1.1 Introduction

A *database* is a collection of related data. *Data* is comprised of facts that can be recorded and have implicit meaning.

The properties of a database are:

- it represents some aspect of the real world – a *miniworld* or *universe of discourse* (UoD)
- it is logically coherent and has inherent meaning
- it is designed, built and populated with data for a specific purpose

A *database management system* (DBMS) is a computerised system that enables users to create and maintain a database. It is a general-purpose software system that facilities defining, constructing, manipulating and sharing databases among users or applications.

- *defining* – specifying datatypes, structures and contraints of data. The definition is stored as a database catalogue or *metadata*.
- *constructing* – storing the data on some medium.
- *manipulating* – functions such as querying the database, updating the data and generating reports from the data.
- *sharing*

An *application program* accesses the database by sending queries/requests to the DBMS. A *query* causes some data to be retrieved, while a *transaction* causes some to be read and some to be written.

The DBMS also protects and maintains the database. *Protection* includes system as well as security protection; *maintenance* allows the DB to evolve over time.

The DB and the DBMS software are together called a *database system*.

1.2 An Example

Design of a new DB or a new application for a DB starts with *requirements specification and analysis* – the requirements are documented and converted to a *conceptual design*, from

which a DB implementation can be created (through *logical* and *physical design*).

1.3 Characteristics of the Database Approach

File processing approaches to data storage leads to redundancy, wasted storage space, and efforts to maintain consistency. In contrast, the database approach has these characteristics:

- self-describing nature
- insulation between programs and data (data abstraction)
- support of multiple views
- sharing of data and multiuser transaction processing

1.3.1 Self-Describing Nature of a DB System

The DB contains a complete definition of its structure and constraints, stored in the DBMS catalogue as metadata (some newer systems don't require metadata).

1.3.2 Insulation between Programs and Data; Data Abstraction

DBMS access programs do not need to be changed even if the structure of the file storing the data is altered. Only the catalogue's description of the file (or record) needs to be updated; the program refers to this to access the data. This is called *program-data independence*.

Some DB systems allow users to define *operations* (or functions, or methods) on data as well. The operation's *interface* (or signature) is distinct from its *implementation* (or method (don't get confused)). They are invoked by their interfaces, independent of their implementations. This is called program-operation independence.

Both the above characteristics are enabled by *data abstraction*. The DBMS gives users a *conceptual representation* of data that does not include the storage and implementation details, using the *data model*.

1.3.3 Support of Multiple Views of the Data

Different users can have different views of the data, which may be a subset of the database or contain virtual data (which is not stored but can be derived from stored data).

1.3.4 Sharing of Data and Multiuser Transaction Processing

The DBMS must include *concurrency control* software to ensure that several users updating the same data do so in a controlled manner. Applications that require this (like flight-booking systems) are called *online transaction processing* applications.

A *transaction* is an executing program or process that includes one or more DB accesses (two definitions? idk). DBMSs must exhibit:

- *isolation* (each transaction appears to execute independently from others)
- *atomicity* (either all DB operations in a transaction are executed or none are)

1.4 Actors on the Scene

The following are the people whose jobs involve the day-to-day use of a large DB.

1.4.1 Database Administrators

The DBA administers the primary (the DB itself) and the secondary (the DBMS and other softwares) resources. They authorise access to, coordinate and monitor the use of, and acquire resources for the DB.

1.4.2 Database Designers

Database designers identify the data to be stored in the DB and choose appropriate structures to represent and store it. They develop views of the DB that meet the requirements of the groups of users.

1.4.3 End Users

End users are the people whose jobs require access to the DB for querying/updating/ generating reports. There are various types of end users:

- *naive* or *parametric end users* constantly query and update the DB through standard (or *canned*) transactions, sometimes in the form of a mobile app interface.
- *casual end users* occasionally access the DB but need different data each time. They use a sophisticated DB query interface and are typically middle- or high-level managers.
- *sophisticated end users* are those who thoroughly familiarise themselves with the DBMS facilities to implement their own applications.
- *standalone users* maintain personal DBs using program packages that provide convenient menu-based or graphical interfaces.

1.4.4 System Analysts and Application Programmers

System analysts determine the requirements of users and develop specifications for the appropriate canned transactions. Application programmers implement these specifications.

1.5 Workers behind the Scene

The following people are associated with the design, development and operation of the DBMS software and system environment.

- *DBMS system designers and implementers* deisgn and implement the DBMS modules and interfaces as a software package. DBMS modules include those for implementing the catalogue, query language processing, interface processing, accessing and buffering data, concurrency control and handling data recovery and security. The DBMS interfaces with the OS, compilers and other system software.
- *Tool developers* design and implement software packages that facilitate DB modelling and design, DB system design and improved performance. These tools

are optional packages, often purchased separately, like DB design, performance monitoring, natural language or graphical interfaces, prototyping, simulation and test data generation.
- *Operators and maintenance personnel* or system administration personnel are responsible for the actual running and maintenance of the hardware and software environment of the DB system.

1.6 Advantages of Using the DBMS Approach

1.6.1 Controlling Redundancy

When (as in file processing approaches) every user group maintains its own files, the same data is frequently repeated in many groups' versions. This redundancy leads to issues like duplication of effort, waste of storage space and possible inconsistencies in data. The DBMS approach stores each data item in only one place, which is called *data normalisation*.

Controlled redundancy, however, sometimes improves the performance by enabling the DBMS to avoid searching multiple files; this is called *denormalisation*.

1.6.2 Restricting Unauthorised Access

A DBMS should provide a *security and authorisation subsystem*, which the DBA uses to specify account restrictions. Similar controls can be applied on the DBMS software itself; for example, only the DBA staff can create accounts.

1.6.3 Providing Persistent Storage for Program Objects

This is one of the main reasons for object-oriented DB systems. It allows data structures to be saved in files, past the lifetime of the programs that created them – such objects are called *persistent*.

Traditional DB systems suffer from the *impedance mismatch problem* – the inconsistency between the PL's data structures and the DBMS's. OODB systems offer compatibility with certain PLs.

1.6.4 Providing Storage Structures and Search Techniques for Efficient Query Processing

DBMSs often use auxiliary files called *indexes* to speed up disk search for records. They also have a buffering or caching module to maintain parts of the DB in main memory buffers (distinct from the OS's facility for this).

The query processing and optimisation module of the DBMS chooses an efficient plan to execute each query based the storage structures. The DBA staff carry out physical DB design and tuning, which entails choosing which indexes to create and maintain.

1.6.5 Providing Backup and Recovery

A DBMS must have a *backup and recovery subsystem* to recover from hardware or software failures.

1.6.6 Providing Multiple User Interfaces

A DBMS should provide various interfaces, like apps (mobile users), query languages (casual users), PL interfaces (application programmers), forms and command codes (parametric users), and menu-driven or natural language interfaces (standalone users). Menu-driven or form-style interfaces come under GUIs.

1.6.7 Representing Complex Relationships among Data

A DBMS should be able to represent and define complex relationships among data, and to retrieve and update data according to these relationships.

1.6.8 Enforcing Integrity Constraints

The simplest kind of integrity constraint is assigning a datatype to each data item; a more complex one is to state a relationship between records in different files (called a *referential integrity* constraint). We can also specify the uniqueness of some values (called a *uniqueness* or *key* constraint). Such constraints are sometimes called *business rules*.

Rules that pertain to a specific data model are called inherent rules of the data model.

1.6.9 Permitting Inferencing and Actions using Rules and Triggers

Deductive database systems allow us to define deduction rules to infer new information. These can be declared as *rules*, and compiled and maintained by the DBMS.
It is possible to associate *triggers* with tables, which perform some operations to some tables when activated by an update. *Stored procedures* are more involved way to enforce rules, which are invoked when certain conditions are met.

Active database systems provide active rules, which can automatically initiate actions when certain events and conditions occur.

1.6.10 Additional Implications of Using the Database Approach

Potential for Enforcing Standards

The DBA can define and enforce standards among DB users, which facilitates communication and cooperation among them. Standards can be defined for names, formats, display formats, report structures, terminology, etc.

Reduced Application Development Time

Developing a new application using DBMS facilities takes very little time once the DB is up and running.

Flexibility

The structure of a DB can be changed without affecting the stored data and application programs.

Availability of Up-to-Date Information

A DBMS makes the DB available to all users, making updates visible instantly.

Economies of Scale

Data and applications can be consolidated, reducing the wasteful overlap between activities of data-processing personnel in different projects.

2.1 Data Models, Schemas and Instances

The DB approach provides some level of *data abstraction*, which is the suppression of details of organisation and storage, and the highlighting of essential features of the data. It allows different users to perceive data at their preferred level of detail.

A *data model* is a collection of concepts used to describe the structure of a DB (its datatypes, relationships and constraints), and it provides the means for abstraction.
The *dynamic aspect*, or *behaviour*, of a DB application is also sometimes specified in the data model. This allows the DB designer to specify a set of valid operations on the DB objects (distinct from the set of *basic data model operations*).

2.1.1 Categories of Data Models

- *High-level* or *conceptual data models* provide concepts close to the users' perception of data
- *Low-level* or *physical data models* provide concepts that describe the details of data storage on disk.
- *Representational* or *implementation data models* provide concepts understandable by end users but corresponding to the organisation of data in computer storage. These are the most frequently used models.

Some concepts specified by data models are:

An *entity* represents a real-world concept or object.
An *attribute* represents some property describing an entity.
A *relationship* among two or more entities represents an association among them.

Some representational data models are:

- the *relational data model*
- the *network* and *hierarchical data models* (legacy data models apparently?)
- the *object data model*, which is a higher-level implementation data model.

Representational data models are sometimes called *record-based data models* since they

represent data using record structures.

Physical data models present information such as record formats, record orderings and *access paths* (which are search structures that make searching efficient). An *index* is an example of an access path; it allows direct access to data using an index term or keyword.

Self-describing data models represent systems which combine the description of the data with the data values themselves; for example, XML and NOSQL.

2.1.2 Schemas, Instances and Database State

The description of a DB is called its *schema*; it is specified during DB design and not expected to change frequently. A displayed schema is called a *schema diagram*. Each object in the schema is called a *schema construct*. The schema constructs and constraints are stored as *meta-data*.

All the data in a DB at a given moment is called a *database state* or *snapshot*; it is the curent set of *instances* (or *occurrences*) in the DB. In a given state, each schema construct has its own current set of instances.

The initial state of the DB is after the DB is first *populated* or *loaded*. Every subsequent update creates a new state of the DB. The DBMS ensures that each state is a *valid state* (it satisfies the constraints specified in the schema).

The schema is sometimes called the *intension* and the DB state its *extension*.

2.2 Three-Schema Architecture and Data Independence

The three-schema architecture helps achieve and visualise the characteristics of self-description, insulation of programs and data, support of multiple user views.

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture is to separate user applications from the physical DB. Schemas can be defined at:

- the *internal level*; the internal schema describes the physical storage structure of the DB, using a physical data model.
- the *conceptual level*; the conceptual schema describes the structure of the whole DB, hiding details of physical storage. It uses a representational data model, describing entities, datatypes, relationships, operations and constraints.
- the *external* or *view level*; the *external schemas* or *user views* each describe a part of the DB that a particular user group is interested in. They are usually implemented using representational data models.

Conceptual and external schemas may be based on the conceptual and external schema designs in a high-level conceptual data model.

Most DBMSs do not separate all three levels completely, but they support the architecture to some extent.

The DBMS must transform a request specified on an external schema into one on the conceptual schema, and then into one the internal schema. In the case of data retrieval, the data must be reformatted to match the external view of the user. These transformations between levels are called *mappings*.

2.2.2 Data Independence

Data independence can be defined as the capacity to change the schema at one level of a DB system without affecting the higher-level schemas.

- *Logical data independence* is the capacity to change the conceptual schema without having to change the external schemas (or application programs). Changes to the conceptual schema might expand the DB, change the constraints, or reduce the DB (in which case external schemas referring only to remaining data should be unaltered).
  Only the view definition and the mappings need to be changed in a DBMS

supporting logical data independence; the application programs that reference external schema constructs must work as before.

- *Physical data independence* is the capacity to change the internal schema without having to change the conceptual or external schemas. The reorganisation of physical files to improve the performance of retrieval or updation should not affect the conceptual schema.

Physical data independence exists in most DBs and file environments where physical details are hidden from the user. Logical data independence is harder to achieve.

Multiple-level DBMSs must have a catalogue that includes information on how to map requests and data among various levels. Additional software is used to accomplish the mappings by referring to this information in the catalogue.

3.1 Using High-Level Conceptual Data Models for Database Design

The first step towards designing a database is *requirements collection and analysis*, in which the *data requirements* and *functional requirements* are understood and documented.
Then, a *conceptual schema* for the database is created, using a high-level conceptual data model (this does not include implementation details).
Next, the schema is implemented using a commercial DBMS – this is called *logical design* or *data model mapping*.
Lastly, in the *physical design* phase, the internal storage structures, organisations, access paths, etc. are specified.

3.2 A Sample Database Application

3.3 Entity Types, Entity Sets, Attributes, and Keys

3.3.1 Entities and Attributes

An *entity* is a thing or an object in the real world with independent existence (physical or conceptual). Each entity has *attributes*, or properties that describe it. There are several types of attributes:

- composite versus simple or atomic attributes
- single-valued versus multivalued attributes
- stored versus derived attributes
- NULL values
- complex attributes

3.3.2 Entity Types, Entity Sets, Keys, and Value Sets

An *entity type* defines a collection of entities that have the same attributes. The collection of all entities of a particular type in the DB at any point is called an *entity set*.
An *entity* type describes the *schema* or *intension* for a set of entities that have the same structure; the entity set is called the *extension* of the entity type.

A *key attribute* is one whose values are distinct for each entity in the entity set. If a set of attributes is such, they are to be grouped into a composite attribute and designated as a key attribute.
This is a constraint on all entity sets in the DB, and not just any single one.

The *value set* or *domain* of an attribute is the set of values that may be assigned to it for each individual entity.

Mathematically, an attribute A of an entity set E whose value set if V can be defined as a function $A : E \rightarrow P(V)$. The valu e of attribute A for entity e is denoted $A(e)$.
For single-valued attributes, the range of the function is restricted to singleton sets. The value set of a composite attribute is derived from those of its component attributes using the Cartesian product operation.

3.3.3 Initial Conceptual Design of the COMPANY Database

3.4 Relationship Types, Relationship Sets, Roles and Structural Constraints

3.4.1 Relationship Types, Sets and Instances

A *relationship type* among n entity types defines a set of associations among entities from these types. This *relationship set* consists of *relationship instances*, which each associate n

entities among each other.

3.4.2 Relationship Degree, Role Names, and Recursive Relationships

The *degree* of a relationship type is the number of participating entity types.

Relationships can be represented as attributes as well.

Each participating entity type has a *role name*, describing its role in the relationship. In case of relationships which have the same entity type participating in different roles (called *recursive* or *self-referencing* relationships), role names are essential.

3.4.3 Constraints on Binary Relationship Types

Cardinality ratios specify the maximum number of relationship instances that an entity can participate in for a given relationship type. Usually, N is understood to mean any number (0 or more, with no maximum).

The *participation constraint* specifies whether an entity must be related to another or not, and the minimum number of relationship instances it must participate in. Entity types can have *total* or *partial* participation in a relationship type.

3.4.4 Attributes of Relationship Types

Attributes of relationship types are similar to those of entity types.

In the case of 1:1 or 1:N relationship types, they can be assigned to one of the participating entity types.

3.5 Weak Entity Types

Weak entity types are those that do not have key attributes of their own. They are identified by being related to another entity type (called the *identifying* or *owner* type) by the *identifying*

relationship.

A weak entity type has a *partial key*, which can uniquely identify weak entities related to the same owner entity.

Weak entity types may be represented as complex attributes of their owner entity types.

3.6 Refining the ER Design for the Company Database

3.7 ER Diagrams, Naming Conventions, and Design Issues

3.7.1 Summary of Notation for ER Diagrams

3.7.2 Proper Naming of Schema Constructs

3.7.3 Design Choices for ER Conceptual Design

3.7.4 Alternative Notations for ER Diagrams

3.8 Example of Other Notation: UML Class Diagrams

3.9 Relationship Types of Degree

Higher than Two

3.9.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

3.9.2 Constraints on Ternary (or Higher-Degree) Relationships

3.10 Another Example: A UNIVERSITY Database

3.11 Summary

4.1 Subclasses, Superclasses and Inheritance

In many cases an entity type has a number of subcategories that need to be represented. These are called *subclasses* or *subtypes* of the entity type (which is the *superclass* or *supertype*). The relationship between a superclass and any of its subclasses is a *class/subclass* or *superclass/subclass relationship*.
The subclasses need not be either mutually exclusive or exhaustive.

The property of *type inheritance* allows entities of the subclass to have all the attributes of its superclass.

4.2 Specialisation and Generalisation

4.2.1 Specialisation

Specialisation is the process of defining a set of subclasses of an entity type.

4.2.2 Generalisation

This is the reverse process, wherein we suppress the differences among several entity types and generalise them into a single superclass, making them subclasses.

4.3 Constraints and Characteristics of Specialisation and Generalisation

4.3.1 Constraints on Specialisation and Generalisation

The subclasses in which membership can be determined by a condition on the value of some attribute are called *predicate-* or *condition-defined subclasses*. The condition that decides the subclass is called the *defining predicate*.

If all subclasses in a specialisation are predicate-defined on the same attribute, the specialisation itself is called *attribute-defined*. The attribute is called the *defining attribute* or the *discriminator*.

Other subclasses are *user-defined*, and specified individually for each entity by the user.

Other constraints that may apply to a specialisation are:

- *disjointness*: the subclasses must be disjoint sets. All attribute-defined specialisation are disjoint. Other specialisations are *overlapping*.
- *completeness/totalness*: if the specialisation is *total*, then every entity must be a member of some subclass. Other specialisations are *partial*.

A superclass identified by generalisation is usually total.

4.3.2 Specialisation and Generalisation Hierarchies and Lattices

Nested subclasses form either specialisation hierarchies or lattices.
In a *specialisation hierarchy*, every subclass has only one parent, *i.e.*, participates as a subclass in only one class/subclass relationship.
In a *specialisation lattice*, a subclass can be a subclass in more than one such relationship.

A *leaf node* in a hierarchy is a class with no subclasses of its own.

A subclass with more than one superclass is called a *shared subclass*. Such a subclass has the property of *multiple inheritance*.

Analogously, we have *generalisation hierarchies* and *generalisation lattices*.

4.3.3 Utilising Specialisation and Generalisation in Refining Conceptual Schemas

Starting with a superclass and repeatedly specialising it is called *top-down conceptual refinement*. The opposite process is called *bottom-up conceptual synthesis*.

5.1 Relational Model Concepts

The relational model represents the DB as a collection of relations, each of which resembles a *flat file* of records (where each record has a simple linear or flat structure).

A relation can be thought of as a *table* of values, each row of which represents a collection of related values.

5.1.1 Domains, Attributes, Tuples, and Relations

A *domain* is a set of atomic values. *Atomic* values are those values which are indivisible w.r.t the relational model.

The logical definitions of domains describe their contents informally (like `Indian_phone_numbers`). Each domain also has a *datatype* or *format*, (like `ddddddddd`).

A *relational schema* or *scheme* $R(A_1, \ldots, A_n)$ is made up of a relation name R and a list of attributes A_i. Each *attribute* A_i is a role played by some domain D_i in the schema R. D_i is called the *domain* of A_i, denoted by $\mathrm{dom}(A_i)$. R is the *name* of the relation described by the schema. n is called the *degree* or *arity* of the schema.

A *relation* or a *relation state* of the relational schema $R(A_1, \ldots, A_n)$ is a set of n-tuples $r = \{t_1, \ldots, t_m\}, t_i = \langle v_1, \ldots, v_n \rangle$. Each value v_i belongs to $\mathrm{dom}(A_i)$.

It is common to call the schema R the *relation intension* and its state $r(R)$ the *relation extension*.

Mathematically, $r(R) \subseteq \mathrm{dom}(A_1) \times \cdot \times \mathrm{dom}(A_n)$.

5.1.2 Characteristics of Relations

- Ordering of tuples in a relation: In a file, the records have some order; however, in the relation it represents, there is no notion of ordering of tuples.
- Ordering of values within a tuple and an alternative definition of a relation: the ordering of values is not important as long as the correspondence between attributes and values is maintained. An alternative definition of a relation

characterises tuples as mappings $t_i : R \rightarrow \bigcup_{1 \leq j \leq n} \text{dom}(A_j)$ such that $t_(A_j) \in \text{dom}(A_j)$. This is equivalent to defining t_i as a set of pairs $\langle A_j, v_j \rangle$, where $v_j \in \text{dom}(A_j)$. This kind of data is called *self-describing data*.

- Values and NULLs in the tuples: Each value in a tuple is atomic (this is called the *flat relational model*). This assumption is called the *first normal form* assumption. The NULL value is used to represent the values of attributes that are unknown for or inapplicable to a tuple.
- Interpretation (meaning) of a relation: The schema can be interpreted as a declaration or a tuple of assertion. Each tuple is a fact or an instance of the assertion. Alternatively, a relation schema can be a predicate, and the tuples the values that satisfy the predicate. By the *closed world assumption*, only true facts are present in the extension of the relation.

5.1.3 Relational Model Notation

5.2 Relational Model Constraints and Relational Database Schemas

Constraints on DBs can be *inherent model-based constraints* (or *implicit constraints*), *schema-based constraints* (or *explicit constraints*), or *application-based constraints* (or *semantic constraints* or *business rules*).

Data dependencies (which include *functional dependencies* and *multivalued dependencies*) form another category of constraints.

5.2.1 Domain Constraints

Domain constraints specify that $t(A_i) \in \text{dom}(A_i)$ for all tuples t.

5.2.2 Key Constraints and Constraints on NULL Values

Usually, there are subsets of attribues of a relation schema R which have different combinations of values for all tuples. Such a set is called a *superkey* of the schema, which specifies a uniqueness constraint on the relation.

A *key* k of R is a minimal superkey. The set of attributes constituting a key is a time-invariant property of a relation.

All the keys of a given a relation schema are called its *candidate keys*, and one of them is designated as a *primary key* (making the rest *unique keys*).

5.2.3 Relational Databases and Relational Database Schemas

A relational database schema S is a set of relation schemas $\{R_1, \ldots, R_m\}$ and a set of integrity constraints IC. A *relational database state* (or *snapshot*) DB is a set of relation states $\{r_1, \ldots, r_m\}$ for each relation schema $R_i \in S$. A state may be valid or invalid depending on whether or not it satisfies all constraints in IC.

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

The *entity integrity constraint* states that no primary key value can be NULL.

The *referential integrity constraint* is specified between two relations. It states that a tuple in one relation that refers to one in another relation must refer to an existing tuple in that relation. More formally, let us define a *foreign key* FK of a relation R_1 that references relation R_2 if $\mathrm{dom}(\mathrm{FK}) = \mathrm{dom}(\mathrm{PK})$ [where PK is the primary key of R_2] and all values of FK either occur in R_2 or are null.
FK is said to *reference* R_2, and if $t_1 \in R_1$'s FK value is the same as $t_2 \in R_2$'s PK value, then t_1 is said to *reference* t_2.

5.2.5 Other Types of Constraints

Constraints that are not part of the DDL have to be specified and enforced in a different way (within the application programs or using a *constraint specification language*). Mechanisms called *triggers* and *assertions* can be used in SQL.

Such constraints can be *state constraints* or *transition constraints*.

5.3 Update Operations, Transactions, and Dealing with Constraint Violations

The relational model's operations can be *retrievals* (specified by the operations of relational algebra) or *updates*.

5.3.1 The Insert Operation

This operation provides a list of attribute values for a new tuple that is to be inserted into a relation.

Key constraints (if the key attribute's value is already taken by some tuple), entity integrity (if any part of the PK is NULL) or referential integrity (if the value referenced by the FK does not exist) can be violated.

5.3.2 The Delete Operation

This operation removes a tuple from a relation.

It can violate only referential integrity (if the key being deleted is referenced by FKs in other tuples). In this case it can be *rejected*, *cascaded*, or *set NULL* (or *set default*).

5.3.3 The Update Operation

This operation is used to change one or more attributes' values in one or more tuples in a

relation. A condition to select the tuple(s) must be specified.

Updating an attribute which is in neither a PK nor an FK causes no problems. In other cases, the issues of both insertion and deletion are relevant.

5.3.4 The Transaction Concept

A *transaction* is an executing program that includes some DB operations. They are executed by DB application programs accessing a relational DB.

Transactions must leave the DB in a valid state after executing. They form an atomic unit of work on the DB.

5.4 Summary

6.1 SQL Data Definition and Data Types

SQL calls relations *tables*, tuples *rows* and attributes *columns*. The `CREATE` statement can be used for schemas, tables, types and domains, as well as views, assertions and triggers.

6.1.1 Schema and Catalogue Concepts in SQL

SQL schemata allow us to group together tables that belong to the same database application. An SQL schema is identified by a schema name and includes an authorisation identifier, as well as descriptors for each element (tables, types, constraints, views, domains, etc.).

A schema is created by the `CREATE SCHEMA` statement.

The privilege to create schemas and other constructs must be explicitly granted to the relevant user accounts by the system administrator.

A *catalogue* is a named collection of schemas, one of which is a special one named INFORMATION_SCHEMA. Integrity constraints can only be declared within catalogues.

6.1.2 The CREATE TABLE Command in SQL

This command specifies a new relation with its attributes and initial constraints. Each attribute has a name, a datatype, and possibly constraints. Integrity constraints can be part of the declaration or added later.

The schema in which it is created is either implicitly specified in the environment or explicitly attached to the table name.

Tables created by this statement are called *base tables*, as opposed to virtual relations, which are made by the CREATE VIEW statement. The latter may or may not correspond to an actual file.

Note that the attributes of a table are ordered, but its rows may not be.

6.1.3 Attribute Data Types and Domains in SQL

The basic datatypes available include numeric, character string, bitstring, Boolean, date and time.

A domain can be declared using the `CREATE DOMAIN` statement.

6.2 Specifying Constraints in SQL

6.2.1 Specifying Attribute Constraints and Attribute Defaults

Constraints can be specified on attributes using keywords like `NOT NULL`. A `DEFAULT` clause can be added to an attribute to specify a default value.

6.2.2 Specifying Key and Referential Integrity Constraints

These can be given in the `CREATE TABLE` statement. Examples include `PRIMARY KEY` (one or more attributes), `UNIQUE`, `FOREIGN KEY`, etc.

An integrity violation is handled by rejecting the operation, but a referential triggered action can also be attached to the constraint, like `SET NULL`, `CASCADE` and `SET DEFAULT`. This must be qualified by either `ON DELETE` or `ON UPDATE`.

6.2.3 Giving Names to Constraints

6.2.4 Specifying Constraints on Tuples Using `CHECK`

`CHECK` is an example of a row-based constraint as it applies to all the tuples.

6.3 Basic Retrieval Queries in SQL

The basic statement for retrieving information in SQL is the `SELECT` statement. Note that SQL allows tables to have two identical tuples by default.

6.3.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries

The simplest form of the `SELECT` command is

```
SELECT <attrs>
FROM <tables>
WHERE <cond>;
```

The basic logical comparison operators can be used in the condition.

The attributes specified are called the *projection attributes*, and the `WHERE` condition is called the *selection condition*. If the condition refers to tuples from two tables, it is called a *join condition*.

6.3.2 Ambiguous Attributes Names, Aliasing, Renaming and Tuple Variables

Aliases are given by the `AS` keyword. If two tables have attributes with the same name, they must be qualified by the table name to avoid ambiguity.

6.3.3 Unspecified `WHERE` Clause and Use of the Asterisk

When there is no `WHERE` clause and multiple tables, the cross product of the tables is selected. In case we want all attributes, we use the `*` instead of listing all of them out.

6.3.4 Tables as Sets in SQL

The keyword `DISTINCT` can eliminate duplicate tuples in the result of `SELECT` command.

The `UNION`, `EXCEPT` and `INTERSECT` operations can be applied to two type-compatible relations. Adding the keyword `ALL` allows us to use these operations on multisets.

6.3.5 Substring Pattern Matching and Arithmetic Operators

The `%` symbol is like `*` in regexes, and `_` is like `.` in regexes. Strings using these characters can be matched against normal strings using `LIKE`.

Arithmetic operators can also be used in the attribute list.

6.3.6 Ordering of Query Results

Tuples can be ordered according to certain attributes using the `ORDER BY` clause.

6.3.7 Discussion and Summary of Basic SQL Retrieval Queries

6.4 `INSERT`, `DELETE` and `UPDATE` Statements in SQL

6.4.1 The `INSERT` Command

It is used to add a single row to a relation. The values should be listed in the same order as the corresponding attributes in the `CREATE TABLE` command, unless a new ordering is specified.

Multiple tuples can be inserted, enclosed in brackets and separated by commas.

The output of a `SELECT` statement can also be used in an `INSERT` command.

6.4.2 The `DELETE` Command

This removes tuples from a relation. It includes a `WHERE` clause.

It may lead to referential triggered actions, but otherwise acts on only one table at a time.

6.4.3 The `UPDATE` Command

This command, too, may lead to referential triggered actions. It also has a `WHERE` clause, and also a `SET` clause for the new values of the tuple.

Additional Features of SQL

Summary

7.1 More Complex SQL Retrieval Queries

7.1.1 Comparisons Involving `NULL` and Three-Valued Logic

A `NULL` has one of three interpretations – value unknown, unavailable or not applicable. SQL does not distinguish among these.

Every `NULL` is considered different from every other. The result of a comparison involving a `NULL` is `UNKNOWN`, which is distinct from `TRUE` and `FALSE`.
Logical operations are extended to deal with `UNKNOWN`.

Comparisons with `NULL` are done using `IS` and `IS NOT` instead of `=` or `<>`.

7.1.2 Nested Queries, Tuples and Set/Multiset Comparisons

Nested queries allow us to use the resluts of a certain query as part of the comparison in the `WHERE` conditino of another.

We can use the keywords `IN`, `ANY` (or `SOME`) and `ALL` as part of such conditions.

As a rule, in case of ambiguities, a reference to an unqualified attribute refers to the relation declared in the innermost nested query.

7.1.3 Correlated Nested Queries

If a condition in a nested query references some attribute of a relation declared in the outer query, the two queries are *correlated*.

7.1.4 The `EXISTS` and `UNIQUE` Functions in SQL

These keywords are Boolean functions. They are applied to relations.

`EXISTS` checks if a relation is empty or not; `UNIQUE` checks for duplicate tuples in a relation.

7.1.5 Explicit Sets and Renaming in SQL

An explicit set can be used with `IN`; it must be enclosed in parantheses.

It is also possible to rename attributes that appear in the *result* of a `SELECT` query.

7.1.6 Joined Tables in SQL and Outer Joins

A joined table is easier to understand than an entire `SELECT` statement. The condition on which the join takes place is specified using the `ON` keyword.
Different types of join, like NATURAL JOIN, OUTER JOIN and EQUIJOIN can be specified.

7.1.7 Aggregate Functions in SQL

These summarise information from multiple tuples in a single tuple. Grouping creates subgroups of tuples prior to this. Built-in aggregate functions include `COUNT`, `SUM`, `MAX`, `MIN` and `AVG`.

Aggregate functions (except `COUNT`) usually discount `NULL` values.

The functions `SOME` and `ALL` can be applied to collections of Boolean values.

7.1.8 Grouping: The `GROUP BY` and `HAVING` Clauses

`GROUP BY` partitions the table into nonoverlapping subsets or groups, on each of which the aggregate function can then be applied. A separate group is created for all tuples with a `NULL` value.

The `HAVING` clause filters out groups with a condition. Note that `WHERE` is applied before

`HAVING`.

8.1 Unary Relational Operations: SELECT and PROJECT

8.1.1 The SELECT Operation

This operation chooses a subset of tuples from a relation that satisfies a certain condition, like a filter. For example, $\sigma_{\mathrm{Dno}=4}(\mathrm{EMPLOYEE})$, or in general, $\sigma_{\mathrm{condition}}(R)$.
The attributes of the resulting relation are the same as those of the input.

The condition is a Boolean expression on the attributes.
The fraction of tuples selected by a condition is called its *selectivity*.

The SELECT operation is commutative and can be cascaded.

$\sigma_c(R)$ is equivalent to the SQL statement

```
SELECT *
FROM R
WHERE c
```

8.1.2 The PROJECT Operation

This operation selects certain columns from a relation and discards the rest. It is denoted $\pi_{\text{attribute list}}(R)$. The degree of the result is equal to the cardinality of the attribute list.

Duplicate tuples are removed, in the case where the attribute list includes only non-key attributes of R.

$\pi_l(R)$ is equivalent to the SQL statement

```
SELECT DISTINCT l
FROM R
```

8.1.3 Sequences of Operations and the RENAME

Operation

This operation allows us to name intermediate relations so as to avoid long sequences of operations. For example,
ParseError: KaTeX parse error: Expected 'EOF', got '_' at position 11: \text{DEP5_EMPS} \gets \si…
ParseError: KaTeX parse error: Expected 'EOF', got '_' at position 63: …ary}(\text{DEP5_EMPS})

We can also rename the attributes of the relations using this:
$\text{TEMP} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$
ParseError: KaTeX parse error: Expected 'EOF', got '_' at position 14: R(\text{First_name, Last_name…

Formally, it is denoted $\rho_{S(B_1,\ldots,B_n)}(R)$, or $\rho_S(R)$, or $\rho_{(B_1,\ldots,B_n)}(R)$. The most general way to write this in SQL is

```
SELECT A1 AS B1, ..., An AS Bn
FROM R as S
```

8.2 Relational Algebra Operations from Set Theory

8.2.1 The UNION, INTERSECTION, and MINUS Operations

When these operations are applied to pairs of relations, the tuples in the two relations must be of the same type; this is called *union* or *type* compatibility (if they have the same degree and the same attribute domains).

Conventially, the result has the same attribute names as the first relation.

UNION and INTERSECTION are both commutative and associative, and can therefore be generalised to act on any number of relations.

8.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

This operation is denoted by $\times$. There is no condition of compatibility; its operation is of the form $R(A_1, \dots, A_n) \times S(B_1, \dots, B_m) \to Q(A_1, \dots, A_n, B_1, \dots, B_m)$. The cardinality of the result is the product of the cardinalities of the operands, and its degree is the sum of degrees of the operands.

The n-ary CARTESIAN PRODUCT is a natural generalisation of this.

In SQL, this operation is equivalent to the `CROSS JOIN`{.sql} statement.

8.3 Binary Relational Operations: JOIN and DIVISION

8.3.1 The JOIN Operation

This operation is denoted by $\bowtie$ and is used to combine related tuples from two relations. The JOIN operation can be expressed in terms of other operations as $R \bowtie_c S = \sigma_c(R \times S)$. The condition c involves attributes of both R and S.

8.3.2 Variations of JOIN: The EQUIJOIN and NATURAL JOIN

A JOIN operation where the only comparison operator is $=$ is called an EQUIJOIN.

In the case of EQUIJOINs, two attributes in the result always have the same values. NATURAL JOIN removes the second of these columns, if they have the same name in both relations. It is denoted by $*$.A

The size of the JOIN result $R \bowtie_c S$ divided by the size of $R \times S$ is called the *join selectivity*.

JOIN operations are *inner joins*; they produce one table from the information contained in 2

tables. These are defined as a combination of CARTESIAN PRODUCT and SELECTION.

8.3.3 A Complete Set of Relational Algebra Operations

The set ${\sigma, \pi, \cup, \rho, -, \times} is a *complete* set. Other operations are included for convenience.

8.3.4 The DIVISION Operation

This operation is denoted by $\div$. Formally, $R(Z) \div S(X)$, where $X \subseteq Z$, is the set $Q(Z - X)$ of all tuples t such that there are t_R in R such that $t_R[Z - X] = t$ and $t_R[X] = t_S$ for all $t_S \in S$.

It can be described using the basic operations as
$Q_1 \leftarrow \pi_Y(R)$
$Q_2 \leftarrow \pi_Y((S \times T_1) - R)$
$Q \leftarrow T_1 - T_2$

8.3.5 Notation for Query Trees

A query tree is a data structure that corresponds to a relational algebra expression. The inputs are leaf nodes and the relational algebra operations are internal nodes.

8.4 Additional Relational Operations

8.4.1 Generalised Projection

This allows functions of attribute values to be included in the projection list, *i.e.*, $\pi_{F_1,\ldots,F_n}(R)$, where F_i are functions over the attributes of R.

8.4.2 Aggregate Functions and Grouping

We can also specify aggregate functions on collections of values from the database, commonly SUM, AVERAGE, MAXIMUM, MINIMUM, and COUNT.

We might also want to apply the aggregate function to groups of tuples rather than the whole relation. This is denoted as $_{\text{grouping attributes}}\Im_{\text{function list}}(R)$. The result has the grouping attributes plus one attribute for each function.

8.4.3 Recursive Closure Operations

Recursive closures apply to relationships between tuples of the same type. For example, we might want to find all supervisees of an employee e, along with all those supervised by one of e's supervisees, and so on.

8.4.4 OUTER JOIN Operations

An OUTER JOIN operation does not result in any loss of information. It behaves like an ordinary EQUIJOIN, but tuples which do not satisfy the condition are retained and a NULL value is inserted for them.

In a LEFT OUTER JOIN, all tuples from the first operand are retained; in a RIGHT OUTER JOIN, those from the second one are retained. We also have a FULL OUTER JOIN, which keeps all the tuples.

8.4.5 The OUTER UNION Operation

This operation carries out a kind of UNION between relations that are not type compatible. For example, it can be carried out between $R(X, Y)$ and $S(X, Z)$, resulting in $T(X, Y, Z)$. It is the same as performing a FULL OUTER JOIN on X.

8.5 Examples of Queries in Relational

Algebra

8.6 The Tuple Relational Calculus

We write a declarative expression to specify a rerieval; the relational calculus is thus a non-procedural language. The expressive power, however, is identical to that of the relational algebra.

8.6.1 Tuple Variables and Range Relations

The calculus is based on specifying a number of tuple variables. Each tuple ranges over a particular relation, e.g. `{t | COND(t)}`.
All tuples satisfying `COND` are then included. A condition can specify the range relation of a variable, as in `{t | EMPLOYEE(t) AND t.Salary > 50000}`.

8.6.2 Expressions and Formulas in Tuple Relational Calculus

A general expression has the form `{t1.Aj, t2.Ak, ..., tn.Am | COND(t1,t2,...,tn,...,tp)}`.

`COND` is either a condition specifying the range, or a copmarison-based condition.

8.6.3 The Existential and Universal Quantifiers

These quantifiers can be used in conditional formulae.

8.6.4 Sample Queries in Tuple Relational Calculus

8.6.5 Notation for Query Graphs

8.6.6 Transforming the Universal and Existential Quantifiers

8.6.7 Using the Universal Quantifier in Queries

8.6.8 Safe Expressions

9.1 Relational Database Design Using ER-to-Relational Mapping

9.1.1 ER-to-Relational Mapping Algorithm

Step 1: Mapping of Regular Entity Types

For each regular entity type E, create a relation R that includes all the simple attributes of E. For a composite attribute, include only the simple component attributes.

Choose one of E's key attributes as R's primary key. If E's key is composite, then all the simple attributes in it form the key.
The knowledge of E's other keys is kept for indexing purposes.

Such relations are called *entity relations*.

Step 2: Mapping of Weak Entity Types

For each weak entity type W, create a relation R and include all simple attributes and simple components of composite attributes. In addition, include as foreign key attributes of R the primary key attributes of the owner type relation.

The primary key of R is the combination of the primary key of the owner type and the partial key of W.

Step 3: Mapping of Binary 1:1 Relationship Types

For each such relationship type R, identify the relations S and T corresponding to the participating entity types. There are three approaches.

Foreign Key Approach

Include the primary key of T as a foreign key in S. The role of S should be an entity with total participation in R.
Include all the simple attributes and simple components of composite attributes as attributes S.

Merged Relation Approach

Both the entity types can be merged into a single relation (this is possible when both the participations are total).

Cross-Reference or Relationship Relation Approach

A third relation R is set up to cross-reference the primary keys of the two relations S and T of the participating entity types. This approach is necessary for binary M:N relations.
R is called a *relationship relation* or *lookup table*.

Step 4: Mapping of Binary 1:N Relationship Types

There are two approaches.

Foreign Key Approach

Let S be the relation for the participating entity type at the N-side. Include as a foreign key in S the primary key of the relation T.

Relationship Relation Approach

As for 1:1 relationships, create a new relation that has as its attributes the primary keys of S and T. R's primary key is the same as S's.

Step 5: Mapping of Binary M:N Relationship Types

When there are no multivalued attributes, the only option is the relationship relation approach. As before, a new relation with the primary keys of the participating types as attributes must be created, but the primary key is now the combination of both these.

Step 6: Mapping of Multivalued Attributes

For each multivalued attribute A, create a new relation R, with an attribute corresponding to A plus the primary key of the entity whose attribute A is.

Step 7: Mapping of N-ary Relationship Types

For this too, we use the relationship relation option, including a new relationship relation for each n-ary attribute ($n > 2$). Include as foriegn key attributes the primary keys of the relations that represent the participating entity types.
The primary key of S should include all the foreign keys except those of entity types with cardinality constraint 1.

9.1.2 Discussion and Summary of Mapping for ER Model Constructs

9.2 Mapping EER Model Constructs to Relations

9.2.1 Mapping of Specialisation or Generalisation

The two main options are to map the whole specialisation into a single table, or to map it into multiple tables.

Step 8: Options for Mapping Specialisation or Generalisation

Convert each specialisation with m subclasses $\{S_1, \ldots, S_m\}$ and superclass C with attributes $\{k, a_1, \ldots, a_n\}$ into relation schemas with one of the following options.

Option 8A: Multiple Relations (Superclass and Subclasses)

Create a relation L for C with attributes $\{k, a_1, \ldots, a_n\}$ and primary key k, and a relation L_i for each S_i, with the attributes $\{k\} \cup \{\mathrm{Attrs}(S_i)\}$ and primary key k.
This works for any specialisation.

Option 8B: Multiple Relations (Subclass Relations Only)

Create a relation L_i for each S_i with the attributes $\{\text{Attrs}(S_i)\} \cup \{k, a_1, \ldots, a_n\}$ and primary key k.
This works for any specialisation with total subclasses, and is only recommended when the subclasses are disjoint.

Option 8C: Single Relation with One Type Attribute

Create a relation L with attributes $\{k, a_1, \ldots, a_n\} \cup \bigcup_{i=1}^{m} \text{Attrs}(S_i) \cup \{t\}$, and primary key k.
t is called a *type* or *discriminating* attribute which indicates the subclass to which a tuple belongs.
This only works for disjoint subclasses.

Option 8D: Single Relation with Multiple Type Attributes

Create a relation L with attributes $\{k, a_1, \ldots, a_n\} \cup \bigcup_{i=1}^{m} \text{Attrs}(S_i) \cup \{t_1, \ldots, t_m\}$, and primary key k.
Each t_i is a Boolean type attribute indicating whether or not a tuple belongs to S_i.
This works for overlapping subclasses as well.

9.2.2 Mapping of Shared Subclasses (Multiple Inheritance)

Any of the above options can be applied.

9.2.3 Mapping of Categories (Union Types)

•

9.3 Summary

14.1 Informal Design Guidelines for Relational Schemas

14.1.1 Imparting Clear Semantics to Attributes in Relations

The relational schema design should have a clear meaning and be easy to explain.

14.1.2 Redundant Information in Tuples and Update Anomalies

14.1.3 `NULL` Values in Tuples

14.1.4 Generation of Spurious Tuples

14.1.5 Summary and Discussion of Design Guidelines

14.2 Functional Dependencies

14.2.1 Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Let the whole database be represented as R = {A1, A2, ..., An}. A functional dependency is denoted X -> Y and means that t1[X] = t2[X] => t1[Y] = t2[Y] for all t1, t2 in all relation states r of R.

Note that if X is a candidate key of R, then X -> Y for all sets of attributes Y.

A FD is a property of the relation schema and *not* of its extension or state. It cannot therefore be derived from a state, but must be defined according to the semantics of R. F denotes the set of FDs specified on a relation schema R.

14.3 Normal Forms Based on Primary Keys

14.3.1 Normalisation of Relations

Normalisation can be considered a process of analysing relation schemata based on their FDs and primary keys to achieve the desirable properties of minimising redundancy and minimising anomalies.

The normal form of a relation refers to the highest NF condition that it meets. In isolation, normal forms do not guarantee good database design, but other properties must be checked for.

14.3.2 Practical Use of Normal Forms

Denormalisation is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

14.3.3 Definitions of Keys and Attributes Participating in Keys

A superkey of a relation schema R is a set of attributes whose combined value is unique for all tuples. A key is a minimal superkey.

If a relation schema has multiple keys, each is called a candidate key. One of them is called the primary key, and the rest are secondary.

An attribute of a relation schema is called a prime attribute if it is a member of some candidate key of R. Otherwise, it is called nonprime.

14.3.4 First Normal Form (1NF)

It disallows having a set, a tuple or a combination of values as an attribute value for a single tuple.

This can be handled by moving the multivalued attribute to a separate relation along with the PK of the old relation; or expand the key to have a separate tuple for each value; or (if the maximum number of values is known) add a number of columns for each value.

14.3.5 Second Normal Form (2NF)

A FD X -> Y is a full FD if the removal of any attribute A from X means that the dependency no longer holds.
R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.

A relation can be converted to 2NF by moving the subset of the PK on which some attribute is dependent to another table with that attribute.

14.3.6 Third Normal Form

A FD X -> Y is a transitive dependency if there is some set of attributes Z that is not a subset of any key such that X -> Z and Z -> Y hold.
R is in 3NF if it is in 2NF and no nonprime attribute of R is transitively dependent on the primary key.

14.4 General Definitions of Second and Third Normal Forms

14.4.1 General Definition of Second Normal Form

R is in 2NF if every nonprime attribute A in R is not partially dependent on any key of R (is fully

dependent on every key of R).

14.4.2 General Definition of Third Normal Form

R is in 3NF if, whenever a nontrivial FD X -> A holds in R, either X is a superkey of R or A is a prime attribute of R.

14.4.3 Interpreting the General Definition of Third Normal Form

The first condition catches cases where a nonprime attribute determines another (violating 3NF), and where a proper subset of a key determined a nonprime attribute (violating 2NF).

In other words, R is in 3NF if every nonprime attribute is fully functionally dependent on every key of R, and nontransitively dependent on every key of R.

14.5 Boyce-Codd Normal Form

It is stricter than 3NF. A relation is in BCNF if whenever a nontrivial FD X -> A holds, then X is a superkey.

14.5.1 Decomposition of Relations not in BCNF

We want decompositions to meet two properties – the nonadditive join property (certainly) and the functional dependency preservation property (preferably). The nonadditive join test for binary decompositions (NJB) allows us to check the first one.
A decomposition D = {R1,R2} of R satisfies the lossless join property w.r.t. a set of FDs F on R iff either

- ((R1 \ints R2) -> (R1 - R2)) is in F+, or
- ((R1 \ints R2) -> (R2 - R1)) is in F+

In general, if X -> A is the FD that causes a violation of BCNF in R. Then decompose R into R-A and XA, and repeat if either is not in BCNF.

15.1 Further Topics in Functional Dependencies: Inference Rules, Equivalence and Minimal Cover

15.1.2 Inference Rules for Functional Dependencies

A FD X -> Y is inferred from or implied by a set of dependencies F if X -> Y holds in every legal state of R.

The set of all dependencies that include F as well as all inferable dependencies is called the closure of F, denoted F+.
F |= X -> Y means that the dependency X -> Y is inferred from F.

The following rules of inference are called Armstrong's axioms:

- IR1: If Y \subseteq X, then X -> Y (reflexive rule)
- IR2: X -> Y |= XZ -> YZ (augmentation rule)
- IR3: X -> Y, Y -> Z |= X -> Z (transitive rule)

Three further rules follow from the above:

- IR4: X -> YZ |= X -> Y (decomposition/projective rule)
- IR5: X -> Y, X -> Z |= X -> YZ (union/additive rule)
- IR6: X -> Y, WY -> Z |= WX -> Z (pseudotransitive rule)

The closure of a set of attributes X under F is denoted X+ and includes all attributes that are functionall determined by X based on F.

15.1.2 Equivalence of Sets of Functional Dependencies

A set F of FDs is said to cover another set E if E \subseteq F+.
Two sets E and F of FDs are equivalent if E+ = F+.

15.1.3 Minimal Sets of Functional Dependencies

A minimal cover of a set E of FDs is a set F of FDs such that E covers F and it does *not* cover any proper subset of F.

An attribute in an FD is an extraneous attribute if we can remove it without changing the closure of the set. Formally, a set F of FDs is minimal if

- every FD has a single attribute on its RHS
- no attribute can be removed from the LHS of any FD
- no dependency can be removed from F

15.2 Properties of Relational Decompositions

15.2.1 Relational Decomposition and Insufficiency of Normal Forms

Let R = {A1, ..., An} be a universal relation schema and D = {R1, ..., Rm} a decomposition of it. The union of all Ri is R – this is the attribute preservation condition.

Further, each Ri should be in BCNF or 3NF. However, this is not enough.

15.2.2 Dependency Preservation Property of a Decomposition

If every dependency in F appeared in or could be derived from the dependencies of one of the Ri. This is called the dependency preservation condition.

The projection of a set F of FDs on Ri, denoted πRi(F), is the set of all X -> Y in F+ such that XY is in Ri.

D is dependency-preserving if the union of the projections of F on Ri is equivalent to F.

15.2.3 Nonadditive (Lossless) Join Property of a Decomposition

This property ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied.

D has the lossless join property w.r.t F if for every relation state that satisfies F, *(πR1(r), ..., πRm(r)) = r.

15.2.4 Testing Binary Decompositions for the Nonadditive Join Property

The NJB test allows us to test binary decompositions.

15.2.5 Successive Nonadditive Join Decompositions

If a decomposition D = {R1, ..., Rm} has the nonadditive join property w.r.t F on R, and if a decomposition Di = {Q1, ..., Qk} of Ri has it w.r.t the projection of F on Ri, then D2 = {R1, ..., R(i-1), Q1, ..., Qk, R(i+1), ..., Rm} also has it.

15.3 Algorithms for Relational Database Schema Design

15.3.1 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

15.3.2 Nonadditive Join Decomposition into BCNF Schemas