أساتذة المادة: د.مهدي دعيمي
سرمد مكي
ايمان حاتم

The Information System Life Cycle

In a large organization, the database system is typically part of the **information system,** which includes all resources that are involved in the collection, management, use, and dissemination of the information resources of the organization. In a computerized environment, these resources include the data itself, the DBMS software, the computer system hardware and storage media, the personnel who use and manage the data (DBA, end users, parametric users, and so on), the applications software that accesses and updates the data, and the application programmers who develop these applications. Thus the database system is part of a much larger organizational information system.

In this section we examine the typical life cycle of an information system and how the database system fits into this life cycle. The information system life cycle is often called the **macro life cycle,** whereas the database system life cycle is referred to as the **micro life cycle.**

The Database Application System Life Cycle

Activities related to the database application system (micro) life cycle include the following phases:

1. *System definition:* The scope of the database system, its users, and its applications are defined. The interfaces for various categories of users, the response time constraints, and storage and processing needs are identified.
2. *Database design:* At the end of this phase, a complete logical and physical design of the database system on the chosen DBMS is ready.
3. *Database implementation:* This comprises the process of specifying the conceptual, external, and internal database definitions, creating empty database files, and implementing the software applications.
4. *Loading or data conversion:* The database is populated either by loading the data directly or by converting existing files into the database system format.
5. *Application conversion:* Any software applications from a previous system are converted to the new system.
6. *Testing and validation:* The new system is tested and validated.
7. *Operation:* The database system and its applications are put into operation. Usually, the old and the new systems are operated in parallel for some time.
8. *Monitoring and maintenance:* During the operational phase, the system is constantly monitored and maintained. Growth and expansion can occur in both data content and software applications. Major modifications and reorganizations may be needed from time to time.

Activities 2, 3, and 4 together are part of the design and implementation phases of the larger information system life cycle. Our emphasis in the next Section is on activity 2, which covers the database design phase. Most databases in organizations undergo all of the preceding life-cycle activities. The conversion steps (4 and 5) are not applicable when both the database and the applications are new. When an organization moves from an established system to a new one, activities 4 and 5 tend to be the most time-consuming and the effort to accomplish them is often underestimated. In general, there is often feedback among the various steps because new requirements frequently arise at every stage. Figure

(1) shows the feedback loop affecting the conceptual and logical design phases as a result of system implementation and tuning.

The Database Design Process

We now focus on step of the database system life cycle, which is database design. We can identify six main phases of the database design process:

- Phase 1: Requirements Collection and Analysis
- Phase 2: Conceptual Database Design
- Phase 3: Choice of a DBMS
- Phase 4: Data Model Mapping (Logical Database Design)
- Phase 5: Physical Database Design
- Phase 6: Database System Implementation and Tuning

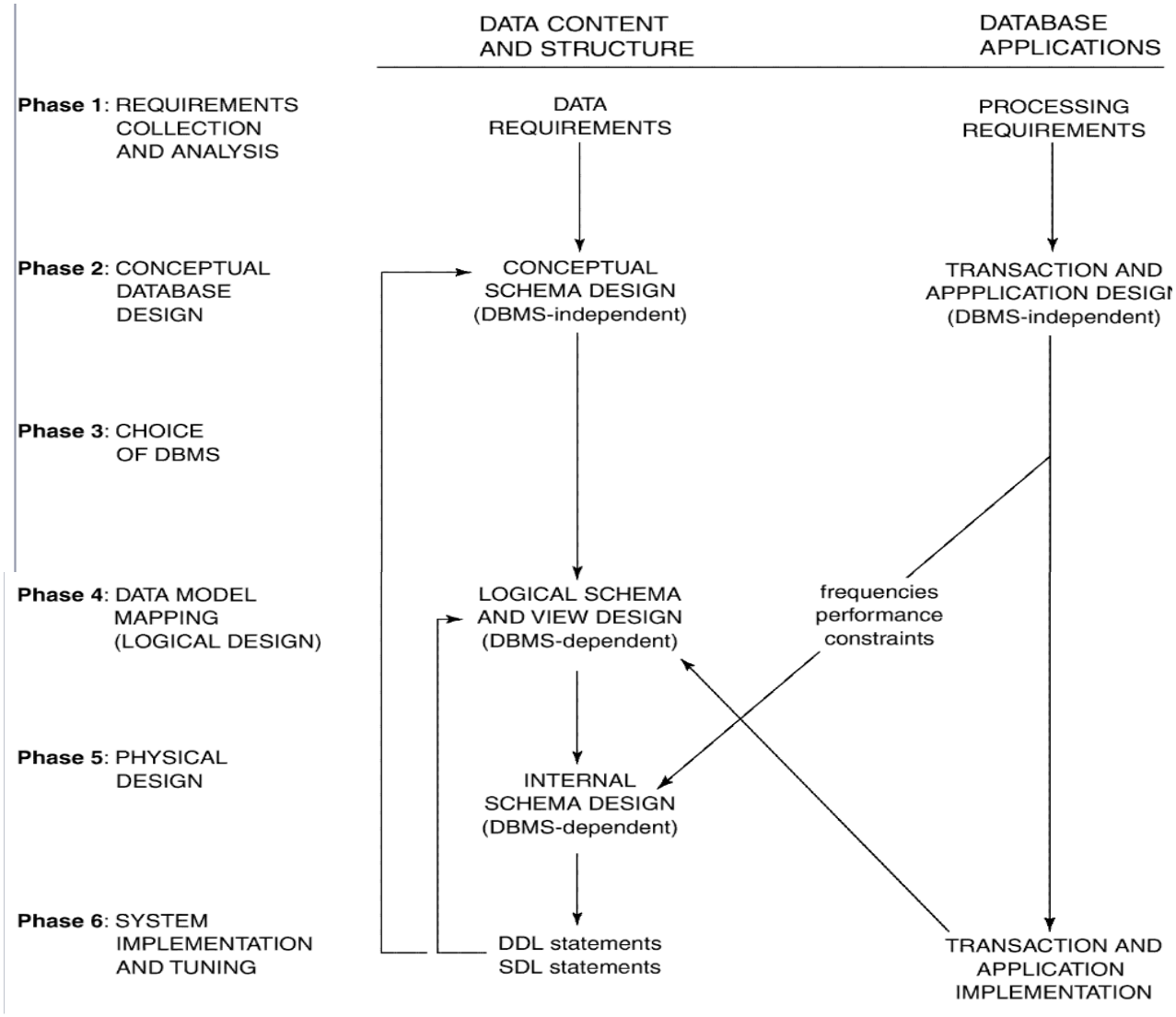


Figure (1) Phases of database design for large databases.

Among the above six phases, below are the explanations of the last two items.

- Physical database design (Phase 5): During this phase, we design the specifications for the stored database in terms of physical storage structures, record placement, and indexes. This corresponds to designing the internal schema in the terminology of the three-level DBMS architecture.
- Database system implementation and tuning (Phase 6): During this phase, the database and application programs are implemented, tested, and eventually deployed for service. Various transactions and applications are tested individually and then in conjunction with each other.

The design process consists of two parallel activities, as illustrated in Figure (1) . The first activity involves the design of the **data content and structure** of the database; the second relates to the design of **database applications.**

Traditionally, database design methodologies have primarily focused on the first of these activities whereas software design has focused on the second; this may be called **data-driven** versus **process-driven design.**

The six phases mentioned previously do not have to proceed strictly in sequence. In many cases we may have to modify the design from an earlier phase during a later phase. These **feedback loops** among phases—and also within phases—are common.

Schema Design

Given a set of requirements, we must create a conceptual schema that satisfies these requirements. There are various strategies for designing such a schema.
We now discuss some of these strategies:

1. *Top-down strategy:* For example, we may specify only a few high-level entity types and then, as we specify their attributes.
2. *Bottom-up strategy:* For example, we may start with the attributes and group these into entity types and relationships.
3. *Mixed strategy:* Instead of following any particular strategy throughout the design, parts of the schema are designed according to a top-down or bottom-up strategies. The various schema parts are then combined.

Transaction Design

The purpose of ***Transaction Design*** phase, which proceeds in parallel with ***Schema Design*** phase, is to design the characteristics of known database transactions (applications) in a DBMS-independent way. When a database system is being designed, the designers are aware of many known applications (or transactions) that will run on the database once it is implemented. An important part of database design is to specify the functional characteristics of these transactions early on in the design process. This ensures that the database schema will include all the information required by these transactions.

A common technique for specifying transactions at a conceptual level is to identify their **input/output** and **functional behavior.** By specifying the input and output parameters (arguments), and internal functional flow of control, designers can specify a transaction in a conceptual and system-independent way.

Transactions usually can be grouped into three categories:

1. **Retrieval transactions,** which are used to retrieve data for display on a screen or for production of a report;
2. **Update transactions,** which are used to enter new data or to modify existing data in the database;
3. **Mixed transactions,** which are used for more complex applications that do some retrieval and some update.

For example, consider an airline reservations database. A retrieval transaction could list all morning flights on a given date between two cities. An update transaction could be to book a seat on a particular flight. A mixed transaction may first display some data, such as showing a customer reservation on some flight, and then update the database, such as canceling the reservation by deleting it, or by adding a flight segment to an existing reservation.

Transactions, Read and Write Operations, and DBMS Buffers

A **transaction** is a logical unit of database processing that includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.

The DBMS will generally maintain a number of **buffers** in main memory that hold database disk blocks containing the database items being processed. When these buffers are all occupied, and additional database blocks must be copied into memory, some buffer replacement policy is used to choose which of the current buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused

Concurrency control and recovery mechanisms are mainly concerned with the database access commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database

Desirable Properties of Transactions

Transactions should possess several properties. These are often called the **ACID properties,** and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Concurrency control techniques are used to ensure the noninterference or isolation property of concurrently executing transactions.

Most of these techniques ensure serializability of schedules using **protocols** (that is, sets of rules) that guarantee serializability. One important set of protocols employs the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently;

Locking Techniques for Concurrency Control

A user writes data access/update programs in terms of the high-level query and update language supported by the DBMS. To understand how the DBMS handles such requests, with respect to concurrency control and recovery, it is convenient to regard an execution of a user program, or transaction, as a series of reads and writes of database objects:

To read a database object, it is first brought into main memory (specifically, some frame in the buffer pool) from disk, and then its value is copied into a program variable. To write a database object, an in-memory copy of the object is first modified and then written to disk.

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.

Binary Locks

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation `lock_item`(*X*) before any `read_item`(*X*) or `write_item`(*X*) operations are performed in T.
2. A transaction T must issue the operation `unlock_item`(*X*) after all `read_item`(*X*) and `write_item`(*X*) operations are completed in T.
3. A transaction T will not issue a `lock_item`(*X*) operation if it already holds the lock on item *X* .
4. A transaction T will not issue an `unlock_item`(*X*) operation unless it already holds the lock on item *X*.

These rules can be enforced by the lock manager module of the DBMS. Between the `lock_item`(*X*) and `unlock_item`(*X*) operations in transaction T, T is said to **hold the lock** on item *X*. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

Other Concurrency Control Issues

In the concurrency control techniques we have discussed so far, a certain degree of checking is done *before* a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

There are three phases for this concurrency control protocol:

1. **Read phase:** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase:** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase:** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

Database Tuning

The performance of a DBMS on commonly asked queries and typical update operations is the ultimate measure of a database design. A DBA can improve performance by adjusting some DBMS parameters (e.g., the size of the buffer pool) and by identifying performance bottlenecks and adding hardware to eliminate such bottlenecks. Tuning must be guided by the nature of the data and its intended use. In particular, it is important to understand the typical workload that the database must support; the workload consists of a mix of queries and updates.

Database Workloads

The key to good physical design is arriving at an accurate description of the expected workload. A workload description includes the following elements:

1. A list of queries and their frequencies, as a fraction of all queries and updates.
2. A list of updates and their frequencies.
3. Performance goals for each type of query and update.

Tuning Aspects

Tuning a database involves dealing with the following types of problems:

- How to avoid excessive lock contention, thereby increasing concurrency among transactions.
- How to minimize overheard of logging and unnecessary dumping of data.
- How to optimize buffer size and scheduling of processes.
- How to allocate resources such as disks, RAM, and processes for most efficient utilization.

The goals of tuning are as follows:

- To make applications run faster.
- To lower the response time of queries/transactions.
- To improve the overall throughput of transactions.

Most of the previously mentioned problems can be solved by setting appropriate physical DBMS parameters. The DBAs are typically trained to handle these problems of tuning for the specific DBMS.

We briefly discuss the tuning of various physical database design decisions below.

Tuning the Database Design

We might need for a possible denormalization, which is a departure from keeping all tables as BCNF relations. If a given physical database design does not meet the expected objectives, we may revert to the logical database design, make adjustments to the logical schema, and remap it to a new set of physical tables and indexes.

The entire database design has to be driven by the processing requirements as much as by data requirements. If the processing requirements are dynamically changing, the design needs to respond by making changes to the conceptual schema if necessary and to reflect those changes into the logical schema and physical design. These changes may be of the following nature:

- Existing tables may be joined (denormalized) because certain attributes from two or more tables are frequently needed together: This reduces the normalization level from BCNF to 3NF, 2NF, or 1NF
- For the given set of tables, there may be alternative design choices, all of which achieve 3NF or BCNF. One may be replaced by the other.
- A relation of the form R(K, A, B, C, D, ...)—that is in BCNF can be stored into multiple tables that are also in BCNF—for example, R1(K, A, B), R2(K, C, D,), R3(K, ...)—by replicating the key K in each table. Each table contains the sets of attributes that are accessed together.

  For example, the table EMPLOYEE(SSN, Name, Phone, Grade, Salary) may be split into two tables EMP1(SSN, Name, Phone) and EMP2(SSN, Grade, Salary). If the original table had a very large number of rows (say 100,000) and queries about phone numbers and salary information are totally distinct, this separation of tables may work better. This is also called **vertical partitioning.**

- Just as vertical partitioning splits a table vertically into multiple tables, **horizontal partitioning** takes horizontal slices of a table and stores them as distinct tables.

  For example, product sales data may be separated into ten tables based on ten product lines. Each table has the same set of columns (attributes) but contains a distinct set of products (tuples). If a query or transaction applies to all product data, it may have to run against all the tables and the results may have to be combined.

- Attribute(s) from one table may be repeated in another even though this creates redundancy and a potential anomaly.

  For example, Partname may be replicated in tables wherever the Part# appears (as foreign key), but there may be one master table called PART_MASTER(Part#, Partname, . . .) where the Partname is guaranteed to be up-to-date.

Database Design Theory and Methodology

Dr. Mehdi Duaimi

The relational model for database management is a database model based on predicate logic and set theory. It was first formulated and proposed in 1969 by Edgar Codd with aims to express database queries and enforce database integrity constraints.

The relational model of data permits the database designer to create a consistent, logical representation of information. Consistency is achieved by including declared constraints in the database design, which is usually referred to as the logical schema. The theory includes a process of *database normalization* whereby a design with certain desirable properties can be selected from a set of logically equivalent alternatives.

Database normalization is a technique for designing relational tables to minimize duplication of information and to safeguard the database against certain types of logical or structural problems, namely data anomalies. A table that is sufficiently normalized is less vulnerable to problems of this kind. Database theory describes a table's degree of normalization in terms of normal forms.

The Concept of Anomalies

The intention of relational database theory is to eliminate anomalies from occurring in a database. Anomalies can potentially occur during changes to a database. An anomaly is a bad thing because data can become logically corrupted.

Now consider a table of students, their majors, and their departmental advisors. Let us assume that each student has one department, and each department has one advisor.

STUDENT	ADVISOR	DEPARTMENT
'Higgins'	' Celko'	' Comp Sci'
'Jones'	' Celko'	' Comp Sci'
'Wilson'	' Smith'	'English'

***1. Insertion anomaly*:** It may not be possible to store some information unless some other information is stored as well. When you try to insert a new student into the English department, you also have to know that Dr. Smith is the departmental advisor or you cannot insert the new row.

***2. Update anomaly*:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated. Higgins decides that Dr. Celko's database course is too hard and switches her major to Sanskrit. Unfortunately, this creates the row ('Higgins', 'Celko', 'Sanskrit'), which contains the false fact that Dr. Celko is the advisor for the Sanskrit department.

3. Deletion anomaly: It may not be possible to delete some information without losing some other information as well. If Dr. Smith gives up his position in the English department, then deleting his row will also destroy the fact that Wilson was an English major. We can assume school policy is not to expel all the students when an advisor leaves. This table really should be two tables, one for students and one for advisors. The process

Keys

Before proceeding further, let us look again at the definitions of keys of a relation schema from . A **superkey** of a relation schema $R = \{A_1, A_2, ..., A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A **key** K is a superkey with the *additional property* that removal of any attribute from K will cause K not to be a superkey any more. The difference between a key and a superkey is that a key has to be *minimal;* that is, if we have a key $K = \{A_1, A_2, ..., A_k\}$ of R, then $K - \{A_i\}$ is *not a key* of R for any i, $1 \le i \le k$. In Figure (1) {SSN} is a key for EMPLOYEE, whereas {SSN}, {SSN, ENAME}, {SSN, ENAME, BDATE}, etc. are all superkeys.

If a relation schema has more than one key, each is called a **candidate key.** One of the candidate keys is *arbitrarily* designated to be the **primary key,** and the others are called secondary keys. Each relation schema must have a primary key. In Figure (1) {SSN} is the only candidate key for EMPLOYEE, so it is also the primary key.

An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key. In Figure (1) both SSN and PNUMBER are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

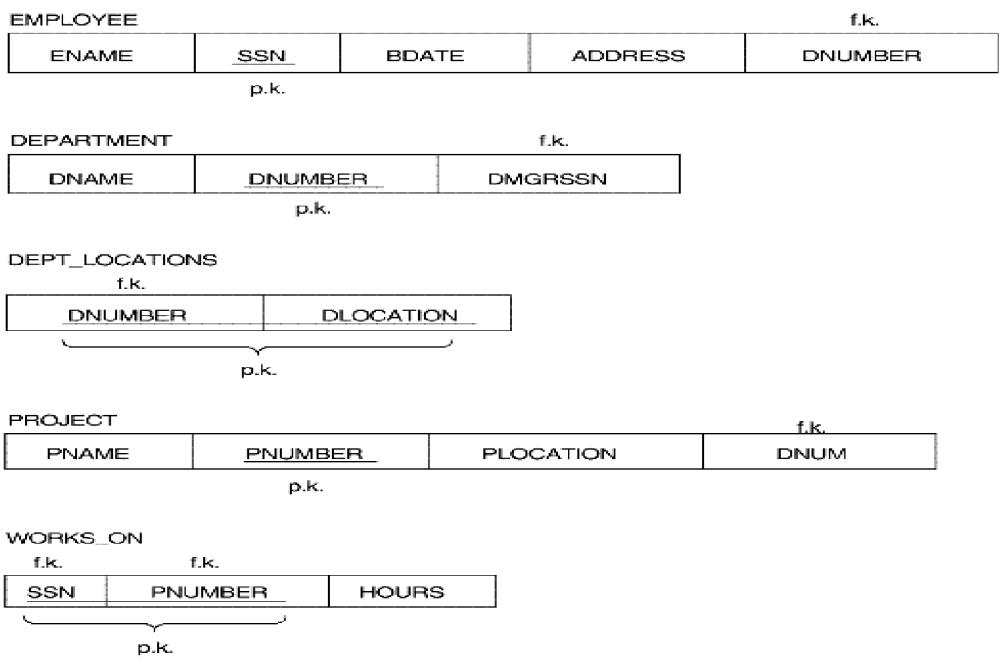


Figure (1) Simplified version of the COMPANY relational database schema.

Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational schema R has *n* attributes:

$$R = (A_1, A_2, \ldots, A_n)$$

A **functional dependency,** denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint*:

For any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, we must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are *determined by*, the values of the X component; or alternatively, the values of the X component of a tuple uniquely (or **functionally**) **determine** the values of the Y component.

Functional dependency is a property of the **semantics** or **meaning of the attributes.** The database designers will use their understanding of the semantics of the attributes of R to specify the functional dependencies that should hold on *all* relation states (extensions) r of R.

Consider the relation schema EMP_PROJ in Figure (2); from the semantics of the attributes, we know that the following functional dependencies should hold:

a. SSN→ ENAME

b. PNUMBER→ {PNAME, PLOCATION}

c. {SSN, PNUMBER}→HOURS

These functional dependencies specify that (a) the value of an employee's social security number (SSN) uniquely determines the employee name (ENAME), (b) the value of a project's number (PNUMBER) uniquely determines the project name (PNAME) and location (PLOCATION), and (c) a combination of SSN and PNUMBER values uniquely determines the number of hours the employee works on the project per week (HOURS). Alternatively, we say that ENAME is functionally determined by (or functionally dependent on) SSN, or "given a value of SSN, we know the value of ENAME," and so on.

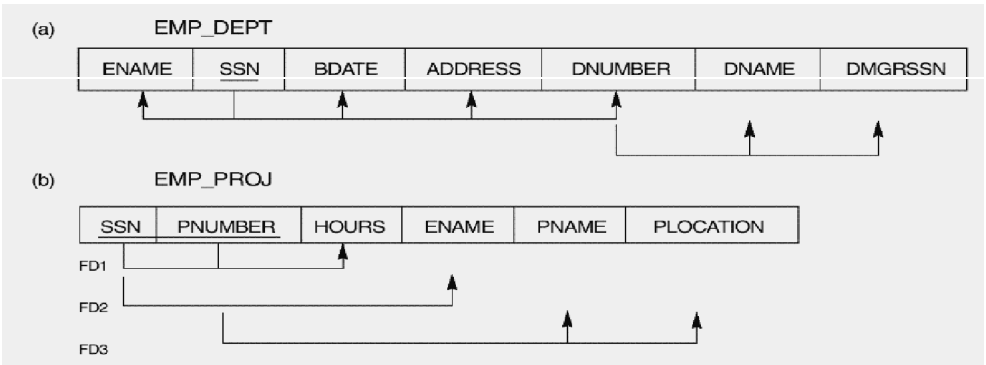
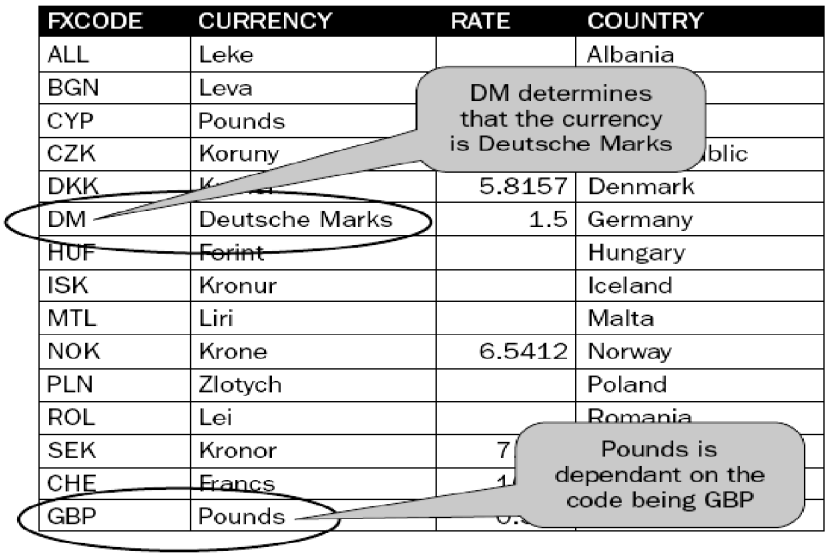


Figure (2) Two relation schemas and their functional dependencies. Both suffer from update anomalies. (a) The EMP_DEPT relation schema. (b) The EMP_PROJ relation schema.

A functional dependency is a rule that has to hold for all possible values that could ever be in the table. For example, A → B can be read as "A determines B". If we know an employee number, we can determine his name; and so forth. A is the determinant—because A determines the value of B.



FXCODE	CURRENCY	RATE	COUNTRY
ALL	Leke		Albania
BGN	Leva		
CYP	Pounds		
CZK	Koruny		blic
DKK	K	5.8157	Denmark
DM	Deutsche Marks	1.5	Germany
HUF	Forint		Hungary
ISK	Kronur		Iceland
MTL	Liri		Malta
NOK	Krone	6.5412	Norway
PLN	Zlotych		Poland
ROL	Lei		Romania
SEK	Kronor	7	
CHE	Francs	1	
GBP	Pounds		

Figure (3) Functional dependency and the determinant.

- **Trivial functional dependency**: A trivial functional dependency is a functional dependency of an attribute on a superset of itself. {Emp_ID, Emp_Address} → {Emp_Address} is trivial, as is {Emp_Address} → {Emp_Address}.
- **Full functional dependency**: An attribute is fully functionally dependent on a set of attributes X if it is:

a) functionally dependent on X, and

b) not functionally dependent on any proper subset of X.

Full functional dependence—This situation occurs where X determines Y, but X combined with Z does not determine Y. In other words, Y depends on X and X alone. If Y depends on X with anything else, there is not full functional dependence. Essentially X, the determinant, cannot be a composite key. A composite key contains more than one field (the equivalent of X with Z). Figure (4) shows that POPULATION is dependent on COUNTRY but not on the combination of RATE and COUNTRY. Therefore, there is full functional dependency between OPULATION and COUNTRY because RATE is irrelevant to POPULATION. Conversely, there is not full functional dependence between POPULATION and the combination of COUNTRY and RATE.

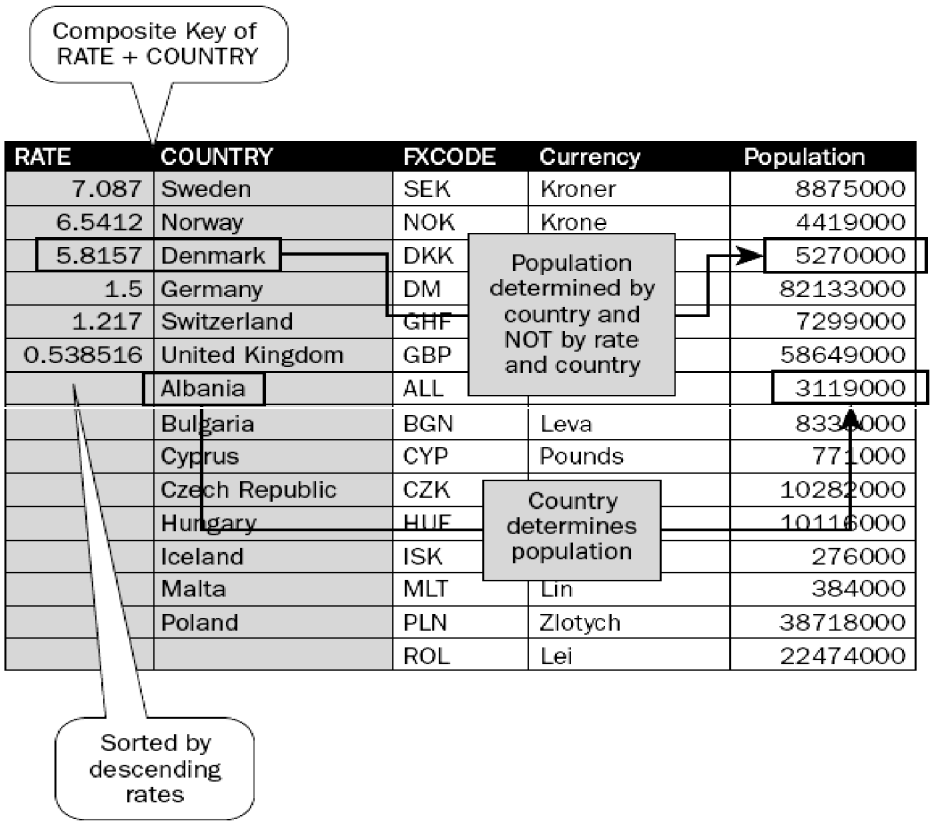


Figure (4) Full functional dependence.

Lec. 4

Inference Rules for Functional Dependencies

We denote by F the set of functional dependencies that are specified on relation schema R. Typically, the schema designer specifies the functional dependencies that are *semantically obvious;* usually, however, numerous other dependencies can be *inferred* or *deduced* from the FDs in F. The set of all such dependencies is called the **closure** of F and is denoted by F^+. For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema of Figure (2) :

$F =$ {SSN → {ENAME, BDATE, ADDRESS, DNUMBER},
DNUMBER → {DNAME,DMGRSSN}}

We can *infer* the following additional functional dependencies from F:

SSN → {DNAME, DMGRSSN},
SSN → SSN,
DNUMBER → DNAME.

The closure F^+ of F is the set of all functional dependencies that can be inferred from F. To determine a systematic way to infer dependencies, we must discover a set of **inference rules** that can be used to infer new dependencies from a given set of dependencies.
We use the notation $F \models X \rightarrow Y$ to denote that the functional dependency $X \rightarrow Y$ is inferred from the set of functional dependencies F.

The following six rules (IR1 through IR6) are well-known inference rules for functional dependencies:

IR1 (reflexive rule): If X $\supseteq$ Y, then $X \rightarrow Y$.

IR2 (augmentation rule): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

IR3 (transitive rule): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

IR4 (decomposition, or projective, rule): $\{X \rightarrow YZ\} \models X \rightarrow Y$.

IR5 (union, or additive, rule): $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.

IR6 (pseudotransitive rule): $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

Formally, a functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial.**

The set of dependencies F^+, which we called the closure of F, can be determined from F by using only inference rules IR1 through IR3. Inference rules IR1 through IR3 are known as **Armstrong's inference rules.**

Typically, database designers first specify the set of functional dependencies F that can easily be determined from the semantics of the attributes of R; then IR1, IR2, and IR3 are used to infer additional functional dependencies that will also hold on R. A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left-hand side of some functional dependency in F and then to determine the set of *all attributes* that are dependent on X. Thus for each such set of attributes X, we determine the set X^+ of attributes that are functionally determined by X based on F; X^+ is called the **closure of X under F.** Algorithm (1) can be used to calculate X^+.

Algorithm (1) Determining X^+, the closure of X under F

$X^+ := X$;
repeat
    old$X^+ := X^+$;
    for each functional dependency $Y \rightarrow Z$ in F do
    if $X^+ \supseteq Y$ then $X^+ := X^+ \cup Z$;
until (X^+ = oldX^+);

Algorithm (1) starts by setting X^+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. Using inference rules IR3 and IR4, we add attributes to X^+, using each functional dependency in F. We keep going through all the dependencies in F (the repeat loop) until no more attributes are added to X^+ *during a complete cycle* (the for loop) through the dependencies in F. For example, consider the relation schema EMP_PROJ in Figure (2); from the semantics of the attributes, we specify the following set F of functional dependencies that should hold on EMP_PROJ:

F = {SSN → ENAME,
PNUMBER → {PNAME, PLOCATION},
{SSN, PNUMBER} → HOURS}

Using Algorithm (1), we calculate the following closure sets with respect to F:

{ SSN }$^+$ = { SSN, ENAME }

{ PNUMBER }$^+$ = { PNUMBER, PNAME, PLOCATION }

{ SSN, PNUMBER }$^+$ = { SSN, PNUMBER, ENAME, PNAME, PLOCATION, HOURS }

Normalization

Dr. Mehdi Duaimi

is the sequence of steps by which a relational database model is both created and improved upon. The sequence of steps involved in the normalization process is called Normal Forms. Essentially, Normal Forms applied during a process of normalization allow creation of a relational database model as a step-by-step progression.

Defining Normal Forms:

The following are the precise definitions of Normal Forms:

1. ***1st Normal Form*** **(1NF)**—Eliminate repeating groups such that all records in all tables can be identified uniquely by a primary key in each table. In other words, all fields other than the primary key must depend on the primary key. 1st Normal Form (1NF)—Removes repeating fields by creating a new table where the original and new table are linked together with a master-detail, one-to-many relationship.
2. ***2nd Normal Form*** **(2NF)**—All non-key values must be fully functionally dependent on the primary key. No partial dependencies are allowed. A partial dependency exists when a field is fully dependent on a part of a composite primary key. 2nd Normal Form (2NF)—Performs a seemingly similar function to that of 1NF, but creates a table where repeating values (rather than repeating fields as for 1NF) are removed to a new table. The result is a many-to-one relationship rather than a one-to-many relationship, created between the original and the new tables. The new table gets a primary key consisting of a single field. The master table contains a foreign key pointing back to the primary key of the new table. That foreign key is not part of the primary key in the original table.
3. ***3rd Normal Form*** **(3NF)**—Eliminate transitive dependencies, meaning that a field is indirectly determined by the primary key. This is because the field is functionally dependent on another field, whereas the other field is dependent on the primary key. Elimination of a transitive dependency implies creation of a new table for something indirectly dependent on the primary key in an existing table.

Many modern-day commercial relational database implementations do not go beyond the implementation of 3NF. This is often true of OLTP[1] databases and nearly always true in properly designed data warehouse databases. Application of Normal Forms beyond that of 3NF tends to produce too many tables, resulting in too many tables in SQL joins. Bigger joins result in poor performance. In general, good performance is much more important than granular perfection in relational database design. In a perfect world, most relational database model designs are very similar.

As a result, much of the basic database design for many applications from accounting to manufacturing (and anything else you can think of) is all more or less the same.

The normal forms based on FDs are first normal form (1NF), second normal form (2NF), third normal form (3NF), and Boyce-Codd normal form (BCNF)—to be explained later. These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in first normal form if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our definition of the relational model. Although some of the newer database systems are relaxing this requirement, in this context we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

Consider the DEPARTMENT relation schema shown in Figure (1) (b) , whose primary key is DNUMBER. As we can see; it is not in 1NF because DLOCATIONS is not an atomic attribute, as illustrated by the first tuple in Figure (1) (b).

(a)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS

(b)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure (1) Normalization into 1NF. (a) Relation schema that is not in 1NF. (b) Example relation instance. (c) 1NF relation with redundancy.

The main technique to achieve first normal form for such a relation is by:
Removing the attribute DLOCATIONS that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key DNUMBER of DEPARTMENT. The primary key of this relation is the combination {DNUMBER, DLOCATION}, as shown in Figure (2). A distinct tuple in DEPT_LOCATIONS exists for ***each location*** of a department. This decomposes the non-1NF relation into two 1NF relations.

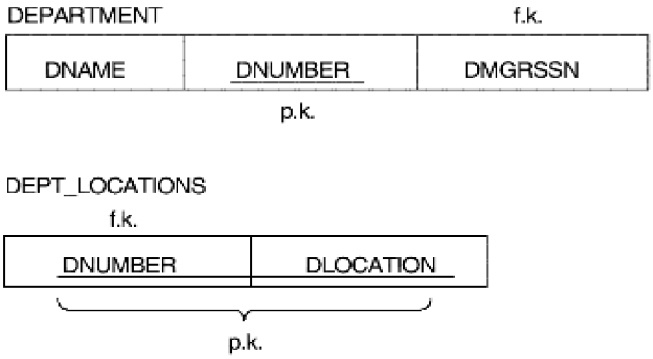


Figure (2) two relations in 1NF

The first normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it.* Figure (3) shows how the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS(PNUMBER, HOURS) ***within each tuple*** represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ(SSN, ENAME, {PROJS(PNUMBER, HOURS)})

(a)

EMP_PROJ

SSN	ENAME	PROJS	
		PNUMBER	HOURS

(b)

EMP_PROJ

SSN	ENAME	PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
666884444	Narayan,Ramesh K.	3	40.0
453453453	English,Joyce A.	1	20.0
		2	20.0
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya,Alicia J.	30	30.0
		10	10.0
987987987	Jabbar,Ahmad V.	10	35.0
		30	5.0
987654321	Wallace,Jennifer S.	30	20.0
		20	15.0
888665555	Borg,James E.	20	null

(c)

EMP_PROJ1

SSN	ENAME

EMP_PROJ2

SSN	PNUMBER	HOURS

Figure (3) Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a "nested relation" PROJS. (b) Example extension of the EMP_ PROJ relation showing nested relations within each tuple. (c) Decomposing EMP_PROJ into 1NF relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

Notice that SSN is the primary key of the EMP_PROJ relation in Figure (3) (a) and Figure (3) (b), while PNUMBER is the partial primary key of the nested relation; that is, within each tuple, the nested relation must have unique values of PNUMBER. To normalize this into 1NF, we remove the nested relation attributes into a new relation and propagate the primary key into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2 shown in Figure (3) (c).

Lec. 6

Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ *does not* functionally determine Y.

A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$.

In Figure (4) (b), {SSN, PNUMBER} → HOURS is a full dependency (neither SSN → HOURS nor PNUMBER → HOURS holds). However, the dependency {SSN, PNUMBER} → ENAME is partial because SSN → ENAME holds.

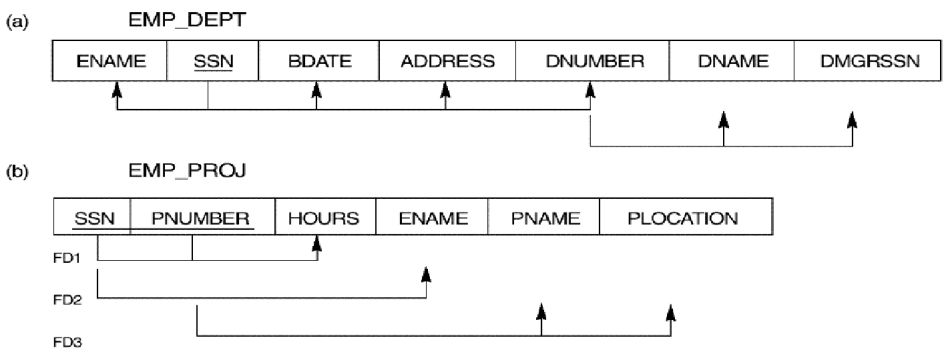


Figure (4) Two relation schemas and their functional dependencies. (a) The EMP_DEPT relation schema. (b) The EMP_PROJ relation schema.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. A relation schema R is in **2NF** if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R. The EMP_PROJ relation in Figure (4) (b) is in 1NF but is not in 2NF. The nonprime attribute ENAME violates 2NF because of FD2, as do the nonprime attributes PNAME and PLOCATION because of FD3. The functional dependencies FD2 and FD3

make ENAME, PNAME, and PLOCATION partially dependent on the primary key {SSN, PNUMBER} of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be "second normalized" or "2NF normalized" into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. The functional dependencies FD1, FD2, and FD3 in Figure (4) (b) hence lead to the decomposition of **EMP_PROJ** into the three relation schemas EP1, EP2, and EP3 shown in Figure (5) (a); Each of which is in 2NF.

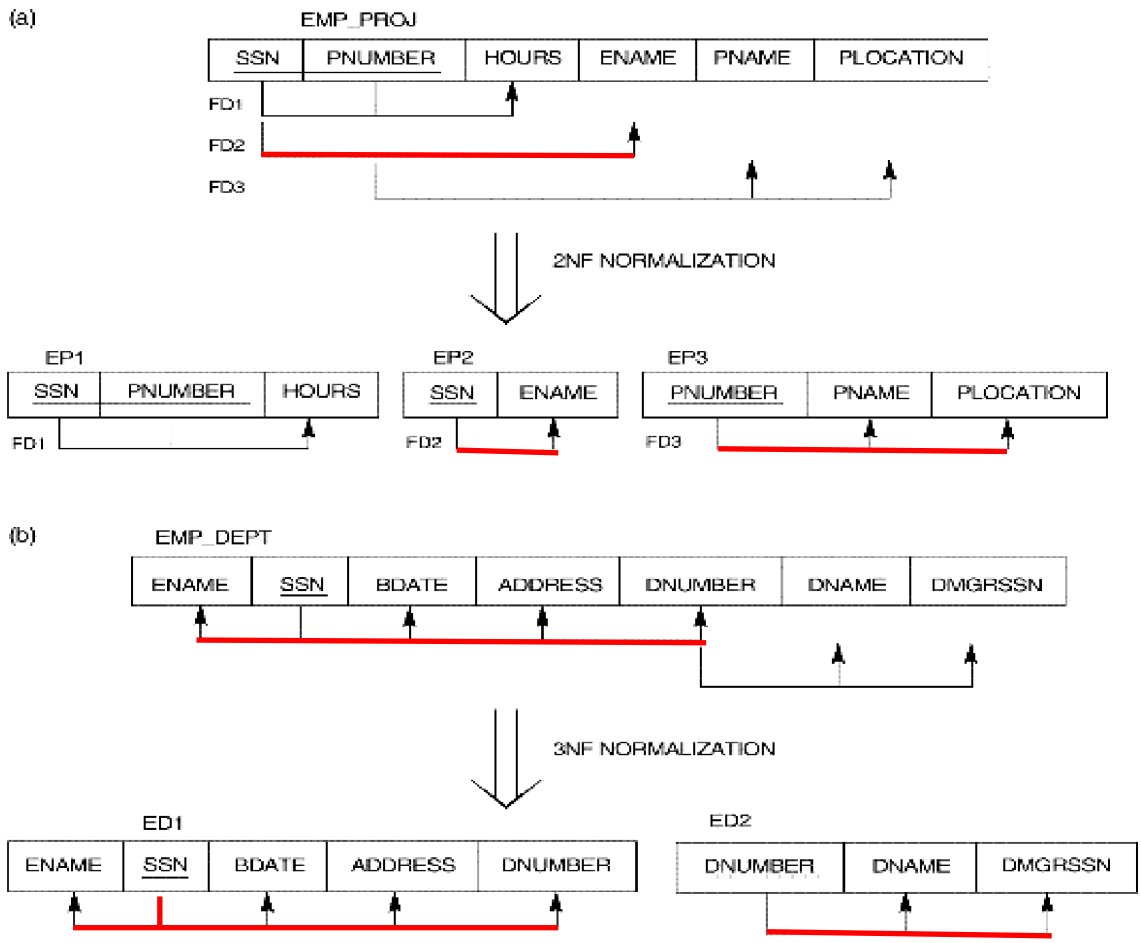


Figure (5) The normalization process. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency SSN $\rightarrow$ DMGRSSN is transitive through DNUMBER **in** EMP_DEPT of Figure (4) (a) because both the dependencies SSN $\rightarrow$ DNUMBER and DNUMBER $\rightarrow$ DMGRSSN hold *and*

DNUMBER is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of DMGRSSN on DNUMBER is undesirable in EMP_DEPT since DNUMBER is not a key of EMP_DEPT.

According to Codd's original definition, a relation schema *R* is in **3NF** if it satisfies 2NF *and* no nonprime attribute of *R* is transitively dependent on the primary key. The relation schema EMP_DEPT in Figure (4) (a) is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of DMGRSSN (and also DNAME) on SSN via DNUMBER. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure (5) (b). Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments.

A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Table (1) informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding "remedy" or normalization to achieve the normal form.

Table (1) Summary of Normal Forms Based on Primary Keys and Corresponding Normalization.

Normal Form	**Test**	**Remedy (Normalization)**
First (1NF)	Relation should have no nonatomic attributes or nested relations	Form new relations for each nonatomic attribute or nested relation
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes.) That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

[1] OLTP ➔ Online Transaction Processing

DATABASE RECOVERY *Dr. Mehdi Duaimi*

The DBMS supports concurrency and recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the transaction manager, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the lock manager, which keeps track of requests for locks and grants locks on database objects when they become available; and the recovery manager, which is responsible for maintaining a log, and restoring the system to a consistent state after a crash.

RECOVERY MANAGER

The recovery manager of a DBMS is responsible for ensuring two important properties of transactions: atomicity and durability. It ensures atomicity by undoing the actions of transactions that do not commit and durability by making sure that all actions of committed transactions survive system crashes, (e.g., a core dump caused by a bus error) and media failures (e.g., a disk is corrupted). The recovery manager is one of the hardest components of a DBMS to design and implement. It must deal with a wide variety of database states because it is called on during system failures.

RECOVERY ALGORITHM

When the recovery manager is invoked after a crash, restart proceeds in three phases:

1. Analysis: Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash.
2. Redo: Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.
3. Undo: Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

The Log

The log, sometimes called the trail or journal, is a history of actions executed by the DBMS. Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes; this durability can be achieved by maintaining two or more copies of the log on different disks (perhaps in different locations), so that the chance of all copies of the log being simultaneously lost is negligibly small.

The most recent portion of the log, called the log tail, is kept in main memory and is periodically forced to stable storage. This way, log records and data records are written to disk at the same granularity (pages or sets of pages).

Every log record is given a unique id called the log sequence number (LSN). As with any record id, we can fetch a log record with one disk access given the LSN. Further, LSNs should be assigned in monotonically increasing order; this property is required for the recovery algorithm. If the log is a sequential file, in principle growing indefinitely, the LSN can simply be the address of the first byte of the log record. For recovery purposes, every page in the database contains the LSN of the most recent log record that describes a change to this page. This LSN is called the pageLSN.

A log record is written for each of the following actions:

- *Updating a page*: After modifying the page, an update type record is appended to the log tail. The pageLSN of the page is then set to the LSN of the update log record. (The page must be pinned in the buffer pool while these actions are carried out.)
- *Commit*: When a transaction decides to commit, it force-writes a commit type log record containing the transaction id. That is, the log record is appended to the log, and the log tail is written to stable storage, up to and including the commit record. The transaction is considered to have committed at the instant that its commit log record is written to stable storage. (Some additional steps must be taken, e.g., removing the transaction's entry in the transaction table; these follow the writing of the commit log record.)
- *Abort*: When a transaction is aborted, an abort type log record containing the transaction id is appended to the log, and Undo is initiated for this transaction.
- *End*: As noted above, when a transaction is aborted or committed, some additional actions must be taken beyond writing the abort or commit log record. After all these additional steps are completed, an end type log record containing the transaction id is appended to the log.
- *Undoing an update:* When a transaction is rolled back (because the transaction is aborted, or during recovery from a crash), its updates are undone. When the action described by an update log record is undone, a compensation log record, or CLR, is written.

Consider the simple execution history illustrated in Figure (1).

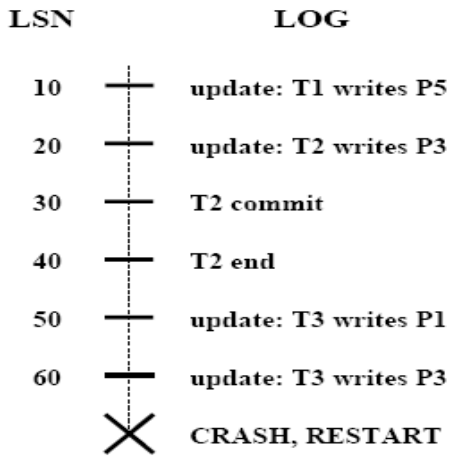


Figure (1) Execution History with a Crash

When the system is restarted, the Analysis phase identifies T1 and T3 as transactions that were active at the time of the crash, and therefore to be undone; T2 as a committed transaction, and all its actions, therefore, to be written to disk; and P1, P3, and P5 as potentially dirty pages. All the updates (including those of T1 and T3) are reapplied in the order shown during the Redo phase. Finally, the actions of T1 and T3 are undone in reverse order during the Undo phase; that is, T3's write of P3 is undone, T3's write of P1 is undone, and then T1's write of P5 is undone.

Semistructured Data

The semistructured-data model plays a special role in database systems:

1. It serves as a model suitable for integration of databases, that is, for describing the data contained in two or more databases that contain similar data with different schemas.

2. It serves as a document model in notations such as XML, that are being used to share information on the Web.

A database of semistructured data is a collection of nodes. Each node is either a leaf or interior. Leaf nodes have associated data;

Lec. 8

XML and Its Data Model

XML (Extensible Markup Language) is a tag-based notation for "marking" documents, much like the familiar HTML or less familiar SGML (Standard Generalized Markup Language). XML tags talk about the meaning of substrings within the document. DTD ("document type definition"), which is a flexible form of schema that we can place on certain documents with XML tags.

Semantic Tags

Tags in XML are text surrounded by triangular brackets, i.e., <...>, as in HMTL. Also as in HTML, tags generally come in matching pairs, with a beginning tag like <F00> and a matching ending tag that is the same word with a slash, like </F00>.

XML is designed to be used in two somewhat different modes:

1. Well-formed XML allows you to invent your own tags,
2. Valid XML involves a Document Type Definition (DTD) that specifies the allowable tags.

This form of XML is intermediate between the relational databases. In order for a computer to process XML documents automatically, It needs to be something like a schema for the documents. The description of the schema is given by a grammar-like set of rules, called a document type definition, or DTD.

Because native XML databases store data in a different way from relational databases, there are different languages used to query and update the database:

XML native databases:

- queries are specified using the XPath language
- updates are defined using the XUpdate language

Providing Access to Databases on the World Wide Web

Today's technology has been moving rapidly from static to dynamic Web pages. The Web server uses a standard interface called the Common Gateway Interface (CGI) to act as the middleware—the additional software layer between the user interface front-end and the DBMS back-end that facilitates access to heterogeneous databases. Users access not only to file systems but to databases and DBMSs to support query processing.

The existing approaches may be divided into two categories:

1. Access using CGI scripts: The database server can be made to interact with the Web server via CGI. which are written in languages like PERL, or C.
2. Access using JDBC: JDBC is a set of Java classes developed by Sun Microsystems to allow access to relational databases through the execution of SQL statements.

Database Connectivity

Refers to mechanisms through which application programs connect and communicate with data repositories. This software is also known as database middleware.

1. **Open Database Connectivity (ODBC)**

   Probably most widely supported database connectivity interface. Allows any Windows application to access relational data sources, using SQL via standard application programming interface (API).

2. **Data Access Objects (DAO)**

   Object-oriented API used to access MS Access, MS FoxPro, and dBase databases from Visual Basic programs.

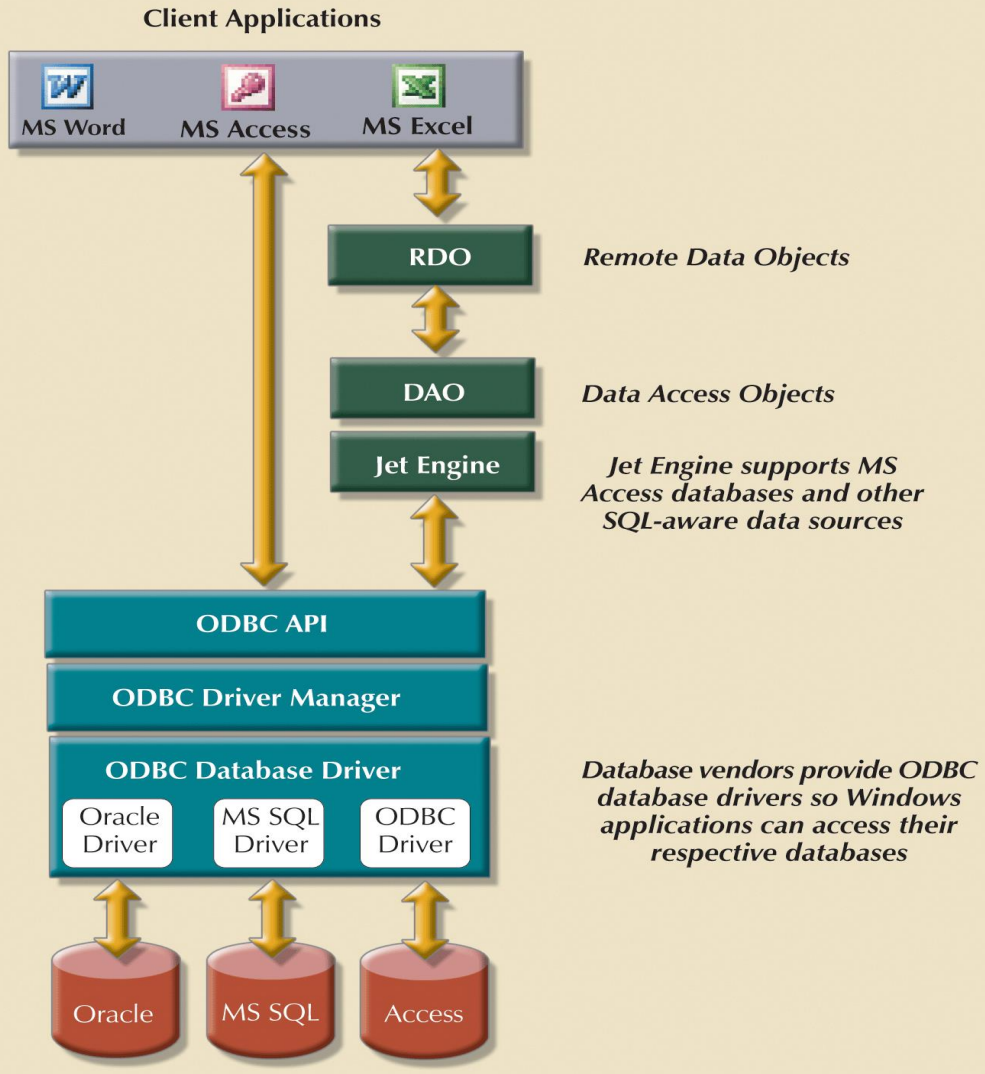
3. **Remote Data Objects (RDO)**

   Higher-level object-oriented application interface used to access remote database servers. Was optimized to deal with server-based databases, such as MS SQL Server, Oracle, and DB2.

4. **ADO.NET** is data access component of Microsoft's .NET application development framework. Introduced two new features critical for development of distributed applications: DataSets and XML support

[Type a quote

Using ODBC, DAO, and RDO to access databases



OLE-DB

Object Linking and Embedding for Database. Database middleware that adds object-oriented functionality for access to relational and nonrelational data.