CODASYL's standardization effort and its final ISO specification produced substantial scientific impact, its popularity in commercial database software industry was not dominant. In the late-1960s, IBM used the "Hierarchical Model" [Bjorner and Lovengreen, 1982], which was a basis for the development of IMS (IBM Information Management System) and DL/I (Data Language 1), used in the Apollo program in part. IBM's IMS and its transaction manager are still in active use in the financial industry [IBM 2008]. The Hierarchical Model is explained in Section 3.2.

Although not the mainstream database model, the concepts of the Hierarchical Model and the Network Model (now also collectively called the navigational database models) have been applied in many data and information systems including knowledge information systems, XML/XPath, DOM (Document Object Model), and Web services, where procedural data navigation operations [Bachman, 1973] are dominant.

Another model, which became the mainstream model for most modern disk-based, general-purpose DBMSs, such as Oracle, DB2, Informix, MySQL, and SQL-Server, is the "Relational Model". The Relational Model was originally proposed by E.F. Codd in 1970 [Codd, 1970]. The Relational Model and the object-oriented concepts are discussed in Sections 3.3 and 3.4, respectively.

The following two generic definitions might help some readers:

Set: a *set* is a collection of objects of the same type. For example, a student set of id, name, major, and age is a collection of student objects (representations) each of which has a certain value on each of the attributes (i.e., id, name, major, and age).

Mapping Cardinality (*n-to-m relationship*): when there is an *n-to-m relationship* (or a *mapping cardinality of n-to-m*) between two sets S1 and S2, both of the following conditions hold: (1) an object in S1 can be associated with (or linked to) m objects in S2; (2) an object in S2 can be associated with (or linked to) n objects in S1. Note, n and m can be any positive integers. When n and m are not specified, it means that they can be any positive integers.

3.1 The Network Model

In the Network Model [Bachman 1973; CODASYL 1971; Silberschatz et al., 2005a], a database consists of records and sets. A record represents a concrete entity in the real world. Every record is an instance of a certain record type and has a unique database key (DBK). A record type is a named definition of all permissible occurrences of records of a certain type. A record type consists of named items (fields) that are either element items or a named group of other items. Figure 2 gives an example of a record type and two records.

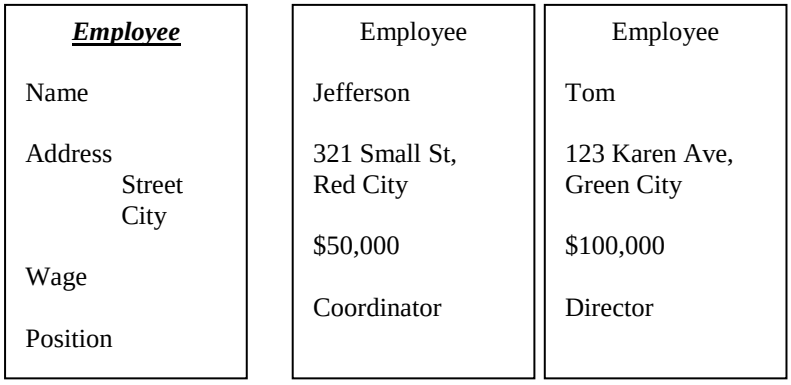


Figure 2. Record type and records

The main structuring concept of the CODASYL Network Model is the set (or data set), which is a one-to-many relationship between two record types. Each set is named and includes one record of the first record type (Owner) and 0, 1, or more records of the other record type (Members). It is allowed for a record type to take the roles of both owner and member in a set. Each set is ordered (i.e., ordered set of records), and the sequence of the records can convey some information.

In contrast to the Hierarchical Model, which is explained in Section 3.2, the Network Model allows records to participate in multiple sets. For example, if a record participates in two sets with two different owners, the record has two owners, breaking the tree-like hierarchy where every node (record) has strictly only one parent (or owner) except for the root. This results in more general (less restrictive) network structures.

The data-structure (database schema) diagram of CODASYL DBTG (Database Task Group) 1971 standard includes record type node, many-to-one link, and link node (a.k.a. Rlink). Rlink is a non-data node that may have a single field of system generated unique identifier. This was introduced to represent many-to-many relationships and n-ary relationships. A network node is

represented as a box; a link is represented as a line with an arrow pointing to the one side of the relationship. Figure 3 shows an example of a Network Data-Structure Diagram (a simple database schema in the Network Model): The Rlink represents the many-to-many relationship between Customer and Account using two many-to-one links.

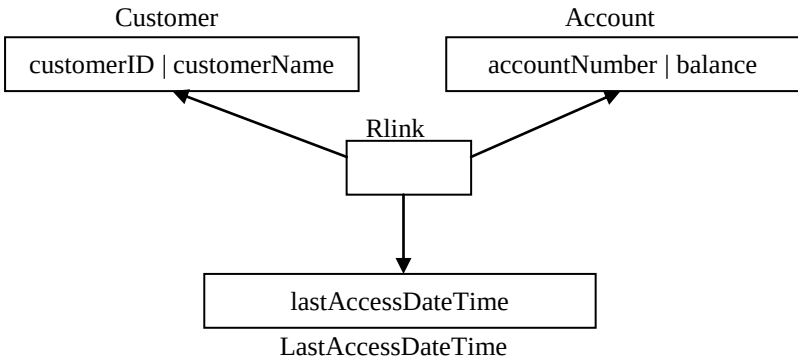


Figure 3. Network-structure diagram

In most early DBMSs based on the Network Model, database keys are physical addresses of the records in the storage. Moreover, each set is implemented in the form of linked list or tree in the storage. These closely tied logical data description and physical data description contributed to the high efficiency of data retrieval, but also made some maintenance and restructuring operations rather expensive.

3.2 The Hierarchical Model

IBM developed a DBMS system, IMS (Information Management System) [IBM, 2008], based on their implementation of the Hierarchical Model and DL/I (Data Language 1). The Hierarchical Model [Bjorner and Lovengreen, 1982] is generally similar to the Network Model, except that no record can have more than one parent in the structure, which is the main conceptual difference. One can represent a network of entities by duplicating entities. However, this duplication (redundant data) can cause data inconsistency problem. Thus, there has been a claim in favor of the Network Model, in comparison to the Hierarchical Model: the Network Model allows a more natural modeling of relationships between real entities in knowledge representation.

Like the Network Model, the tree-structure diagram (database schema diagram) of the Hierarchical Model consists of nodes and arrowed lines. However, a tree-structure diagram must be organized in the form of a rooted tree. Figure 4 shows a tree-structure diagram corresponding to the schema shown in Figure 3.

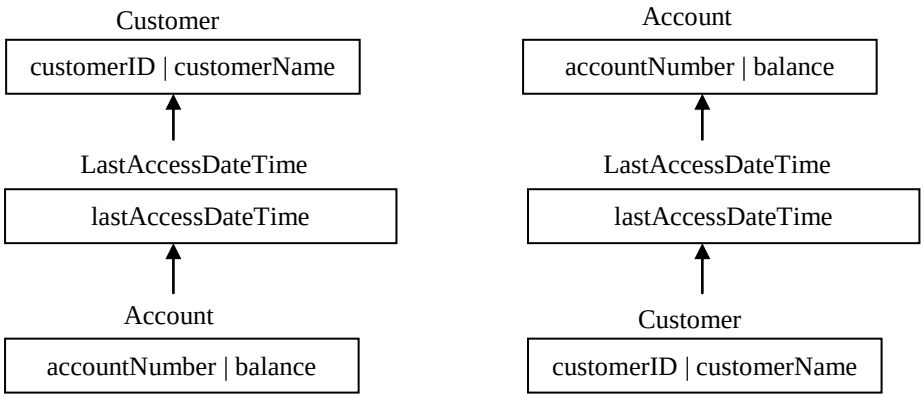


Figure 4. Tree-structure diagram

In the IBM Hierarchical Model, a database is a tree (hierarchy) of nodes called segment, which is the smallest retrieval unit. Segment is similar to the concept of record in the Network Model and consists of one or more fields (items in the Network Model). IMS stores segment trees on IBM's hierarchical file systems, such as ISAM, VSAM, HSAM (Hierarchical Sequential Access Method), and HDAM (Hierarchical Direct Access Method).

IMS has transaction management capabilities called IMS TM typically supporting IMS network databases as well as IBM's relational databases (DB2 introduced in 1982).

3.3 The Relational Model

The Relational Model for database management was proposed by Edgar Codd in 1969 and 1970 [Codd, 1972]. This model enables the application programs and users to express database queries and integrity constraints by describing "what" they want, without "how" to navigate the database to achieve it.

For example, to retrieve every employee whose age is less than 30, instead of writing a procedure of navigation and selection (or find-and-get) steps, one can simply say "report every employee *e* whose age is less than 30" or $\{e \mid \exists_{e \in \text{employee}} (e.age < 30)\}$.

Informally speaking, the term "relation" and "relational" respectively mean "table" and "based on tables": a database is a set of tables. Normally, in a relational database, every table is linked to one or more other tables in the same database. A table can also be linked to itself. Unlike some early implementations of the Network Model and the Hierarchical Model, the Relational Model completely separates a "logical database" from its physical implementation (the actual database represented in the platform's physical storage) by providing a complete basis for a logical representation of the database. This basis can also support the applications' different and secured "views" of the same database.

For example, considering a university database, an academic department's view, the registrar's view, and the research office's view are all different, but each of these represents a customized (and possibly reorganized) subset of the whole database that is systematically structured, shared, and managed for consistency. This multi-level data independence provides much security, flexibility, accessibility, and database migration facilities. This advantage increased the popularity of the Relational Model in many application domains. Historically, this data independency has provided the database community with an open opportunity to contribute to the technology development by devising efficient physical database implementations and

mappings between the different levels (physical data organizations, indexing methods, and query processing techniques), increasing its popularity in relevant research communities.

Importantly, the mathematical foundation of the Relational Model was implemented and has been matured, resulting in the dominant mainstream database products and applications, collectively called SQL (Structured Query Language) database systems.

SQL query capability, which is largely based on the Relational Model, is currently the mainstream database technology. Although still far from perfection [Date and Darwen, 2000], this has been tried and found to be reliable, robust, and flexible enough to support ad hoc queries as well as predefined queries to some good extent.

Traditionally, the relational database approach has used Entity-Relationship (ER) Model [Chen 1976] to represent a database schema diagram. Figure 5 shows a basic example corresponding to the diagrams shown in Figures 3 and 4.

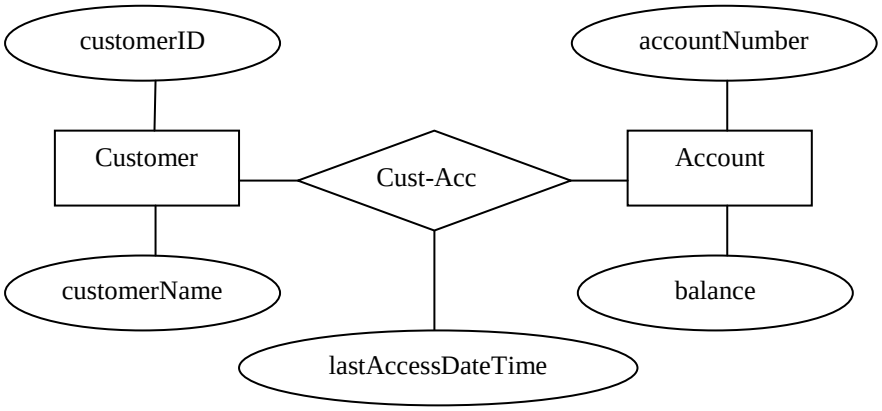


Figure 5. Entity-Relationship diagram

This model and more recently developed alternatives are extensively discussed along with the details of the Relational Database Model in Section 4.

3.4 The Object-oriented Model and Native Query

The Object-oriented Model [Date and Darwen 2000; Silberschatz et al., 2005c] and its implementations became major issues over the 1980s and the 1990s as the object-oriented programming languages became popular.

Object-oriented databases (or simple object databases) are primarily designed to work with object-oriented programming languages such as C++, Java, C#, Python, and some proprietary languages. ODBMSs (Object or Object-oriented Database Management Systems) use the exact same object-oriented model (including type system, object encapsulation, inheritance, composition) as the supported object-oriented language(s).

Consequently, the database schema representation of the Object-oriented Model adopted UML (Unified Modeling Language) that is intended, in part, to unify past experience about object-oriented software modeling techniques. UML is a large modeling language consisting of at least 11 diagrams. Particularly, Class Diagram, which represents the static structural view of object-oriented software, is popular in object-oriented database schema design.

In Class Diagram, the main constituents are “class” and “relationship”. Class represents a set of the same type objects (instances). A class has three components – name, data attributes, and operations. Note that, unlike the other models, the Object-oriented Model can represent the operations (the static behavior aspects) of data objects. As in the modern ER model (discussed in Section 4.1), the relationship is enhanced by various constraints and further categorized into association, generalization, realization, and various kinds of dependency. Figure 6 shows a simple example of UML Class Diagram corresponding to the ER diagram shown in Figure 5. Interested readers are referred to [Rumbaugh et al., 2005].

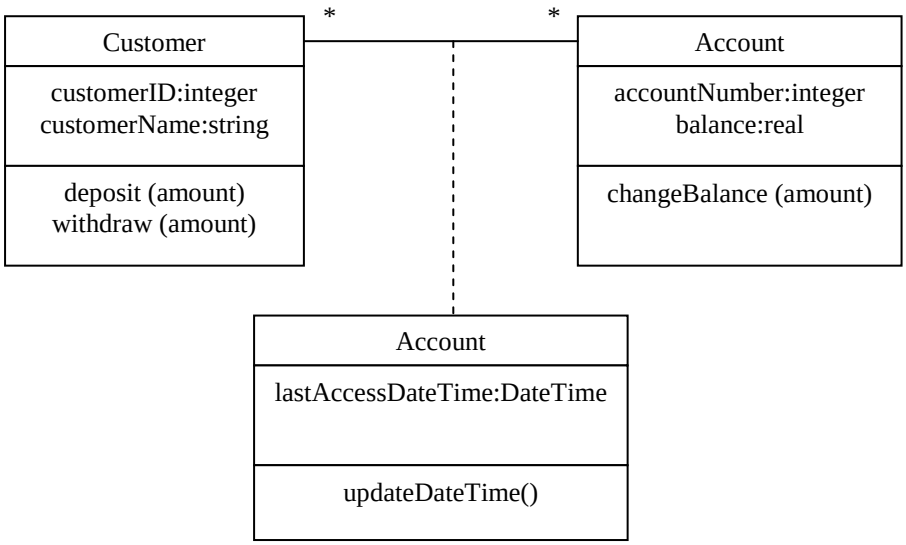


Figure 6. UML class diagram

Well-developed ODBMSs provide their supported object-oriented programming languages with core database capabilities including persistent object storage, consistency control mechanisms for concurrent accesses to the objects, transaction and recovery, navigational and declarative queries, and other database capabilities.

During the late 1990s and the early 2000s, object-oriented database standards for Java (ODMG Java Language Binding), which is part of ODMG (Object Data Management Group) standard 3.0, became popular. In 2001, many members of the ODMG decided to focus only on the Java Data Objects specification, and the ODMG disbanded.

More recently, in 2006, the Object Management Group (OMG) formed Object Database Technology Working Group to develop a standard object databases specification based on ODMG3.0 and other recent developments in the area of object databases.

Object databases typically show the best performance for predefined (defined and programmed when the database was designed) navigational queries using the pointers connecting the objects. Because of this, as the network and the Hierarchical Models, the Object-oriented Model is categorized into the navigational models. However, recently developed ODBMSs, such as Objectivity/SQL++, Cache, and Matisse, also support SQL (Structured Query Language originally developed based on the mathematical foundation of the Relational Model) to some limited extent.

Although most object-oriented database management technologies (products) are based on the traditional C++ object oriented concepts and/or the Java part of the ODMG 3.0, the standard basis for pure object-oriented databases is relatively weak, compared to the strong basis of the relational databases. Unlike the Relational Model, the object model has a lack of mathematical foundation. Because of this, pure object-oriented query language standardization efforts have not resulted in a complete standard. Recently, an alternative solution has been investigated: instead of introducing an additional database language, one can use the object-oriented programming language itself to express queries in application programs (a.k.a. the native queries). Examples include Microsoft's Language Integrated Query, a database query extension to C# and .NET.

Switching from an SQL DBMS to a purely object-oriented DBMS means you lose the capability to create sound and complete SQL queries and its mathematical basis for retrieving ad-hoc combinations of data, even though many newer commercial object-oriented databases are able to process SQL queries to a limited extent.

Much of the object database ideas were also adopted in recent SQL standards [ISO 1999; ISO 2003], which is currently supported by mainstream relational DBMSs (now called object-relational DBMSs).

4. Database Design: The Three-Step Approach

This chapter provides a guided tour in developing an application system based on a mainstream SQL DBMS. It is recommended that interested readers apply the techniques introduced in this chapter to other object-oriented database management systems, such as CACHE, that support both pure object-oriented concepts as well as the mathematical Relational Model basis. For application-specific pure object-oriented applications, since database objects are managed as program objects and since database queries and navigations are expressed in the native programming languages, interested readers are recommended to follow the well-established and up-to-date Unified Modeling Language (UML) approach as well as a specific object-oriented database management product's specifications.

Given a database application requirement (written verbally in natural human language) and a DBMS, one can design a database schema (structure) and pre-determined queries to build a database system. This section (Section 4) introduces a professional database schema design techniques. Database query and some logical database optimization steps are discussed later.

4.1 Step 1: Application Requirements to Conceptual Design

The first step of the database design process is reaching an agreement with the requirement issuers on the interpretation of the original requirement and necessary assumptions. For this, a visual language that can visually and intuitively express the concepts/views of the database designer is required for systematic interpretation of the original requirement. The primary goals of this conceptual design process are finding and clarifying all potential ambiguities and necessary assumptions with the customers.

Popular languages for this purpose include UML (Unified Modeling Language) and the ER-Model [Chen 1976]. This article uses an extended ER-model [Lyngbaek and Vianu, 1987; Markowitz and Shoshani, 1992], which supports some object concepts, such as inheritance and composition hierarchies. Interested readers are recommended to consider UML for pure object-

oriented database design purposes in order to model not only the structures but also the behaviors of database objects in a supported object-oriented programming language.

The key concepts of the ER-model are introduced through Sections 4.1.1 – 4.1.5. Then Section 4.1.6 gives an example of converting an application requirement into an ER-diagram, a conceptual database schema. Section 4.1.7 gives section closing remarks.

4.1.1 Entity Set

An entity is a concrete object (e.g., a person or a book) or an abstract object (e.g., a checking account, a holiday, or a concept) in the real world that can be uniquely identified.

An entity set is a set of entities of the same type that have the same properties (attributes). In the ER-model, an entity set is represented as a rectangle and each of its attributes is represented as an ellipse connected to the rectangle. Figure 7 shows an example of ER-diagram consisting of only one entity set.

One or more attributes of each entity set is underlined -- Collectively, the underlined attributes of an entity set form the *primary key* of the set. The primary key is the principle means of identifying entities within the set (i.e., every entity has a unique not-null value on the key). A dashed ellipse represents a *derived attribute*. The value "AVAILABLE_CREDIT" in Figure 7 can be derived (i.e., CREDITLINE – CURRENT_BALANCE). Derived attributes need not be stored in the database, but can be computed at run-time in the application.

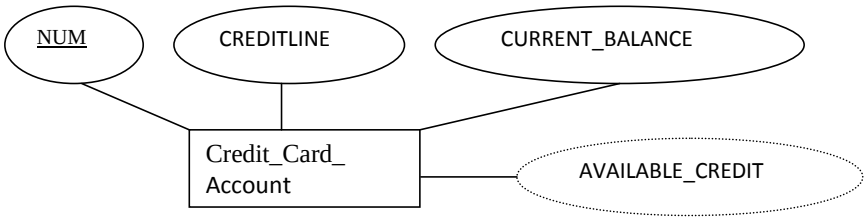


Figure 7. ER diagram of a single entity set

4.1.2 Relationship Set

A relationship is an association among several entities. An *n*-ary relationship *r* is an ordered *n*-tuple (e_1, e_2, …, e_n), where e_i is a member of entity set E_i for $1 <= i <= n$. This is read as "the entities e_1, e_2, …,e_n participate in relationship *r*".

A relationship set is a set of relationships of the same type that have the same properties. An *n*-ary relationship set R is {(e_1, e_2, …, e_n), *r-att* | e_1 is a member of E_1, e_2 is a member of E_2, …, e_n is a member of E_n, *r-att* is a set of zero, one, or more attributes the participating entities share}.

In the ER-model, a relationship set is represented as a rhombus (informally called diamonds) linked to participating entity sets (rectangles). Each of these links needs a careful and thoughtful attention, and this process is one of the keys for finding potential ambiguities and necessary assumptions that must be clarified by the issuer of the original application requirement.

Figure 8 shows an example ER-diagram of binary relationship. In our version of the ER-model used in this article, each linear line connecting a participating rectangle to the rhombus has one

or more integer ranges on it. In the case of binary relationship (i.e., a rhombus with two connected rectangles), a range represents the minimum and the maximum number of entities in the corresponding rectangle (entity set) that can be associated with one entity of the other rectangle on the other side of the rhombus. Similarly, in the case of a rhombus connected to N rectangles (N-ary relationship), each range represents how many (the minimum and the maximum number of) entities of the corresponding rectangle (entity set) can be associated with a certain set of entities from all the other participating rectangles. Although the original notation for mapping/relationship cardinalities (multiplicities) employs more visual elements, such as double line and arrow, this article employs a modification based on integer ranges. For example, Figure 8 says that 1 to 5 departments can be associated with a certain employee; the diagram also says that 1 – 10 employees can be associated with a certain department (i.e., each department must have at least one and up to 10 employees); Each employee has a certain "position" in each department he/she works for.

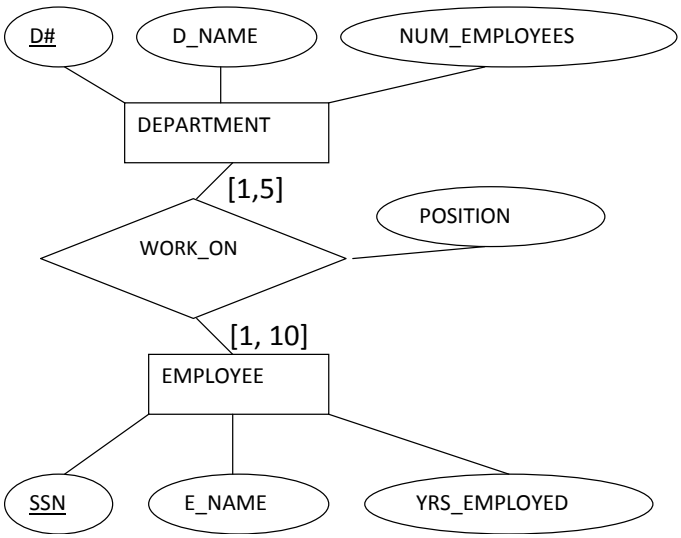


Figure 8. An example of binary relationship

4.1.3 Weak Entity: A Concept of Nested Entities

Thus far, we have discussed only strong entity sets. Every strong entity has a valid primary key and the existence of a strong entity is not dependent on the existence of any other entity. If an entity set has no primary key, i.e. one cannot uniquely identify the entities, the entity set is considered to be a weak entity set. Each weak entity requires a dominating (owner) entity to exist. A weak entity set can be an owner of another weak entity set, resulting in a tree of weak entity sets. However, the root of the tree must be a strong (normal) entity set. Figure 9 shows an example of weak entity set.

The relationship between a weak entity set and the dominator must be "many-to-one" (i.e., each weak entity must be associated with exactly one and only one owner entity: "1" on the owner side relationship line). If it is "one-to-one" every attributes of the weak entity set will be placed in the dominator as its attributes and the weak entity can be safely eliminated.

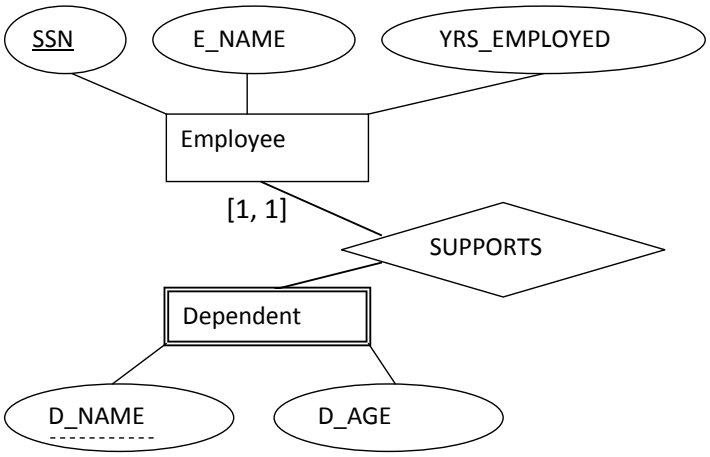


Figure 9. An example of weak entity set

The weak entity set has a "discriminator" (or weak key), instead of a primary key. Theoretically, one can uniquely identify each entity in the weak entity set with {*dk*, *wd*}, where *dk* is the key of the dominating (owner) entity set and *wd* is the discriminator (dashed-underlined attributes) of the weak entity set. Because of this, the cardinality of the relationship between a weak entity set and its dominator cannot be "many-to-many" (i.e., a weak entity cannot be associated with multiple owners).

Weak entity set can be represented by a multi-valued composite attribute of the dominator (a double ellipse represents a multi-valued attribute). An example is a list of old names used by employees. The author recommends readers consider converting multi-valued attributes into weak entities (nested entities).

4.1.4 Specialization & Generalization: A Concept of Inheritance

In the ER-model, “ISA” (a short for “is a”) in an upside-down isosceles triangle is generalization/specialization. Figure 10 shows a simple example of this. The diagram says that “Savings-account” and “Checking-account” are specializations (i.e., more specific sub-types) of Account; “Account” is a generalization (i.e., more generic type representing the common properties) of Savings-account and Checking-account. Both of the subtypes inherit all the attributes of the parent (i.e., “Account”).

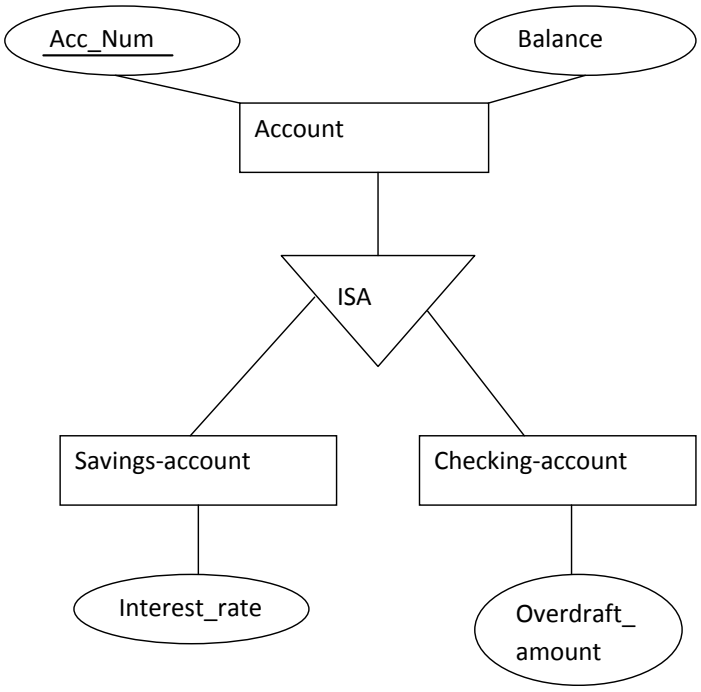


Figure 10. Generalization/Specialization

4.1.5 Aggregation: A Concept of Entity Composition

The ER-model also supports a composite entity concept. This feature is useful when one wants to declare a conceptual entity consisting of other entities. The other way of understanding this concept is rather clarified when one thinks of it as a method of representing a relationship between an entity and another relationship. For example, let us consider the following: a bank wants to link a loan account to a certain existing checking account. One has to make sure that the checking account and the loan account are of the same customer. That is, the loan account is associated with a pair of a certain customer and a certain checking account of the customer (useful for setting up an automatic payment out of the checking). In the ER-model, this is an aggregation, a rectangle enclosing the component entities and their relationships. Figure 11 shows this example of aggregation.

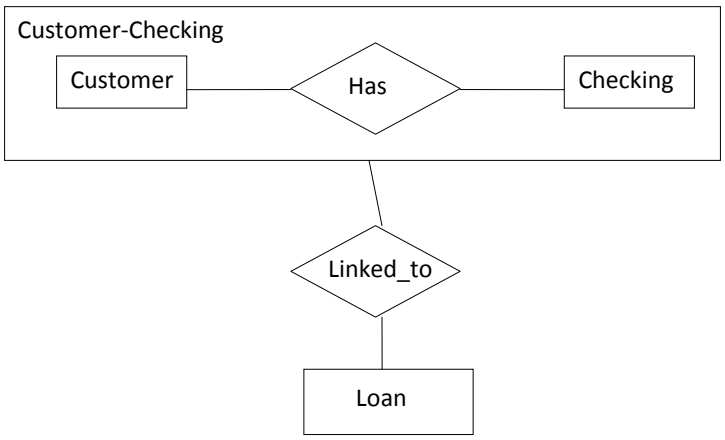


Figure 11. Aggregation

4.1.6 Example

Figure 12 shows an example of typical operational database application requirements.

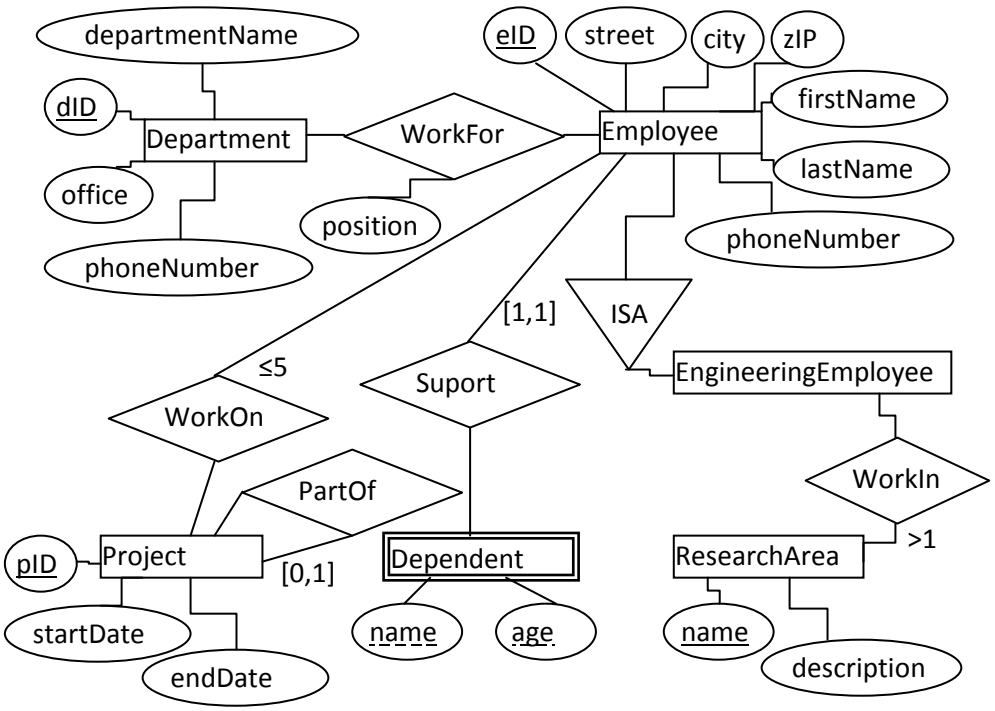
RDB Tech, Inc., needs an operational database of its employees, departments, and projects.

Every employee has an employee number (EID), name, address, and phone number. Each employee has a certain position in a department, and may have dependents. For each dependent, a dependent name and age are stored in the database. Each engineering type employee is associated with one or more research areas, and the company has a set of research areas. Each research area is represented by a unique name and short description.

Each department has a unique department number (DID), a department name, a phone number, and an office.

Each project has a unique project number (PID), project name, and duration. Each project is carried out by up to 5 employees. A project can be a subproject of another project.

(a)



(b)

Figure 12. Conceptual design example: a requirement description (a); an ER diagram (b)

4.1.7 Remarks

The expressive power of the ER-model largely overlaps that of modern mainstream SQL database models, but not completely. Especially, some of SQL constraints (i.e., table-wide Boolean conditions that must hold all the time over all database states/snapshots), assertions

(i.e., database-wide Boolean conditions that must hold all the time over all database states/snapshots), and triggers (Boolean conditions and their associated automatic operations on the database, changing the state/snapshot of the database when the conditions are met) are not supported. For such, additional free notes and annotations are often added to production-level designs. These concepts of modern database systems are further discussed in Section 5.

4.1.8 IDEF1X 1999: an integrated alternative notation

In 1993, the Computer Systems Laboratory of the US National Institute of Standards and Technology (NIST) released IDEF1X [NIST 1993] as a standard for data modeling in US federal information processing.

Like the ER-model, the IDEF1X notation system is focused on the concepts of relational databases. The IDEF1X supports entity concept using a visual box similar to the class box of the class diagram defined in UML (but without the operations). All the basic concepts of the ER-model discussed in this article are supported in the IDEF1X model through various notions including Entities, Connection Relationships (relationships in ER-model), Categorization Relationships (generalization & specialization), (Identifier-)Dependent entities, and various types of attribute and key.

Importantly, this integrated model's visual notation also covers some relational database concepts, such as candidate key (called alternate key in IDEF1X), foreign key, and some non-ER constraints, discussed in Section 4.2. Interested readers are referred to [NIST 1993].

4.2 Step 2: Conceptual Design to Logical Design

This article uses the modern Relational Model for logical database design. This section discusses how to convert a conceptual database design (in our case, an ER-diagram) into a relational database design.

4.2.1 Basic Components

The Relational Model is a table-based model (recall that "relation" means table, informally). This logical model has the following main components for database schema representation (more advanced concepts, which can enhance the original database design, will be discussed later in Section 4.3):

a. *relation schema* is a list of attribute names (defined domains), e.g. Account = {branch-name, account#, balance}. By means of access-by-name, a relation schema is often represented as a set of attribute names each of which is an application-defined name implicating (or paired with) a certain domain of values.

b. *relation instance* is a subset of a Cartesian product of a list of domains. Example: account(Account): "account" is a relation instance on "Account" schema. The relation instance "account" is a subset of D1 $\times$ D2 $\times$ D3, where D1, D2, and D3 are the domains of "branch-name", "account#", and "balance", respectively.

c. *tuple* is an element of a relation instance. If there are n attributes, each tuple is an ordered n-tuple. Thus, each tuple is an n-ary relationship between n attribute values. A relation is, in fact, a set of such relationships.

d. At the logical level, *database schema* is a set of relation schemas and *database instance* is a set of relation instances.

In the relational data model, each relation schema consists of *atomic domains*. A domain is atomic if each and every element of the domain is conceptually indivisible. The balance of an account is atomic. However, the domains of "date" or "author-list" of a book are not atomic. Modern object-relational DBMSs allow some non-atomic data types (domains), such as date, list-type, set-type, interval, time, and user-defined types (objects). It is important to note the definition of “atomic attribute” is determined by the requirements of the user. For example, a phone number with area code is atomic for a simple system, but not for a phone company where they would need the area code and extension broken out into their own fields.

In addition, in the table model, the concept of “key” is expanded to logically express inter-table links or relationships. In the Relational Model, the keys are defined as follows:

Superkey: a combination of one or more attributes (columns) on which every tuple (record) has unique combination of values.

Candidate key: a superkey that has no proper subset that is a superkey.

Primary key: one of the candidate keys chosen by the designer.

Foreign key: a combination of one or more attributes such that the combination is a candidate key of another relation (table).

4.2.2 Conversion Process

Given a conceptual ER design, one can convert the design into a logical design through the following conversion guidelines:

a. Strong entity sets:
A strong entity set A with attributes a1, a2, …, an is represented by a table called A with n distinct columns a1, a2, …, an. For example, the following table “Employee” represents the entity set Employee in Figure 8.

Employee

SSN	E_NAME	YRS_EMPLOYED

b. Weak entity sets:
A weak entity set A with attributes a_1, a_2, …, a_n can be represented by a table called A with distinct columns $\{b_1, b_2, \ldots, b_m\} \cup \{a_1, a_2, \ldots, a_n\}$, where $\{b_1, b_2, \ldots, b_m\}$ is the primary key of the dominating (owner) entity set. The primary key of this table will be $\{b_1, b_2, \ldots, b_m\} \cup$ {discriminator of A}. For example, the weak entity set "Dependent" in Figure 9 is converted into the following table "Dependent" (SSN is part of the primary key and also a foreign key referencing Employee(SSN)):

Dependent

SSN	**D_NAME**	D_AGE

c. Relationship sets:
An n-ary relationship set R, which has attributes r_1, r_2, …, r_k is represented by a table called R with distinct columns $\{k_1, k_2, k_3, \ldots, k_n\} \cup \{r_1, r_2, \ldots, r_k\}$, where $\{k_1, k_2, \ldots, k_n\}$ is the concatenation of the primary keys of all participating entity sets. The primary key of this table is $\{k_1, k_2, \ldots, k_n\}$. For example, the relationship set "Work_on" in Figure 8 is represented as follows (D# and SSN are foreign keys referencing Department(D#) and Employee(SSN), respectively):

Work_on

D#	**SSN**	POSITION

d. Many-to-one binary (or unary) relationship set:
We need not create a separate table for this type relationship set. Instead, add the primary key columns of "one" side entity set to the "many" side table as a foreign key referencing the primary key of the “one” side table.

e. One-to-one binary (or unary) relationship set:
We need not create a table for the relationship set. Add the primary key of either table to the other table as a foreign key

f. Generalization:
f.1 Convert the parent into a table;

f.2 Convert each child entity set into a table. Then, add the primary key of the parent to the table as its primary key attributes. For example, "Savings-account" in Figure 10 becomes Savings-account = {Acc_Num, Interest-rate} w/ primary key={Acc_Num} and foreign key={Acc_Num} referencing Account(Acc-Num).

g. Aggregation:
This is very straight forward. For example, relationship set R_2 in Figure 11 can be represented by a table R_2 that consists of columns {the primary key of R_1}∪{the primary key of E_3}∪{the attributes of R_2}.

4.2.3 Example

Figure 13 shows a logical (in our case, relational) version of the design in Figure 12. Note that this schema is the first version that requires some additional checking. This process is discussed in Section 4.3.

RDB Tech, Inc.,

Department={dID, office, departmentName, phoneNumber}
primary key:{dID}; no other candidate keys; no foreign keys

Employee={eID, firstName, lastName, phoneNumber, street, city, zip}
primary key:{eID}; no other candidate keys; no foreign keys

EngineeringEmployee={eID}
primary key:{eID}; no other candidate keys

```
    foreign keys:{eID} referencing Employee(eID)

Project={pID, startDate, endDate}
    primary key:{pID}; no other candidate keys; no foreign keys

Dependent={eID, name, age}
    primary key:{eID, name, age}; no other candidate keys
    foreign key:{eID} referencing Employee(eID)

ResearchArea={name, description}
    primary key:{name}; no other candidate keys; no foreign keys

WorkFor={dID, eID, position}
    primary key:{dID, eID}; no other candidate keys
    foreign key: {dID} referencing Department(dID)
    foreign key: {eID} referencing Employee(eID)

WorkOn={pID, eID}
    primary key: {pID, eID}; no other candidate keys
    foreign key: {pID} references Project(pID)
    foreign key: {eID} references Employee(eID)

WorkIn={eID, areaName}
    primary key: {eID, areaName} no other candidate key
    foreign key: {eID} references EngineeringEmployee(eID)
    foreign key: {areaName} references ResearchArea(name)
    //renaming is always allowed

PartOf={pID, partPID}
    primary key: {pID, partPID}; no other candidate keys
    foreign key: {pID} referencing Project(pID)
    foreign key: {partPID} referencing Project(pID)
```

Figure 13. Logical version of the database design in Figure 12 -- subject to normalization.

4.2.4 Remarks

The database schema in Figure 13 is not at a production level yet. It needs to be enhanced through some process summarized in the next section.

4.3 Step 3: Logical Design to Normalized Database

Given a database design application, there are always many alternatives: some viable, some not.

Let us consider the following relation: supplier={S_NAME, ADDR, PART#, PRICE}. This table is used to manage various parts and their suppliers. Unfortunately, this design can cause some potential problems in future as shown below, because each supplier's address ADDR is repeated for every part# associated with the supplier.

- An address change should be reflected in all tuples involving the supplier; but will it?
- It is impossible to store the name and address of a supplier, unless it supplies at least one part.
- If all supplied parts are deleted, we will unwillingly delete the supplier (i.e., the name and address).

Database normalization is finding and eliminating such potential problems before the database is created and populated.

There are several different levels of database normalization. Some of the normalization levels are as follows: First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), Boyce-Codd Normal Form (BCNF), Fourth Normal Form (4NF), etc.

Generally speaking, it is recommended that every relation (table) is normalized to 3NF. Then, relations are further normalized when it does not incur some other problems, such as loss of functional dependencies (defined later in this article).

Before the main discussion about the normal forms, Section 4.3.1 gives a background knowledge known as “functional dependency” that is required to understand database normalization processes.

Then, Sections 4.3.2 – 4.3.4 cover the first four normal forms. Finally, Section 4.3.5 gives section closing remarks.

4.3.1 Functional Dependency

Informally, a functional dependency (FD) appears when the value of a set (combination) of attributes uniquely determines the value of another set of attributes.

Example: schedule (PILOT, FLIGHT#, DATE, TIME)

1. FLIGHT# $\rightarrow$ TIME
2. {PILOT, DATE, TIME} $\rightarrow$ FLIGHT#

Note, the "$\rightarrow$" sign is read "functionally determines"

Definition 1. Functional Dependency (FD):

Let r(R) be a relation (R is the schema and r is an instance);
Let X $\subseteq$ R, and Y $\subseteq$ R;
Let s(X) = Π_X(r) (i.e., X part of r; vertical projection cutting off all columns except for X) and q(Y) = Π_Y(r) (i.e., Y part of r; vertical projection cutting off all columns except for Y).
There is a FD X $\rightarrow$ Y for every state of r if the mapping f:s(X) $\rightarrow$ q(Y) is one-to-one or many-to-one.

Properties of FDs

1. If X $\rightarrow$ Y, then $\forall$ t1, t2 $\in$ r(R), t1[Y]=t2[Y] if t1[X]=t2[X].
2. FD is a time invariant property of a relation r, i.e. it holds for all states of r.

4.3.2 The First Three Normal Forms

Definition 2. First Normal Form (1NF): A relation r(R) is in 1NF if all tuples contain single-valued fields (i.e., an attribute field cannot contain a set of values from its domain). A relational database is in 1NF if all of its relations are in 1NF.

Example:

graduates (	STUDENT#	F_NAME	YEAR)
	{001, 005}	{John, Peter}	1997
	{003, 010, 101}	{Ann, Ivan, Sam}	2001

Problem(1): STUDENT# → F_NAME? (i.e., given a student number, what is the corresponding first name?)

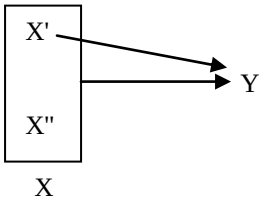
Problem(2): How can we change Ivan's graduation year to 2002?

With the following 1NF, those problems do not occur:

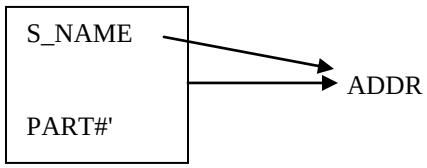
graduates (	STUDENT#	F_NAME	YEAR)
	001	Peter	1997
	005	John	1997
	.	.	.
	.	.	.

Remaining Problems: Recall, the supplier relation is in 1NF and there are four problems (i.e., repetition of info. and anomalies). *Partial FD* causes these problems.

Definition 3. Partial FDs: Let r(R) and X, Y $\subset$ R. A non-trivial FD X$\rightarrow$Y is a full FD if $\forall$X' $\subset$ X, s.t. X'-/-> Y. Otherwise, if $\exists$X' $\subset$ X, s.t. X'$\rightarrow$Y, then X$\rightarrow$Y is a partial FD. Note: s.t. is a short for "such that".



Example: supplier (S_NAME, ADDR, PART#, PRICE), FDs: {S_NAME$\rightarrow$ADDR, S_NAME PART#$\rightarrow$P}. Then, S_NAME PART# $\rightarrow$ ADDR is a partial FD.



ADDR is not fully dependent on the primary key. Therefore, an ADDR value must be repeated for all tuples whose S_NAME values are the same.

By eliminating every partial FD, one can eliminate some anomalies.

Definition 4. Second Normal Form (2NF): Let attributes of candidate keys be called primary attributes, and let all others be secondary attributes. A relation r(R) is in 2NF iff it is in 1NF and no secondary attribute is functionally determined by a proper subset of a candidate key.

A database is in 2NF if every relation is in 2NF.

Example:
supplier (S_NAME, ADDR)
F1={S_NAME → ADDR} //functional dependencies
K1={S_NAME} // the only candidate key of this relation
supplies (S_NAME, PART#, PRICE)
F2={S_NAME PART# → PRICE} //functional dependencies
K2={S_NAME, PART#} // the only candidate key of this relation
These relations are in 2NF, since:
(1) In "supplier", the secondary attribute ADDR is fully dependent on K1
(2) In "supplies", the secondary attribute PRICE is fully dependent on K2

If r(R) has no secondary attributes, then it is automatically in 2NF.

If, in r(R), all candidate keys have a single attribute, then it is automatically in 2NF.

Note that, by the definition of 2NF, primary attributes need not be fully dependent on candidate keys.

Example:
r (ABCDEG) is in 1NF.
F={ABC->DE, DE->ABC, DE->G, AB->D, E->C}
K={D,E}, {A,B,C}, {A,B,E}
The only secondary attribute G is fully dependent on DE, ABC, and ABE. Thus, the relation is in 2NF.
However, DE->C is a partial FD, since E->C. Also, ABC->D is a partial FD, since AB->D.

Remaining Problems: Restriction imposed by 2NF is not rigorous enough to prevent all anomalies.

Example: Consider the following relation in 2NF
department (DEPT D_OFFICE SECRETARY)
F={SECRETARY→DEPT, DEPT→D_OFFICE}
Note, DEPT -/-> SECRETARY, D_OFFICE -/-> SECRETARY
That is, a department can have more than one secretary. Each secretary must work for only one department. Each department has only one office.

(1) Repetition of Information: D_OFFICE must be unnecessary repeated for every secretary working in the same DEPT.

(2) Update Anomaly: If the department office changes its address, many tuples must be updated.

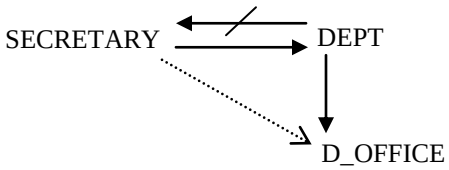
(3) Insertion Anomaly: New department cannot be inserted before at least one secretary is assigned.

(4) Deletion Anomaly: If all secretaries of a department retire, the department will be deleted.

Definition 5. Transitive Dependency: Let r(R) and X,Y,Z $\subset$ R. Then, X→Z is a transitive dependency iff X→Y, Y→Z, and Y-/->X hold, Y→Z is a non-trivial FD, i.e., Z is not a subset of Y, and Z is all secondary attributes.

Example: department (DEPT D_OFFICE SECRETARY)

SECRETARY → DEPT
DEPT → D_OFFICE
DEPT -/-> SECRETARY



By eliminating every transitive FD, we can eliminate some anomalies.

Definition 6. Third Normal Form (3NF): r(R) is in 3NF iff it is in 2NF and no secondary attribute of r is transitively dependent on a candidate key of r. A relational DB is in 3NF if every relation is in 3NF.

If r(R) is in 3NF, then it is in 2NF.

3NF Normalization Example 1:
department (DEPT D_OFFICE SECRETARY)

F = {SECRETARY → DEPT, DEPT → D_OFFICE}
DEPT -/-> SECRETARY
(1) R1={DEPT, D_OFFICE, SECRETARY}
(2) X={SECRETARY}, Y={DEPT}, Z={D_OFFICE}
(3) R1=R1-Z={DEPT, SECRETARY}, R2=YZ={DEPT, D_OFFICE}
(4) No more transitive FD

3NF Normalization Example 2:
patrol(PLACE, TIME_COVERAGE, PATROL_CAR, *L_P*), where TIME_COVERAGE is a period of the day during which the PLACE is being PATROLED and *L_P* is the *Location of the PATROL_CAR*

F={PLACE TIME_COVERAGE→PATROL_CAR, PATROL_CAR→L_P}
Candidate Key: {PLACE, TIME_COVERAGE} //r has only one candidate key

Note: This relation is in 2NF, but not in 3NF: the L_P of a PATROL_CAR is repeated for all <PLACE TIME_COVERAGE>s associated with the car. This is more pronounced as the location (L_P) of the patrol car is more frequently updated.

3NF Normalization output: patroling(PLACE, TIME_COVERAGE, PATROL_CAR), patrol_car(PATROL_CAR, L_P)

Remaining Problems: Does not eliminate transitive FDs of primary attributes on candidate keys, which can also cause anomalies. Considering the following schema:

address (STREET CITY ZIP),
F={STREET CITY → ZIP, ZIP → CITY}
Candidate Keys: {STREET CITY}, {ZIP STREET}

Since "CITY" is not a secondary attribute, the relation is in 3NF. However, there are anomalies:

(1) Repetition of Information: CITY is unnecessary repeated for every STREET with the same ZIP.
(2) Update Anomaly: If CITY changes its name, then many tuples must be changed.
(3) Insertion Anomaly: (CITY ZIP) cannot be represented without a STREET.
(4) Deletion Anomaly: If one deletes every STREET of a ZIP, then the <CITY ZIP> pair will be deleted.

4.3.3 Boyce-Codd Normal Form (BCNF)

Definition 7. BCNF: Let r(R) be a relation in 1NF, $X,Y \subset R$, and $X \cap Y = \emptyset$. The relation r(R) is in BCNF iff for all FDs $X \rightarrow Y$, X is a candidate key of r(R).

For example, the last example in Section 4.3.2 can be decomposed into two BCNF relations address(STREET, CITY) and zip(ZIP, CITY).

Relational DB is in BCNF if all relations are in BCNF. If r(R) is in BCNF, then it is in 3NF. BCNF schema decomposition is lossless but may induce loss of FDs – a checking is required.

4.3.4 Example

Let us consider the two relations Department and Employee in Figure 13. These two relations are interesting, since they have relatively large relation cardinalities (number of columns).

Note that functional dependencies (FDs) are statements about the real world made by the database designer in accordance with the semantics of the attributes and the reality which is modeled. FDs are axioms about the world which cannot be proven. Given this fact, suppose that the following FDs hold:

Department: phoneNumber → office
Employee: phoneNumber→ street city; street city → zip; zip → city

Then, one may want to decompose/normalize the tables as follows:

Department={dID, departmentName, phoneNumber}
OfficePhone={phoneNumber, office}
Employee={eID, firstName, lastName, phoneNumber}
PhoneAddress={phoneNumber, street, zip}
ZipCity={zip, city}

4.3.5 Higher Normal Forms

Thus far, the normalizations have been based on functional dependencies. The next level normalization is based on a different form of dependency, called *multivalued dependency* (*MVD*). Let R be a relation and let $X \subseteq R$, $Y \subseteq R$, and R-X-Y be non-empty. A multivalued

dependency X $\rightarrow\rightarrow$ Y states that, for all possible instances of R, each certain value of X determines a certain set of values of Y, independent to all the other attributes R-X-Y. An MVD X $\rightarrow\rightarrow$ Y is said to be non-trivial if Y is not a subset of X and X is not a superkey. Each non-trivial MVD X $\rightarrow\rightarrow$ Y causes data redundancy in the relation.

For example, considering a relation FranchiseRestaurant={franchiseName, menuItem, branchPhone}, if all branches of each franchise must offer the same manu, the following non-trivial MVDs hold: *franchiseName* $\rightarrow\rightarrow$ *menuItems*; *franchiseName* $\rightarrow\rightarrow$ *branchPhone*. Considering a franchise with 10 menu items and 10 branch phone numbers, the franchise name will be appear 100 times, each item name 10 times, and each branch phone number 10 times.

To reduce this repeatition of information, the relation can be decomposed into two relations: {franchiseName, menuItem} and {franchiseName, branchPhone}. The result is the following: the franchise name appears 10 times in each table (total 20) and no duplications in menuItem and brachPhone.

This process, eliminating all non-trivial MVDs, are called the 4NF (Fourth Normal Form) normalization and a relation that has no non-trivial MVDs is said to be in 4NF. The formal definitions and theoretical details are outside the scope of this article. For inspired readers, formal definitions of MVD, 4NF, and relevant algorithms can be found in [Garcia-Molina et al., 2009]. For interested readers, even higher level normal forms (e.g., DKNF) can be found in [Silberschatz et al., 2005c].

4.3.6 Remarks

Database normalization is a critical and mandatory database schema design step and its importance cannot be understated. Functional dependencies are not automatically generated; database designers declare functional dependencies in their databases. Because of space limitation, this article provides only the key concepts of the normalization. Serious database designers are referred to complete database design textbooks.

5. Database Query

SQL (a short for “structured query language”) was developed as an implementation of the Relational Model’s pure database languages including relational algebra [Garcia-Molina et al., 2009, Silberschatz et al., 2005c] and calculus [Silberschatz et al., 2005c]. SQL standards SQL:1992 (aka SQL II) [ISO, 1999] and SQL:1999 (aka SQL III) [ISO, 1999] are widely supported by all relational database management systems and arguably the most popular database language based on a strong logical and mathematical foundation. SQL:2003 [ISO, 2003], which supports some object-oriented concepts, is under deployment and field testing.

It is important to note that each individual DBMS’s SQL implementation often deviates from the standard (e.g., some not-implemented operations, minor differences in punctuation rules, and additional operations not defined in the standard, etc). For this, interested readers are recommended to carefully review their specific DBMS’s SQL manual. The readers are also recommended to review the pure relational database languages (relational algebra and calculus) found in full textbooks to better understand the logics behind SQL. In addition, because of the space limitation, this article reviews only some basic features of the SQL standards. Some additional information can easily be found on the web (e.g., [ISO, 1999; ISO, 2003; UMIT, 2008].

SQL enables applications and users to communicate with the underlying DBMS for "storage", "update", and "retrieval" of data. Because of space limitations, this article briefly introduces the fundamental features of SQL. More details can be found in SQL manuals that are available at many Web sites – readers are recommended to study the full standard SQL at "www.w3schools.com/sql/" [W3Schools, 2008].

5.1 Storage: Database Schema Manipulation

SQL provides the following basic commands for database schema manipulation:

- CREATE TABLE – creates a new empty database table (creates a new relation schema)
- ALTER TABLE – changes the structure of an existing database table (modifies an existing relation schema)
- DROP TABLE – deletes a database table
- CREATE INDEX – creates an index on selected attributes (search key) of a database table for faster manipulation and search
- DROP INDEX – deletes an existing index

For example, Dependent and ResearchArea tables in Figure 13 can be created as follows:

```
CREATE TABLE Dependent (
        eID INTEGER,
        name VARCHAR,
        age INTEGER,
        PRIMARY KEY (eID, name, age),
        FOREIGN KEY (eID) REFERENCING Employee (eID)
);

CREATE TABLE ResearchArea (
        name VARCHAR,
        description VARCHAR,
        PRIMARY KEY (name)
);
```

5.2 Update: Database Instance Manipulation

SQL provides several commands for populating and changing database instance at record (tuple) and attribute levels. Such commands include "INSERT INTO *table-name* …", "UPDATE *table-name* SET …", "DELETE FROM *table-name* WHERE …". For example, one can change the instance of Dependent as follows:

```
INSERT INTO Dependent (eID, name, age) VALUES (100, Thomas, 9);
UPDATE Dependent SET age=10 WHERE eID=100 AND name='Thomas' AND age=9;
DELETE FROM Dependent WHERE eID=100;
```

5.3 Retrieval

SQL provides “SELECT *attribute_name(s)* FROM *table_name(s)* WHERE *selection_condition(s)* …” command for data retrieval. For example, “SELECT * FROM Dependent” is read “report all Dependent records”. More examples:

Report all dependents of Employee 100:
SELECT * FROM Dependent WHERE eID=100;

For each employee, report his/her lastName, phoneNumber and the average age of all his/her dependents:
SELECT eID, lastName, phoneNumber, AVG(age) FROM Dependent, Employee WHERE Dependent.eID=Employee.eID GROUP BY (eID);

5.4 Additional Features

For database consistency, SQL provides a rich set of commands that can be used by database designers to describe “mandatory” conditions that must hold all the time. Such commands include column-level constraints, table-level constraints, and database-wide constraints (assertions). A column-level constraint (e.g., NOT NULL and UNIQUE) is associated with and can be placed right next to an attribute (next to the type). Every value in the column must satisfy the condition. The Primary Key and Foreign Key constraints in Section 5.1, for example, are frequently declared table-level constraints that must be upheld by all states of the table. Database-wide constraints, also known as the assertions, can be declared in SQL. A rich set of various constraints are defined in SQL [W3Schools, 2008].

Database “views” are virtual tables (stored retrieval queries) based on concrete tables. SQL’s “create view” command [W3Schools, 2008] can be used to create a permanent virtual table in the database (of course, “drop view” command can remove any view). Although there are non-materialized views, this article recommends users view each “view” as named table holding the result of a retrieval query. Such views can enable us to open up only a carefully selected portion of the database to a certain group of users, while securing and protecting the rest of the database from the user group.

A database transaction is an atomic sequence of queries. For example, moving $500 from a checking account to a saving account may consist of two update queries. The point is that it must be atomic – the final state is either the state of both statements completed or the state before the first statement started. This is “all or nothing” or “atomic transaction” concept. In SQL, “commit” command [W3Schools, 2008] represents the “all” and “rollback” command [W3Schools, 2008] represents the “nothing”. Partially processed transactions are never allowed to cause any actual changes in the database.

A database trigger is a registered and all-time running query consisting of two parts: trigger-condition and trigger-action. Each trigger-condition is evaluated on every state of the database and the action is invoked whenever the condition is met by the database state. This is a very powerful and useful mechanism for building “active” database systems that can respond to external and internal events without human intervention. SQL:1999 [ISO, 1999; ISO, 2003] has “create trigger” command [W3Schools, 2008] defined.

6. Database Optimization

Since database applications are typically designed to deal with large sets of increasing data, performance and scalability issues are often pronounced. Fortunately, today's database management systems are smart enough to automatically enhance the physical structures of all databases as well as queries in main memory, in secondary storage (e.g., disks), and in tertiary storage (e.g., back-up tape) in such a way that the overall performance and scalability of the system are substantially enhanced.

One simple and easy performance optimization database administrators can do is creating appropriate indexes. An index is a separate data structure or file that contains only the selected columns of the base table and that is physically optimized for some types of look-up operations. Such structures can be effectively used by the DBMS for quick search.

For example, when a new data tuple/record is inserted into a table, the DBMS must make sure that the new tuple satisfies "all candidate key values are unique in the table" condition. A separate index on the candidate key shall speed-up this checking that, otherwise, requires accessing all existing records. Many DBMS implementations, such as Oracle, automatically create an index on every primary key. Recall that the primary key of a table is just one of the candidate keys of the table chosen by the designer/DBA.

Each non-primary-key candidate keys consisting of *n* attributes a1, a2, …, a*n* must be specified in the "CREATE TABLE" statement as follows: Each of the attributes comes with "NOT NULL" column constraint following its data type in the statement; a table constraint "UNIQUE (a1, a2, …, a*n*)" is added to the statement. Then, the DBA can build an index on the candidate key using the following SQL statement "CREATE [UNIQUE] INDEX [*index_name*] ON (a1, a2, …, a*n*)", where the bracket represents an optional component. The following SQL statements give an example of candidate key.

```
CREATE TABLE Student (
        …
        name VARCHAR(100) NOT NULL,
        phone INTEGER NOT NULL,
        …
        UNIQUE (name, phone)
        …
);
…
CREATE UNIQUE INDEX Stu_candi_key1 ON Student(name, phone);
```

Similarly, all foreign keys must satisfy the referential integrity condition: Every foreign key value must exist in the matching candidate key columns in the referenced table. That is, when the matching candidate key value is changed or deleted, something should be done by the DBMS (e.g., give a warning to the DBA or automatically change the foreign key value, etc). For the DBMS to efficiently do this, it is also recommended to create a non-unique index on every foreign key in the database. For example, Dependent table in Figure 13 may take benefit from "CREATE INDEX Dep_owner ON Dependent(eID)".

Moreover, building indexes on some attributes that appear in many query search conditions (e.g., "WHERE" clauses) possibly provides the underlying DBMS with better optimization possibilities. However, this requires some insight into the DBMS internals and deeper understanding of query optimization that are beyond the scope of this article. Interested readers

are referred to full DBMS textbooks, specific DBMS manuals, and database query optimization technical articles.

7. Database Support for New Emerging Applications: Spatiotemporal Applications

An increasing number of emerging applications deal with a large number of *continuously changing* (or moving) *data objects* (*CCDO*s), such as vehicles, sensors, and mobile computers. For example, in earth science applications, temperature, wind speed and direction, radio or microwave image, and various other measures (e.g., CO2) associated with a certain geographic region can change continuously. Accordingly, new services and applications dealing with large sets of CCDOs are appearing. In the future, more complex and larger applications that deal with higher dimensional CCDOs (e.g., a moving sensor platform capturing multiple stimuli) will become commonplace – increasingly complex sensor devices will continue to proliferate alongside their potential applications. Efficient support for these CCDO applications will offer significant benefit in many broader challenging areas including mobile databases, satellite image analysis, sensor networks, homeland security, internet security, environmental control, and disease surveillance.

To support large-scale CCDO applications, one requires a data management system that can store, update, and retrieve CCDOs. Each CCDO has both multidimensional-temporal (i.e., 2 or 3D geographic space or other information dimensions that vary with time) properties representing its continuous trajectory in an information space-time continuum as well as non-temporal properties such as identification, associated phone number, and address. Importantly, although CCDOs can continuously move or change (thus drawing continuous trajectories in a space-time), computer systems cannot deal with continuously occurring infinitesimal changes – this would effectively require infinite computational speed and sensor resolution. Thus, each object's continuously changing attribute values (e.g., location, velocity, and acceleration) can only be discretely updated. Hence, they are always associated with a degree of uncertainty, especially when there is a considerable time gap between two updated points.

7.1 Related Research

Considering an observer who records (or reports) the state of a continuously moving object as often as possible, the object is viewed as a sequence of connected segments in space-time, and each segment connects two consecutively reported states of the object. Table 1 explicates a point-type CCDO concept.

Figure 14 shows a generic CCDO schema that can support the ontological concepts explicated in Table 1. By examining Table 1 and the commensurate ER diagram in Figure 14, one may observe the following: (1) *snapshots* are not represented in the schema; (2) only a subset of *states*, called *reported states*, are included in the schema. These differences exist due to the fact that a database cannot be continuously updated. The reported states are the known states of the CCDOs, and these known states can be committed to the database. All in-between states and future states of the CCDOs are then interpolated and extrapolated on the fly only when it is necessary for query processing, data visualization, index maintenance, or data management.

Table 1. Multi-level semantic abstraction of CCDO

***Trajectory*:** A trajectory consists of *dynamics* and *f:time* → *snapshot*, where *time* is a past, current, or future point in time.

***Snapshot*:** A snapshot is a probability distribution that represents the probability of every possible state in the data space at a specific point in time. Depending on the *dynamics* and update policies, the probability distribution may or may not be bounded.

***State*:** A state is <*P*, *O*>, where *P* is a location (position) in the data space-time (i.e., a point in the space-time), and *O* is optional property list including zero or more of the following: orientation, location velocity vector (and rotation velocity vector, if orientation is used), location acceleration vector (and rotation acceleration vector, if orientation is used), and even higher derivatives at the time of *P*.

***Dynamics*:** The dynamics of a trajectory is the lower and upper bounds of the optional properties of all states of the trajectory.

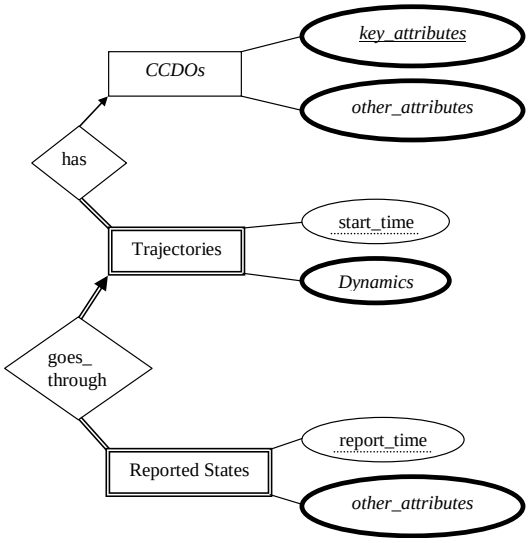


Figure 14. An ER schema representing a generic CCDO set – a bolded ellipse represents a set of customizable attributes

In summary, for any CCDO that draws a continuous trajectory in space-time, only a subset of the object's states can be stored in the database. Each pair of consecutive reported states of the CCDO represent a single trajectory segment. Any number of unknown states can exist between two known states. In order to support queries referring to the trajectories of CCDOs without false dismissals, one needs an estimation model, called the uncertainty model that covers all possible unknown states (conservative estimation of all possible unknown states).

This challenging aspect of CCDO management is much more pronounced in practice, especially when limited detection intervals (e.g., 1-sec GPS interval), message transmission delays (e.g., SMS or SMTP), more dynamic object movements (e.g., unmanned airplanes), and the scalability to larger data sets are considered. In some applications, even reported states have uncertainty: each reported state is associated with a snapshot (uncertainty) due to various

instrument and measurement errors (e.g., limited sensor resolution and data value normalization, and data approximation). For these reasons, a number of uncertainty models have been proposed.

Several application-specific trajectory models have been proposed. One model is that, at any point in time, the location of an object is within a certain distance *d*, of its last reported location. If the object moves further than *d*, it reports its new location and possibly changes the distance threshold *d* for future updates [Wolfson et al., 1999]. Given a point in time, the uncertainty is a circle with a radius *d*, bounding all possible locations of the object. Some other models assume that an object always moves along straight lines (linear routes). The location of the object at any point in time is within a certain interval, centered at its last reported location, along the line of movement. Different trajectory models that have no uncertainty consideration are found in the literature [Pfoser and Jensen, 2001]. These models make sure that the exact velocity is always known by requiring reports whenever the object's speed or direction changes. Some other models assume that the object travels with a known velocity along a straight line, but can deviate from this path by a certain distance [Sistla et al., 1997; Trajcevski et al., 2002; Trajcevski et al., 2004]. These models presented as spatiotemporal uncertainty models that produce 3-dimensional cylindrical uncertainty regions representing the past uncertainties of trajectories. Another model called the parametric polynomial trajectory model [Ni and Ravishankar, 2005] was designed not to represent unknown states but to approximate known locations of a trajectory, assuming that all points of the trajectory are given (approximation of discretized and complete trajectories).

Another study on the issues of spatial uncertainty in the recorded past trajectories (history) of CCDOs is found in [Pfoser and Jensen, 1999]. Assuming that the maximum velocity of an object is known, they prove that all possible locations of the object during the time interval between two consecutive observations (reported states) lie on an error ellipse. A complete trajectory of any object is obtained by using linear interpolation between two adjacent states. That is, a trajectory is approximated by a sequence of connected straight lines, each of which connects two consecutively reported CCDO observations. By using the error ellipse, the authors demonstrate how to process uncertainty range queries for trajectories. This error ellipse is the projection of a three-dimensional spatiotemporal uncertainty region onto the two-dimensional data space. Similarly, [Hornsby and Egenhofer, 2002] represents the set of all possible locations based on the intersection of two half cones that constraint the maximum deviation from two known locations. It also introduces multiple granularities to provide multiple views of a moving object.

More recently developed spatiotemporal uncertainty models reported in [Yu, 2006; Yu et al., 2007] formally define both past and future "spatiotemporal" uncertainties of CCDO trajectories of any dimensionality. In this model, the uncertainty of the object during a time interval is defined to be the overlap between the two spatiotemporal volumes, called funnels or tornados. This higher degree model [Yu et al., 2007] reduces the uncertainty by taking into account the temporally-varying higher order derivatives, such as velocity and acceleration. This model can also represent the uncertainties of not only locations but also some higher order derivatives, the velocity and acceleration.

8. Conclusion

As the computers grew in popularity and capability, more data-intensive applications (or database applications) required reliable, robust, and secure mechanisms for managing and

sharing large collections of complicated data. A rapidly increasing number of applications manage large collections of data.

This article provided readers with a comprehensive overview of modern database technology. Standard database development methods were presented along with the relational database foundation and the fundamental database system theories and practical techniques.

Moreover, the article presented the current and future database research and practice issues in the area of emerging spatiotemporal applications.

Glossary

Database (DB): Collection of Structured Data – the structure part is called schema and the actual data part is called instance.

Database Management System (DBMS): Computer software that manages databases and that controls all accesses to the databases.

Database System (DBS): A complete application system consisting of databases, database management system, and application programs that interact with the database management system to access the databases.

Database Model: A conceptual (visual) or logical language for expressing databases and database accesses.

Structured Query Language (SQL): An ISO standard and textual database query language developed based on the relational database model.

Spatiotemporal Database: A database of objects that exist in space-time continuum.

Continuously Changing Data Object (CCDO): A data object that can continuously change its position in a geographical, physical, or multi-dimensional information space over time.

References

Bachman C., Programmer as a Navigator, *Communications of ACM*, Vol. 16, Issue 11, pp. 653-658, 1973.

Bjorner, D. and Lovengreen, H.H., Formalization of Database Systems and a Formal Definition of IMS, *Proc. of Very Large Data Bases*, pp. 334-347, 1982.

Chen, P.P. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1 (1), pp. 9-36, 1976.

CODASYL, DBTG Report, April 1971, 264 pp., *ACM, SIGMOD Anthology*, Vol. 6, 2007, Digitized from paper originals in the Charles Babbage Institute at the University of Minnesota, 1971.

Codd, E.F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* 13 (6): 377–387, 1970.

Date, C. J., Darwen, H. *Foundation for Future Database Systems: The Third Manifesto*, 2nd edition, Addison-Wesley Professional. ISBN 0-201-70928-7, 2000. –SQL, OO, OR, Rel DB

Garcia-Molina, H., Ullman, J.D., Widom, J. *Database Systems The Complete Book*, 2nd edition, Pearson Prentice Hall, ISBN 0-13-606701-8, 2009.

Hornsby, K. and Egenhofer, M.J. Modeling Moving Objects over Multiple Granularities. *Annals of Mathematics and Artificial Intelligence*, Vol. 36 (No. 1-2), pp. 177-194, 2002.

IBM, Information Management System, www.ibm.com/ims, 2008

ISO. SQL:1999. ISO/IEC 9075-*:1999 or INCITS/ISO/IEC 9075-*-1999 from eStandards Store at http://www.ansi.org, 1999.

ISO. SQL:2003. ISO/IEC 9075-*:2003 or INCITS/ISO/IEC 9075-*-2003 from eStandards Store at http://www.ansi.org, 2003.

Kim, S. H., Yu, B., and Chang, J., Zoned-Partitioning of Tree-Like Access Methods, the *Journal of Information Systems*, Vol. 33, Issue 3, pp. 315-331, *Elsevier*, 2008

Lyngbaek, P. and Vianu, V. Mapping Semantic Database Model to the Relational Model. *Proc. of the ACM SIGMOD Conf. on Management of Data*, pp. 132-142, 1987.

Markowitz, V.M. and Shoshani, A. Represented Extended Entity-Relationship Structures in Relational Databases. *ACM Transactions on Database Systems*, Vol. 17, pp. 385-422, 1992.

NIST: National Institute of Standards and Technology. Integration Definition for Information Modeling (IDEF1X), Federal Information Processing Standards Publication (FIPSPUB) 184, National Technical Information Service, US Department of Commerce, Springfield, VA 22161, 1993 (also available at http://www.idef.com/Downloads.htm).

Ni, J. and Ravishankar, C.V. PA-Tree: A Parametric Indexing Scheme for Spatiotemporal Trajectories. *Proc. Int. Symposium on Spatial and Temporal Databases, LNCS Lecture Notes in Computer Science*, Vol. 3633, pp. 254-272, 2005.

Pfoser, D. and Jensen, C.S. Capturing the Uncertainty of Moving-Objects Representations. *Proc. SSDBM Int. Conf. on Scientific and Statistical Database Management*, pp. 123-132, 1999.

Pfoser, D. and Jensen, C.S. Querying the Trajectories of On-Line Mobile Objects. *Proc. ACM MobiDE International Workshop on Data Engineering for Wireless and Mobile Access*, pp. 66-73, 2001.

Rumbaugh, J., Jacobson, I., and Booch, G. The Unified Modeling Language Reference Manual, Second Edition. Addison-Wesley, ISBN 0-321-24562-8, 2005.

Silberschatz, A., Korth H.F., Sudarshan, S., Network Model, http://codex.cs.yale.edu/avi/db-book/online-dir/a.pdf (available as of March 10, 2008), 2005a.

Silberschatz, A., Korth H.F., Sudarshan, S., Hierarchical Model, http://codex.cs.yale.edu/avi/db-book/online-dir/b.pdf (available as of March 10, 2008), 2005b.

Silberschatz, A., Korth, H., and Sudarshan, S. *Database System Concepts*, 5th edition, McGraw Hill. ISBN 0-07-295886-3, 2005c.

Sistla, A.P., Wolfson, O., Chamberlain, S., and Dao, S. Querying the Uncertain Position of Moving Objects. *Temporal Databases: Research and Practice, LNCS Lecture Notes in Computer Science*, Vol. 1399, pp. 310-337, 1997.

Trajcevski, G., Wolfson, O., Hinrichs, K., and Chamberlain, S. Managing Uncertainty of Moving Objects Databases. *ACM Trans. On Databases Systems*, Vol. 29 (No. 3), pp. 463-507, 2004.

Trajcevski, G., Wolfson, O., Zhang, F., and Chamberlain, S. The Geometry of Uncertainty in Moving Object Databases. *Proc. Int. Conf. on Extending Database Technology*, May, 2002.

UMIT. Basic SQL Syntax Diagram: http://ii.umit.at/teaching/isdb05/sql92.pdf, retrieved on Dec. 30, 2008.

Wolfson, O., Sistla, A.P., Chamberlain, S., and Yesha, Y. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases*, Vol. 7, No. 3, pp. 257-387, 1999.

W3Schools. *SQL*, www.w3schools.com/sql, 2008.

Yu, B. A Spatiotemporal Uncertainty Model of Degree 1.5 for Continuously Changing Data Objects. Proc. ACM Int. Symposium on Applied Computing, Mobile Computing and Applications, pp. 1150-1155, 2006.

Yu, B., Kim, S.H., Alkobaisi, S., Bae, W.D., and Bailey, T. The Tornado Model: Uncertainty Model for Continuously Changing Data. *Int. Conf. on Database Systems for Advanced Applications, LNCS Lecture Notes in Computer Science*, Vol. 4443, pp. 624-636, 2007.