Database Management Systems Review Notes

Aditya Soni

Computer Science and Engineering, NIT Silchar

Reference: Database Management Systems (2nd ed.)
By Ramakrishnan R, Gehrke J

Contents

Chapter 1

Introduction

Database Collection of data, typically describing the activities of one or more related organisations.

Database management systems Software designed to assist in maintaining and utilizing large collections of data, and the need for such systems, as well as their use is growing rapidly.

1.1 Drawbacks of Storing Data as File System

(1) Have to bring relevant parts of data into memory as and when required.

(2) Data redundancy and inconsistency due to multiple file formats.

(3) Concurrent access by multiple users can cause inconsistencies.

(4) Security problems in encrypting and decrypting files.

(5) Ensuring data is restored in consistent state in case of system crash.

1.2 Advantages of DBMS

(1) Data independence: Application programs should be as independent as possible from details of data representation and storage. DBMS provides an abstract view of data.

(2) Efficient data access: DBMS utilizes variety of sophisticated techniques to store and retrive data efficiently.

(3) Data integrity and security: If data is always accessed through the DBMS, it can enforce integrity constraints on the data along with access control.

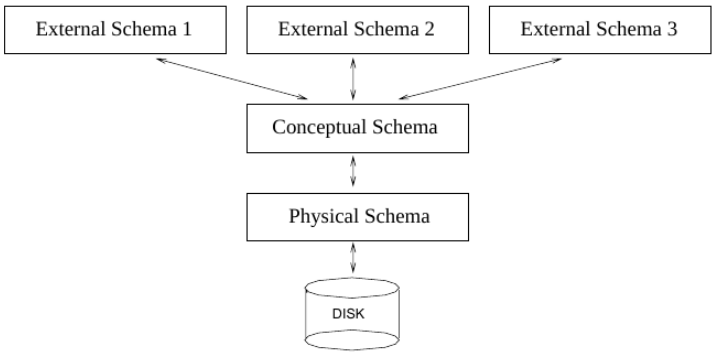
(4) Data administration: Centralizing the administration of data can offer significant improvement like minimizing redundancy and fine tuning for efficient retrieval.

(5) Concurrent access and crash recovery: DBMS schedules concurrent access to the data in a manner that users think only one user is accessing. It protects users for effects of system failures.

(6) Reduces application development time: DBMS supports many important function with high level interface to data.

Data Model Collection of high-level data description constructs that hide many low-level storage details. DBMS allows user to define the data to be stored in terms of data model.

1.3 Levels of Abstraction in DBMS



(1) Conceptual Schema (or Logical Schema): It describes the stored data in terms of data model of DBMS. In relational model, the conceptual schema describes all relations that are stored in database. The choice of fields and relations is always obvious, but the process of arriving at good conceptual schema is called conceptual database design.

(2) Physical Schema: It summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices. Decision about physical schema are based on understanding of how data is typically accessed, like creating auxiliary data structures called indexes to speed up retrievals. The process of arriving at good physical schema is called physical database design.

(3) External Schema: allows data to be customized and authorized at the level of individual users or group of users. This schema design is guided by end user requirements. Each external schema consists of a collection of one or more views and relations from the conceptual schema.

Data Independence Relations in the external schema (view relations) are in principle generated on demand from the relations corresponding to the conceptual schema. If the underlying data is reorganized, that is, the conceptual schema is changed, the definition of a view relation can be modified so that the same relation is computed as before. Thus, users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored. This property is called logical data independence. In turn, the conceptual schema insulates users from changes in the physical storage of the data. This property is referred to as physical data independence.

Chapter 2

Entity-Relationship Model

It allows users to describe the data involved in real world enterprise in terms of objects and their relationships and is used to develop initial database design.

2.0.1 Steps in Database Design

(1) Requirement Analysis

(2) Conceptual Database Design

(3) Logical Database Design

- Steps beyond the E-R model,

(4) Schema Refinement

(5) Physical Database Design

(6) Security Design

Entity A distinguishable object in real world. An instance of entity is a manifestation of the entity type and becomes rows of the database. Types are:

- Strong Entity: has a primary key (can uniquely identify all the entities).
- Weak Entity: cannot be uniquely identified by its own attributes and relies on the relationship with other entity.

Attributes Describes the characterstics of an entity. Types are:

- Key attribute: can uniquely identify entities from an entity set.
- Composite attribute: made of one or more simple attributes.
- Multivalued attribute: can hold more than one value.
- Derived attribute: value is dynamic and can be derived from other attribute(s).

Relationship An assosiation among two or more entities. It can also have descriptive attributes. The entity sets that participate in relationship set need not be distinct, however, they can play different roles.

2.1 Features of E-R Model

2.1.1 Key Constraints

Any rule that is applied to the data that is entered is a **constraint**. If entity set E has a key constraint in relationship set R, each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R. To indicate a key constraint on entity set E in relationship set R, an arrow is drawn from E to R.

2.1.2 Mapping Cardinality

Cardinality is the count of number of times one entity can (or must) be assosiated with each occurence of another entity via some relationship set. It can be One-to-one, One-to-many, Many-to-one or Many-to-many.

2.1.3 Participation Constraints

refers to whether an entity must participate in a relationship with another entity to exist. Participation can be:

- Total participation: every entity in entity set must participate in at least one relationship from relationship set. Example, an Employee must work in at least one department to be called 'employee'.
- Partial participation: an entity can exist without participating in a relationship with other entity. Example, entity Course can exist even if there are no currently enrolled Students in the course.

Weak entity set must have total participation in the identifying relationship set.

2.1.4 Class Hierarchies

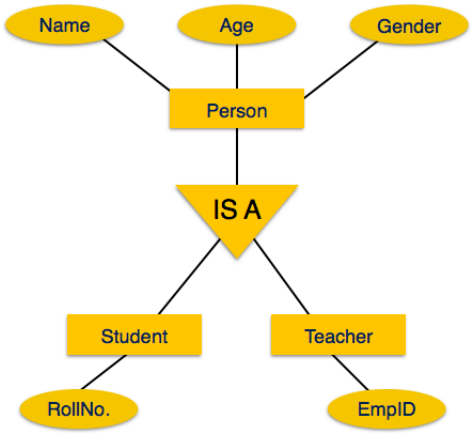
Structurally similar entities can be organized through inheritance into sub-classes and super-classes. B **ISA** A means B inherits all the attributes from A and add it's own. Superclass is specialized into subclasses, while subclasses are generalized by superclass. The need for identifying subclasses are: (i) wanting to add descriptive attributes that only makes sense for subclasses, and (ii) identifying set of entities that participate in some relationship that is restricted to entities in subclass.

Constraints defined with respect to ISA hierarchies:

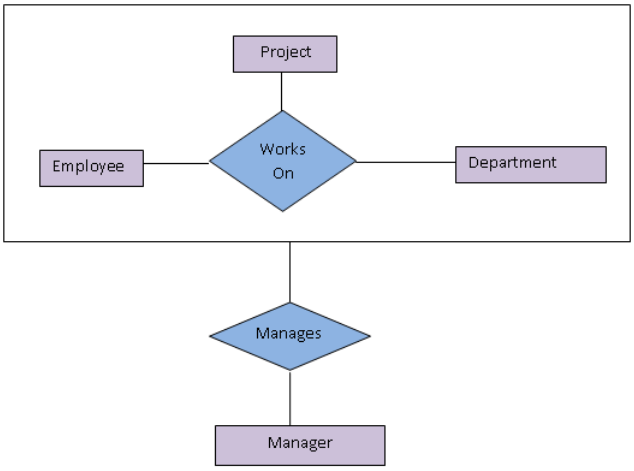
- Overlap Constraints: determines whether two subclasses are allowed to contain the same entity.
- Covering Constraints: determines whether the entities in the subclasses collectively include all entities in the superclass.

2.1.5 Aggregation

Aggregation conceptually transforms a relationship set into an entity set such that the resulting construct can be related to other entity sets.

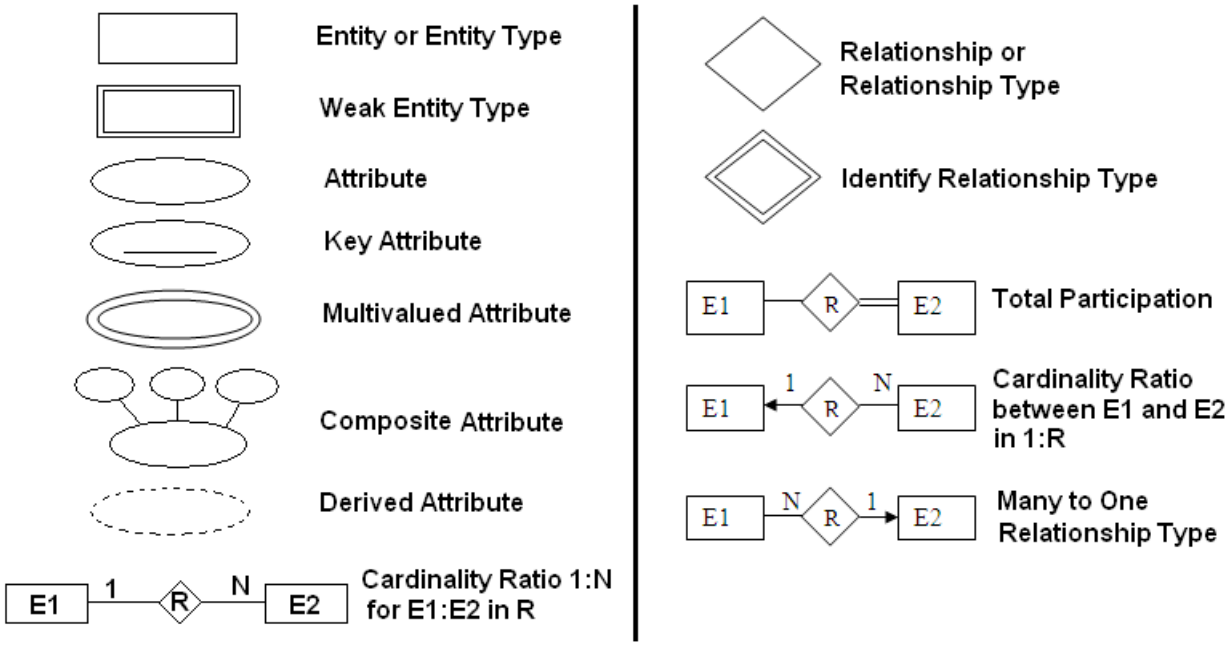


(a) ISA (inheritance)



(b) Aggregation

2.2 Symbols in ER diagrams



Chapter 3

Relational Model

Relation consists of a relation schema and a relation instance. An instance of relation is a set of tuples, also called records, in which each tuple has the same number of fields. The Degree (or arity) of relation is the number of fields, and the Cardinality of relation instance is the number of tuples in it.

Schema specifies the relation's name, name of each field (or column) and the domain of each field. The values appearing in the column must be drawn from the domain assosiated with that column.

Structured Query Language (SQL) is the most widely used language for creating, manipulating and querying relational DBMS.

- **Data Definition Language (DDL)** supports the creation, deletion and modification of schema objects. Integrity constraints can be defined on tables, and can specify access rights and privileges to tables and views. Commands in DDL are CREATE, DROP, ALTER, TRUNCATE, RENAME etc.
- **Data Manipulation Language (DML)** allows users to access and manipulate data in existing schema objects. Commands in DML are INSERT, UPDATE, DELETE, LOCK etc.
- **Data Query Language (DQL)** is used to fetch data from the database. Command in DQL is SELECT.
- **Data Control Language (DCL)** is used to give or modify rights and permissions to users. Commands in DCL are GRANT and REVOKE.
- **Transaction Control Language (TCL)** is used to manage transactions in the database. Commands in TCL that are commonly used include COMMIT, ROLLBACK, and SAVEPOINT.

3.1 Intergrity Constraints in Relational Model

A condition specified on the database schema, and restricts the data that can be stored in an instance of database. DBMS enforces integrity constraints and permits only legal instance to be stored in the database.

3.1.1 Domain Constraints

can be violated if an attribute value is not appearing in the corresponding domain or it's not of the appropriate data type.

3.1.2 Key Constraints

A certain minimal subset of the fields of a relation is a unique identifier for a tuple. The value in key attribute should be unique for each tuple.

- **Super Key** is set of one or more attributes which can uniquely identify a tuple in a table. Candidate keys are selected from the set of super keys.
- **Candidate Key** (or just key) for the relation is a set of fields that uniquely identifies a tuple according to a key constraint. This means, (i) two distinct tuples cannot have identical values in all fields of key, and (ii) no subset of set of fields in a key is a unique identifier for the tuple.
- **Primary Key** is a candidate key selected by database adminitrator to uniquely identify tuples in a table. PRIMARY KEY is used for it's declaration.

3.1.3 Referential Integrity Constraint

specifies that the **Foreign key** (key used to link two or more tables) in the referencing relation must match the primary key of the referenced relation. When a row in parent table is deleted, it can (must, in case of weak entity) be made sure that the referenced rows in the child table are deleted as well, using ON DELETE CASCADE line in the schema after defining foreign key.

3.2 Views

A view is a relation whose instance is not explicitly stored but is computed as needed. Logical data independence can be enabled by defining the external schema through views (i.e. it can be used to define relations in external schema that mask changes in the conceptual schema of the database). Also, views play an important role in restricting access to data for security reasons.

Basic Code Example: (MySQL)

```
CREATE DATABASE CollegeRecords;
USE CollegeRecords;
                                        |
CREATE TABLE Departments(               |   CREATE VIEW
    dept_id INT,                        |   [List of Departments] AS
    dept_name VARCHAR(30),              |   SELECT dept_name FROM
    PRIMARY KEY(dept_id)                |   Departments;
);                                      |
CREATE TABLE Faculties(
    faculty_id INT,
    name VARCHAR(30),
    dept_id INT,
    joined_on DATETIME,
    PRIMARY KEY(faculty_id),
    FOREIGN KEY(dept_id) REFERENCES Faculties(faculty_id)
);

DROP TABLE Faculties;
DROP DATABASE CollegeRecords;
```

Chapter 4

Relational Queries

4.1 Relational Algebra

A Relational Algebra query describes a procedure for computing the output relation from the input relations by applying *relational algebra operators*. Mainly useful in query optimization due to procedural nature. The operators in relational algebra are:

1. **Selection** (σ), to retain rows from a relation through a selection condition.
2. **Projection** (π), to project columns i.e. to extract specific columns from a relation. This operator eliminates duplicate rows in the result.

For example, a table Student exists as follows, and an instance is called S,

Sid	Name	Age	Grade
1	John	19	8
2	Maria	18	9
3	Dexter	20	7
4	Ferb	19	10

Selection and projection operators can be used as:

Sid	Name	Age	Grade
2	Maria	18	9
4	Ferb	19	10

(a) $\sigma_{Grade>8}(S)$

Name	Age
John	19
Maria	18
Dexter	20
Ferb	19

(b) $\pi_{Name,Age}(S)$

Since the result of relational algebra expression is always a relation, an expression can be substituted where a relation is expected. For example,

Name	Grade
Maria	9
Ferb	10

$$\pi_{Name,Grade}(\sigma_{Grade>8}(S))$$

3. **Union** ($\cup$), $R \cup S$ returns relation instance containing all tuples that occur in either relation instance R or relation instance S (or both). R and S must be union-compatible i.e.

   - they have same number of fields
   - corresponding fields, taken in order from left to right, have same domains.

   and the schema of the result is defined identical to the schema of R (Field names are not used in defining union compatibility).

4. **Set-difference** ($-$), $R - S$ returns a relation instance containing all tuples that occur in R, but not in S. Again, R and S must be union compatible and schema of the result is identical to the schema of R.

5. **Cross-product** (also called Cartesian-product) ($\times$), $R \times S$ returns a relation instance whose schema is concatenation of all the fields of R (in order) followed by all the fields in S (in order). The result of $R \times S$ contains one tuple (r, s) for each pair of tuples $r \in R, s \in S$. There could be a naming conflict if R and S contains one or more fields with the same name. This could be resolved using the renaming operator.

6. **Rename** (ρ), The expression $\rho(R(\overline{F}), E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. The field names in relation R are the same as in E, except for fields renamed in the renaming list F, which is a list of terms having the form oldname $\rightarrow$ newname or position $\rightarrow$ newname.

7. **Intersection** ($\cap$), $R \cap S$ returns a relation instance containing all tuples that occur in both R and S. Again, R and S must be union compatible and schema of the result is identical to the schema of R. $\cap$ operator is redundant as $R \cap S$ can be defined as $R - (R - S)$.

For Example, Let $S1$ and $S2$ be instances of relation Students. Then above operations will result in the following:

S_id	Name	Grade
1	Ferb	A+
2	Maria	B-
3	Smith	B+
4	Lisa	C+

(a) Students $S1$

S_id	Name	Grade
3	Smith	B+
6	Henry	F
7	Peter	D-

(b) Students $S2$

S_id	Name	Grade	S_id	Name	Grade
1	Ferb	A+	3	Smith	B+
1	Ferb	A+	6	Henry	F
1	Ferb	A+	7	Peter	D-
2	Maria	B-	3	Smith	B+
2	Maria	B-	6	Henry	F
2	Maria	B-	7	Peter	D-
3	Smith	B+	3	Smith	B+
3	Smith	B+	6	Henry	F
3	Smith	B+	7	Peter	D-
4	Lisa	C+	3	Smith	B+
4	Lisa	C+	6	Henry	F
4	Lisa	C+	7	Peter	D-

(a) Cross Product $S1 \times S2$

S_id	Name	Grade
1	Ferb	A+
2	Maria	B-
3	Smith	B+
4	Lisa	C+
6	Henry	F
7	Peter	D-

(b) Union $S1 \cup S2$

S_id	Name	Grade
3	Smith	B+

(c) Intersection $S1 \cap S2$

S_id	Name	Grade
1	Ferb	A+
2	Maria	B-
4	Lisa	C+

(d) Set-difference $S1 - S2$

8. **Join** ($\bowtie$), can be defined as cross product followed by selections and projections. The variants of join operation are as follows:

   - Condition joins (or theta join), i.e. cross product followed by selection based on some condition.

   $$R \bowtie S = \sigma_c(R \times S)$$

   - Equijoin, a special case of theta join where join condition only consists of equalities i.e. of the form $R.field1 = S.field2$.
   - Natural Join, a special case of equijoin in which equalities are specified on all fields having same names in R and S. The condition can be omitted and simply denoted by $R \bowtie S$. To prevent redundancy, S.field2 is dropped from the resulting relation. If no fields are common in R and S, then natural join $R \bowtie S$ is simply the cross product.
   - Left Outer Join, returns all rows from the left table and matching rows from the right table. Rows from left table without a match is returned with $NULL$ values in fields from right table.
   - Right Outer Join, returns all rows from the right table and matching rows from the left table. Rows from right table without a match is returned with $NULL$ values in fields from left table.
   - Full Outer Join, returns all rows from the left and right table. For rows with no matching, $NULL$ is returned in the fields.

9. **Division**: Considering two relation instances A and B, where attributes of B is subset of attributes of A, the division operation will return all attributes of A except attributes that are of B, with tuples of A that are assosiated with all tuples of B. Formally, A/B is defined as

$$A/B = \pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$$

Note: Intersection, Join and Division are *extended* operators in relational algebra.

For Example, Let Students and Courses be relation instances. Then join operation results will be as follows:

S_id	Name	Grade	Course_id
1	Ferb	A+	204
2	Maria	B-	101
3	Smith	B+	305
4	Lisa	C+	102
5	Dexter	A-	101
6	Henry	F	104
7	Peter	D-	201

(a) Students

Course_id	C_name
101	Physics
102	Chemistry
103	Biology
202	Blockchain
204	Calculus
305	English

(b) Courses

S_id	Name	Grade	Course_id	Course_id	C_name
1	Ferb	A+	204	101	Physics
2	Maria	B-	101	101	Physics
4	Lisa	C+	102	101	Physics
5	Dexter	A-	101	101	Physics
6	Henry	F	104	101	Physics
7	Peter	D-	201	101	Physics
5	Dexter	A-	101	305	English

(a) Theta join
($Students \bowtie_{Students.Name<Courses.C\_name} Courses$)

S_id	Name	Grade	Course_id	Course_id	C_name
2	Maria	B-	101	101	Physics
5	Dexter	A-	101	101	Physics
4	Lisa	C+	102	102	Chemistry
1	Ferb	A+	204	204	Calculus
3	Smith	B+	305	305	English

(b) Equijoin
($Students \bowtie_{Students.Course\_id=Courses.Course\_id} Courses$)

Course_id	S_id	Name	Grade	C_name
101	2	Maria	B-	Physics
101	5	Dexter	A-	Physics
102	4	Lisa	C+	Chemistry
204	1	Ferb	A+	Calculus
305	3	Smith	B+	English

(c) Natural join ($Students \bowtie Courses$)

S_id	Name	Grade	Course_id	Course_id	C_name
1	Ferb	A+	204	204	Calculus
2	Maria	B-	101	101	Physics
3	Smith	B+	305	305	English
4	Lisa	C+	102	102	Chemistry
5	Dexter	A-	101	101	Physics
6	Henry	F	104	(null)	(null)
7	Peter	D-	201	(null)	(null)

(d) Left-Outer join

S_id	Name	Grade	Course_id	Course_id	C_name
2	Maria	B-	101	101	Physics
5	Dexter	A-	101	101	Physics
4	Lisa	C+	102	102	Chemistry
(null)	(null)	(null)	(null)	103	Biology
(null)	(null)	(null)	(null)	202	Blockchain
1	Ferb	A+	204	204	Calculus
3	Smith	B+	305	305	English

(e) Right-Outer join

S_id	Name	Grade	Course_id	Course_id	C_name
1	Ferb	A+	204	204	Calculus
2	Maria	B-	101	101	Physics
3	Smith	B+	305	305	English
4	Lisa	C+	102	102	Chemistry
5	Dexter	A-	101	101	Physics
6	Henry	F	104	(null)	(null)
7	Peter	D-	201	(null)	(null)
(null)	(null)	(null)	(null)	103	Biology
(null)	(null)	(null)	(null)	202	Blockchain

(f) Full-Outer join

4.2 Relational Calculus

Relational Calculus is non-procedural, or declarative, which allows describing set of answers without being explicit about how they should be computed. It has two variants:

- Tuple Relational Calculus (TRC), where variables take on tuples as values. It had great influence on development of SQL
- Domain Relational Calculus (DRC), where variables range over field values. It had great influence on development of Query-By-Example (QBE).

4.2.1 Tuple Relational Calculus

A tuple relational calculus query has the form $T \mid p(T)$, where T is a tuple variable (only free variable in the formula) and $p(T)$ denotes a formula that describes T. The result of this query is the set of all tuples t for which the formula p(T) evaluates to true with $T = t$.
Formally, let *Rel* be a relation name, R and S be tuple variables, a be an attribute of R, and b be an attribute of S. Let op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$. Now, an Atomic formula is one of the following:

- $R \in Rel$
- $R.a\ op\ S.b$
- $R.a\ op\ constant$, *or* $constant\ op\ R.a$

A formula is recursively defined to be one of the following, where p and q are themselves formulas, and p(R) denotes a formula in which the tuple variable R appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$
- $\forall R(p(R))$

Quantifiers ($\exists$ and $\forall$) *binds* the variable. Otherwise, if quantifiers are not present in the formula, then the variable is said to be *free*.

All relational algebra queries can be expressed in relational calculus. If a query language can express all the queries that can be expressed in relational algebra, it is said to be **relationally complete**.

For example, for a relation Students with attributes $\{s\_id, name, grade\}$, and relation Courses with attributes $\{s\_id, c\_id\}$,

(a) find the student details for students whose grade is greater than 8: $\{t \mid t \in Students \wedge t.grade > 8\}$.

(b) find the name of student with $s\_id = 123$: $\{t \mid \exists s \in Students(s.s\_id = 123 \wedge t.name = s.name)\}$.

(c) find the names of students taking course with $c\_id = 1024$:
$\{t \mid \exists s \in Students \wedge \exists c \in Courses(s.s\_id = c.s\_id \wedge t.name = s.name \wedge c.c\_id = 1024)\}$.

Chapter 5

Schema Refinement

Redundancy i.e. storing same data or duplicate copies of same data in different locations, can lead to issues like insertion, update and deletion anomalies. Schema refinement or normalization are systematic approaches of decomposing relations to eliminate data redundancy.

5.1 Functional Dependencies

Functional dependency (FD) is a kind of integrity constraint. Let R be a relation schema and let X and Y be non-empty sets of attributes in R. Then, an instance r of R satisfies FD $X \to Y$ (read as X functionally determines Y, or simply X determines Y) if the following holds for every pair of tuples t_1 and t_2 in r:

$$If\ t_1.X = t_2.X,\ then\ t_1.Y = t_2.Y$$

FDs can help to refine subjective decisions made during conceptual design. The set of all FDs implied by a given set F of FDs is called the closure of F and is denoted as F^+.

5.1.1 Armstrong's Axioms

can be applied to infer all FDs implied by a set F of FDs. Let X, Y and Z be sets of attributes over a relation schema R. Now, the axioms are:

- Reflexivity: If $X \supseteq Y$, then $X \to Y$
- Augmentation: If $X \to Y$, then $XZ \to YZ$ for any Z
- Transitivity: If $X \to Y$ and $Y \to Z$, then $X \to Z$

Armstrong's axioms are sound in the sense that they generate only FDs in F^+ when applied to a set F of FDs. They are complete in the sense that repeated application of the rules will generate all FDs in the closure F^+. Also, there is a set of derived rules which are handy:

- Union: If $X \to Y$ and $X \to Z$, then $X \to YZ$
- Decompostion: If $X \to YZ$, then $X \to Y$ and $X \to Z$
- Pseudo-transitivity: If $X \to YZ$ and $Y \to W$, then $X \to WZ$

Prime Attributes refers to attributes which are part of any candidate key of the relation. Otherwise, referred to as non-prime attributes.

Trivial FD : If $X \to Y$ and $Y \subseteq X$ (i.e. Y is a subset of X).

5.1.2 Attribute Closure

If we just want to check whether a given dependency, say, $X \to Y$, is in the closure of a set F of FDs, it can be efficiently done without computing F^+. First the attribute closure X^+ is computed with respect to F, which is the set of attributes A such that $X \to A$ can be inferred using the Armstrong Axioms. The algorithm for computing X^+ is as follows:

```
X+ = X;
For each FD: Y → Z in F, Do{
     If Y ⊆ X+, Then
          X+ = X+ ∪ Z
}
Return X+
```

5.1.3 Derivation of Candidate Keys from FD

Steps that can be used as reference:

(1) Find the attribute(or attributes), say, necessary attributes that will definitely be present in the candidate key. These are the attributes(or attribute) that's not present on the right side of any FD.

(2) Find the attribute closure of necessary attribute(or attributes). If they can determine all the attributes, that is the candidate key. Else,

(3) Combine the necessary attribute(or attributes) with other attributes that are on left side of FDs and check by finding their attribute closure.

For example, let $R = \{A, B, C, D, E\}$ be a relation scheme with the FDs $\{A \to BC$, $CD \to E$, $B \to D$, $E \to A\}$. So, to find the candidate keys,

First, there are no such attributes that are not on the right side of FDs. So, each attribute has to be checked,

- $A^+ = A \to ABC \to ABCD \to ABCDE$.
- $E^+ = E \to EA \to EABC \to EABCD$.
- $B^+ = B \to BD$.
- $C^+ = C$.
- $CD^+ = CD \to CDE \to CDEA \to CDEAB$.
- $BC^+ = BC \to BCD \to BCDE \to BCDEA$.

So, the candidate keys are A, E, CD, and BC. Any combination of attributes that includes these is a super key.

5.2 Decomposition

Decomposition of relation schema R refers to splitting R into smaller relation schemas having attributes that are subset of the attributes of R and together include all attributes of R.

5.2.1 Lossless-Join Decomposition

Decomposition of relation schema R into two schemas with attribute sets X and Y is lossless-join with respect to set of FDs F if for every instance r of R that satisfies the FDs in F,

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

All decompositions must be ***lossless****.*

- The necessary and sufficient condition for decomposition to be lossless-join is that F^+ contains either FD $X \cap Y \to X$ or the FD $X \cap Y \to Y$.
- If an FD $X \to Y$ holds over a relation R and $X \cap Y$ is empty, the decomposition of R into $R - Y$ and XY is lossless. Here, X appears in both $R - Y$ and XY, and it's the key for XY.

5.2.2 Dependency-Preserving Decomposition

The decomposition is dependency-preserving if $(F_X \cup F_Y)^+ = F^+$ i.e. all FDs that are given to hold on R can be enforced by enforcing FDs on X and FDs on Y independently.

Projection of FD Let R be relation schema decomposed into relation schemas with attribute sets X and Y, and let F be set of FDs over R. The projection of F on X (denoted as F_X) is the set of FDs in the closure F^+ that involve only the attributes of X.

5.3 Normal Forms and Normalization

Normalization is a process of designing a consistent database with minimum redundancy which support data integrity by grating or decomposing given relation into smaller relations while preserving constraints on the relation. Several normal forms have been proposed for guidance:

- **1NF**: A relation is in 1NF if every field contains only atomic values (or single values).
- **2NF**: A relation is in 2NF if it is in 1NF and does not contain any partial dependencies.
- **3NF**: Let R be a relation schema, X be a subset of the attributes of R, and let A be and attribute of R. Now, R is in 3NF if for every FD $X \to A$ that holds over R, one of the following statements is true:
  - $A \in X$ i.e. it is a trivial FD, or
  - X is a superkey, or
  - A is part of some (candidate) key for R

  Alternatively, A relation is in 3NF if it is in 2NF and does not contain any transitive dependencies.
- **Boyce-Codd NF**: Let R be a relation schema, X be a subset of the attributes of R, and let A be and attribute of R. Now, R is in BCNF if for every FD $X \to A$ that holds over R, one of the following statements is true:
  - $A \in X$ i.e. it is a trivial FD, or
  - X is a superkey

Relevant terms:

- Partial Dependency: $X \to A$ where X is a proper subset of some key K. Here, (X,A) pairs are stored redundantly.
- Transitive Dependency: $X \to A$ where X is not a proper subset of any key. That means there exist a chain of dependencies $K \to X \to A$ i.e. an X value cannot be assosiated with a K value unless it's also assosiated with an A value.
- Fully-functional Dependency: Y is fully-functional dependent on X if Y is functionally dependent on X ($X \to Y$) but not on any proper subset of X.

Chapter 6

Transactions

Transaction is any one execution of a user program in DBMS, defined as a series of reads and writes of database objects. As it's final action, each transaction either commits (successful completion) or aborts (successful termination).

6.0.1 ACID Properties

Properties that DBMS must ensure in case of system failures and concurrent access. They are as follows:

- **A**tomicity to ensure that either the transaction is carried out completely or doesn't occur at all.
- **C**onsistency to ensure that integrity of data is maintained and database remains in consistent state before and after the transaction.
- **I**solation to ensure that concurrent transactions are protected from interfering with each other.
- **D**urability to ensure that after successful transaction, changes to data must persist and are not undone in case of system failure.

Recovery Manager of a DBMS ensures atomicity if transactions abort and durability if the system crashes or storage media fails. It maintains a log of all modifications to the database, and can undo the actions of aborted and incomplete transactions, and redo the actions of committed transactions.

Two-Phase Locking (2PL) A locking protocol that has two rules:

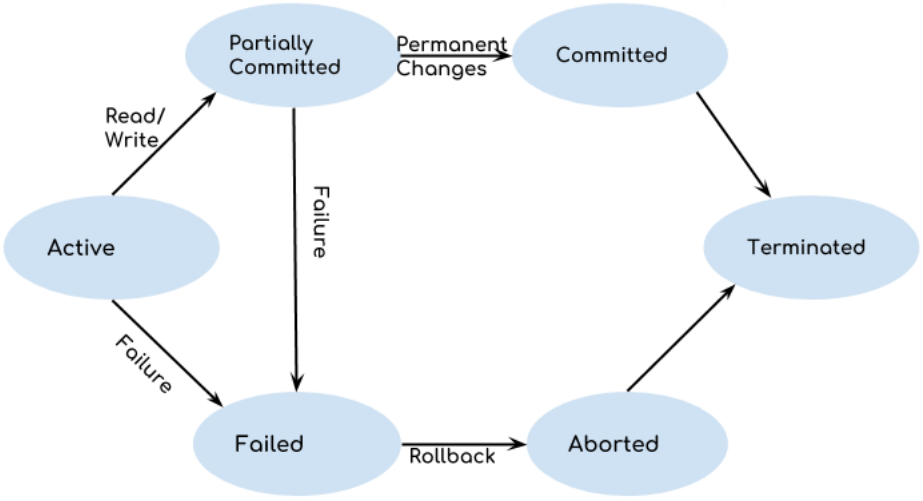
(1) If transaction T wants to read (or modify) and object, it first requests a *shared* (or *exclusive*, respectively) lock on the object.

(2) A transaction cannot request additional locks once it releases *any* lock (i.e. every transaction has 'growing' phase where it acquires locks and 'shrinking' phase where it releases locks).

Note: A variation of 2PL called Strict-2PL exists for which first rule remains the same but second rule is modified as follows:

(2) All locks held by a transaction are released when the transaction is completed.

6.0.2 Transaction States



- Active: If instructions are being executed, DBMS is in active state.
- Partially Committed: All the read and write operations are done but changes are still in main memory and are yet to be committed to actual database.
- Committed: All the changes made by transaction has been applied to the actual database.
- Failed: When some failure occurs during the transaction and the transaction cannot be continued.
- Aborted: After transaction fail, all the changes are reverted to previous consistent state.
- Terminated: After committed state or aborted state, the life cycle of transaction comes to an end with system in consistent state.

6.1 Schedules

Schedule is a potential execution sequence for the actions in a set of transactions.

Serial Schedule refers to when actions of different transactions are not interleaved i.e. transactions are executed from start to finish, one by one.

Recoverable Schedule refers to a schedule in which transactions commit only after all transactions whose changes they read have committed. A schedule *avoids cascading aborts* if it only reads data written by already committed transactions. Types of recoverable schedule:

- **Cascading Rollback**: Failure of one transaction causes several other dependent transactions to rollback or abort.
- **Cascadeless Schedule**: T_i can read the object that is written by T_j only after T_j has committed or aborted.
- **Strict Schedule**: A transaction only reads or writes data written by already committed transactions.

6.2 Concurrent Execution

Concurrent execution of transactions improves system performance by increasing system throughput (number of transactions completed in given time) and response time (average time required to complete a transaction).

6.2.1 Problems in Concurrent Transactions

(1) **Dirty Read**: In a *write-read* conflict, one transaction could read uncommitted data from another transaction. The problem also occurs if the uncommitted transaction fails later and rolls back.

(2) **Unrepeatable Read**: In a *read-write* conflict, a transaction could read a data object twice with different results.

(3) **Lost Update**: In a *write-write* conflict, a transaction overwrites a data object written by another transaction.

(4) **Phantom Read**: A transaction retrieves a collection of objects twice and sees different result, even though it does not modify the objects itself.

6.3 Serializability

When a complete serial schedule is executed on a consistent database, also results in consistent database. There are two types: (i)Conflict serializability and (ii)View serializability.

Serializable Schedule is a schedule that has identical effect on the database to that of a complete serial schedule. DBMS ensures atomicity by *undoing* the actions of incomplete transactions.

6.3.1 Conflict Serializability

Conflicting Operations - Two operations belonging to differet transactions are in conflict if they are working on the same data item, and at least one of them is write operation. So, $\{R_1(X), W_2(X)\}$, $\{W_1(X), W_2(X)\}$, and $\{W_1(X), R_2(X)\}$ are conflict pairs.

Conflict Equivalent schedules contain same set of transactions and order every pair of conflicting actions in the same way. They can transform into one another by swapping non-conflicting operations.

Conflict Serializable schedule is one which is conflict equivalent to some serial schedule. The conflict serializability of a schedule (and corresponding serial schedule) can be found using *precedence graph*. Conflict serializability is sufficient but not necessary condition for serializability.

Precedence Graph (or serializability graph) is useful to capture potential conflicts between transactions in a schedule. The graph for schedule S contains:

- Node for each committed transaction in S, and
- Directed edge from T_i to T_j if an action of T_i precedes and conflicts with one of T_j's actions.

A schedule is conflict serializable if and only if its precedence graph is **acyclic**. And, an equivalent serial schedule is given by any **topological sort** over the precedence graph. Also, the presence of cycle in precedence graph does **NOT** imply that schedule must be non-serializable.

Example: Checking whether the given schedule S is conflict serializable or not.

T1	T2	T3	T4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

Here, the conflicting pairs are:

(a) $\{R_4(A), W_2(A)\}$

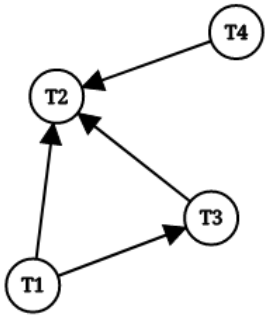
(b) $\{R_3(A), W_2(A)\}$

(c) $\{W_1(B), R_3(B)\}$

(d) $\{W_1(B), W_2(B)\}$

(e) $\{R_3(B), W_2(B)\}$

Now, precedence graph for schedule S will be,



Since there are no cycles in the graph, S is conflict serializable. The serialized schedule will be a topological sort i.e. $<T_1, T_3, T_4, T_2>$.

6.3.2 View Serializability

Two schedules *S1* and *S2* over same set of transactions are **view equivalent** if they satisfy the following conditions:

(1) If T_i reads the initial value of object A in *S1*, it must also read the initial value of A in *S2*.

(2) If T_i reads a value of A written by T_j in *S1*, it must also read the value of A written by T_j in *S2*.

(3) For each data object A, the transaction (if any) that performs the final write on A in *S1* must also perform the final write on A in *S2*.

View Serializable schedule is one which is view equivalent to some serial schedule. Some key points regarding view serializable schedules are:

- Every conflict serializable schedule is view serializable, but the converse is not true.
- Writing without reading an object is called **blind write**.
- Any view serializable schedule that is not conflict serializable contains a blind write. Blind write is necessary but not sufficient condition for a schedule to be view serializable.
- Finding if a schedule is view serializable is NP-complete problem. To proceed, check if schedule is conflict serializable, if not, check if it contains a blind write, again if not, the schedule may or may not be view serializable.

Chapter 7

Data Storage and Indexing

Data must be in main memory for operations. The unit of data transfer between disk and main memory is a **block**. If a single page on a block is needed, the entire block is transferred. Reading or writing a disk block is called I/O operation.

Access Time The access time for a record is defind by

$$Access\ time = Seek\ time + Rotational\ delay + Transfer\ time$$

where,

- Seek time is time taken to move the disk heads to the track on which desired block is located.
- Rotational delay is time taken for the desired block to rotate under the disk head.
- Transfer time is time taken by disk to rotate over the block and read or write the block.

7.0.1 File Structures to store records in DBMS

(a) Heap files, which is an unordered collection of records and can be organized as linked list of data pages. I/O operations require linear time complexity.

(b) Sorted files, which is an ordered collection of records based on some search key. Time complexity of operations can be reduced using binary search.

(c) Hashed files, which contains a table of entries with key-value pair with value as, say, location of required block. Operations when using this structure is even faster than that of sorted files.

7.1 Indexes

An index is a data structure that speeds up operation on file. A *search key* is used, which is a (or set of) record field, on the index which has enough information to retrieve data entries based on the search key.

7.1.1 Key Terms

- Search key (k): key used to search a record in the index.
- Data entry (k*): entry in the index pointed to by k. It contains record id(s) or record itself.
- Records: actual tuples that are pointed to by the record ids.

7.1.2 Classification of Indexes

(a)
- Clustered Index: The order of records in the file matches the order of data entries in the index. These are expensive to maintain as after operations like insertion and update, the file must be reorganized to ensure order and performance.
- Unclustered Index : The order of data entries in index does not match the physical order of actual data.

(b)
- Dense Index: There exists at least one data entry per search key that appears in the file.
- Sparse Index: It contains one entry for each page of records in the data file, where each page is a group of data entries.

(c)
- Primary Index: If the search key includes the primary key, then the index is called primary index.
- Secondary Index: The index that is not primary index.

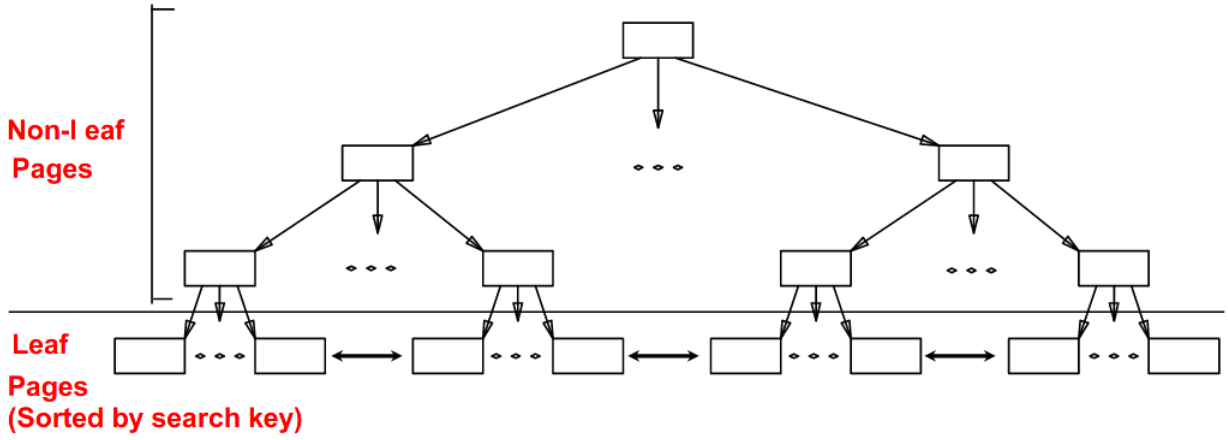
7.2 Methods for Indexing

7.2.1 Tree-based

Ideal for range selections and equality selection.

Index sequential access method (ISAM) is a static tree-structured index where only leaf pages are modifies by inserts and deletes. If a leaf page is full, an *overflow page* is added and chained to the leaf page.

B+ Tree is dynamic, height-balanced index structure providing efficient operations. Each node except the root has between d and $2d$ entries, and d is called the *order* of the tree. The leaf nodes contain data entries, and leaf pages are chained in a doubly linked list. During insertion, nodes that are full are *split* to avoid overflow pages, this might increase the height of the tree. Cost of insertion and deletion is $\log_F N$.



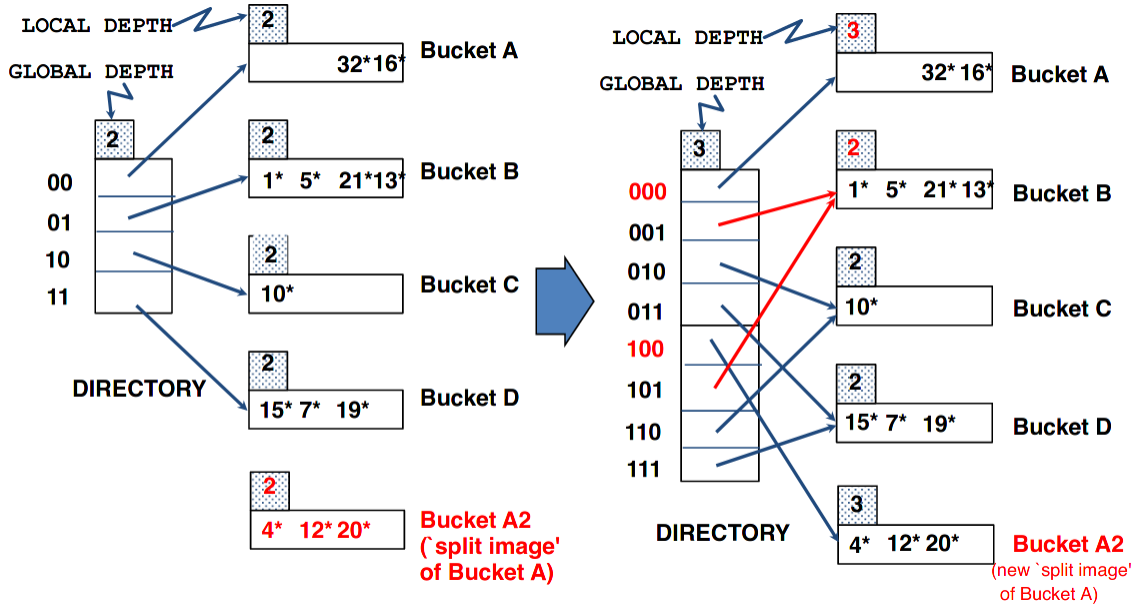
Index using B+ tree

7.2.2 Hash-based

Best for equality queries. A *hashing function* is applied to a search field value and return a *bucket number*, which corresponds to page containing the records.

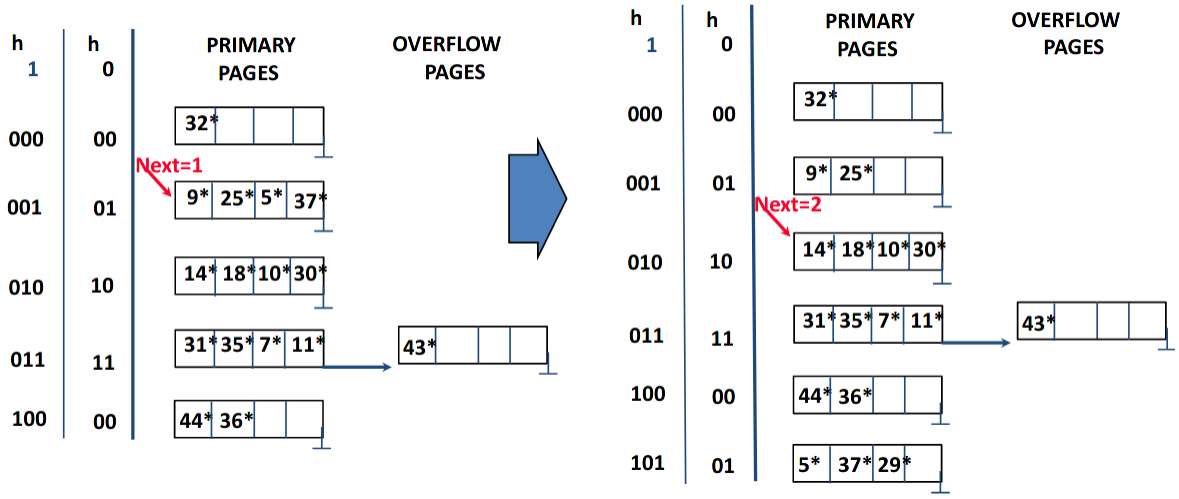
Static Hashing index has fixed number of primary buckets. If primary bucket data entry is full, an overflow page is allocated and linked to primary bucket. Long overflow chains can slow down operations.

Extendible Hashing extends static hashing by introducing a level of indirection in the form of *directory*. Usually the size of directory is 2^d for some d, which is called the global depth of index. The correct directory entry is found by looking at first d bits of the result of hashing function. If a page is full and new data entry is to be added, data entries from that page is redistributed according to first l bits of the hashed values. l is the local depth of the page. Sometimes, insertions like this need doubling of directory to distinguish between two split buckets.



Extendible hashing (bucket split and directory doubling)

Linear Hashing avoids directory structure like in extendible hashing by having a predefined order of buckets to split i.e. in round-robin fashion. Space utilization might be lower because here, bucket splits are not concentrated where data density is the highest.



Linear hashing (inserting 29*)