**Chapter 1**

A **database** is a collection of interrelated data items

**A DBMS (DataBase Management System)** provide a way to store and retrieve information about a particular enterprise

* 1. **Collection of interrelated data**
  2. **Set of programs to access the data**
  3. **An environment that is both convenient , efficient, credible, and secure to use**

**A DBS (DataBase System) contains the following components:**

* 1. **Database**
  2. **DBMS**
  3. **Hardware to support DBMS**
  4. **Operators**

**Levels of Abstraction:**

**Physical level:** describes how a record (e.g., customer) is stored.

**Logical level:** describes what data are stored in database, and what relationships exist among those data.

**type** customer = **record**

customer\_id : string;   
 customer\_name : string;  
 customer\_street : string;  
 customer\_city : integer;

**end**;

**View level:** application programs hide details of data types. Views can also hide information (such as an employee’s salary) for security purposes.

**Schema** – the logical structure of the database

* 1. Example: The database consists of information about a set of customers and accounts and the relationship between them)
  2. Analogous to type information of a variable in a program
  3. **Physical schema(Internal schema):** database design at the physical level
  4. **Logical schema(Conceptual schema):** database design at the logical level
  5. **Subschema(external schema):** describe different views of the database

**Instance** – the actual content of the database at a particular point in time

**Physical Data Independence** – the ability to modify the physical schema without changing the logical schema

* 1. Applications depend on the logical schema
  2. In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

**Logical Data Independence** – the ability to modify the logical schema without updating application

**Data Model**: A collection of conceptual tools for describing

* 1. Data
  2. Data relationships
  3. Data semantics
  4. Data constraints

Two classes of languages

* 1. **Procedural** – user specifies what data is required and how to get those data
  2. **Declarative (nonprocedural)** – user specifies what data is required without specifying how to get those data

**SQL**: widely used non-procedural language

**Chapter 2**

Formally, given sets D1, D2, …. Dn a **relation** r is a subset of   
 D1 x D2  x … x Dn  
Thus, a relation is a set of n-tuples (a1, a2, …, an) where each ai ∈ Di

**Attribute Types:**

1. Each attribute of a relation has a name
2. The set of allowed values for each attribute is called the **domain** of the attribute
3. Attribute values are (normally) required to be **atomic**; that is, indivisible
   1. Note: multivalued attribute values are not atomic
   2. Note: composite attribute values are not atomic
4. The special value null is a member of every domain
5. The null value causes complications in the definition of many operations
   1. We shall ignore the effect of null values in our main presentation and consider their effect later

**Relation Instance**

1. The current values (relation instance) of a relation are specified by a table
2. An element t of r is a **tuple**, represented by a row in a table

K is a **superkey** of R if values for K are sufficient to identify a unique tuple of each possible relation r(R)

K is a **candidate key** if K is minimal

Six basic operators

* 1. select: σ
  2. project: ∏
  3. union: ∪
  4. set difference: –
  5. Cartesian product: x
  6. rename: ρ

**Select Operation**

1. Notation: σ p(r)
2. p is called the **selection predicate**
3. Defined as:  
     
    σp(**r**) = {t | t ∈ r **and** p(t)}

Where p is a formula in propositional calculus consisting of **terms** connected by : ∧ (**and**), ∨ (**or**), ¬ (**not**)  
Each **term** is one of:

<attribute> op <attribute> or <constant>

where op is one of: =, ≠, >, ≥. <. ≤

Example of selection:  
 σ branch\_name=“Perryridge”(account)

**Project Operation**

1. Notation:  
   where A1, A2 are attribute names and r is a relation name.
2. The result is defined as the relation of k columns obtained by erasing the columns that are not listed
3. Duplicate rows removed from result, since relations are sets

Example: To eliminate the branch\_name attribute of accou

**Union Operation**

1. Notation: r ∪ s
2. Defined as:
3. r ∪ s = {t | t ∈ r or t ∈ s}
4. For r ∪ s to be valid.

1). r, s must have the same arity (same number of attributes)

2). The attribute domains must be compatible (example: 2nd column of r deals with the same type of values as does the 2nd  column of s)

Example: to find all customers with either an account or a loan  
 ∏customer\_name (depositor) ∪ ∏customer\_name (borrower)

**Set Difference Operation**

1. Notation r – s
2. Defined as:

r – s = {t | t ∈ r **and** t ∉ s}

1. Set differences must be taken between **compatible** relations.
   1. r and s must have the same **arity**
   2. attribute domains of r and s must be compatible

**Cartesian-Product Operation**

1. Notation r x s
2. Defined as:

r x s = {t q | t ∈ r **and** q ∈ s}

1. Assume that attributes of r(R) and s(S) are disjoint. (That is, R ∩ S = ∅).
2. If attributes of r(R) and s(S) are not disjoint, then renaming must be used.

**Rename Operation:**

1. Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
2. Allows us to refer to a relation by more than one name.
3. Example:
   * x (E)
   * returns the expression E under the name X
4. If a relational-algebra expression E has arity n, then returns the result of expression E under the name X, and with the attributes renamed to A1 , A2 , …., An .

**Additional Operations**

1. Set intersection
2. Natural join
3. Division
4. Assignment

**Set-Intersection Operation**

1. Notation: r ∩ s
2. Defined as:
3. r ∩ s = { t | t ∈ r **and** t ∈ s }
4. Assume:
   1. r, s have the same arity
   2. attributes of r and s are compatible
5. Note: r ∩ s = r – (r – s)

**Natural-Join Operation**

1. Notation: r s
2. Let r and s be relations on schemas R and S respectively.   
   Then, r s is a relation on schema R ∪ S obtained as follows:
   1. Consider each pair of tuples tr from r and ts from s.
   2. If tr and ts have the same value on each of the attributes in R ∩ S, add a tuple t to the result, where
      1. t has the same value as tr on r
      2. t has the same value as ts on s

**Division Operation**

1. Notation:
2. Suited to queries that include the phrase “for all”.
3. Let r and s be relations on schemas R and S respectively where
   1. R = (A1, …, Am , B1, …, Bn )
   2. S = (B1, …, Bn)

The result of r ÷ s is a relation on schema

R – S = (A1, …, Am)

r ÷ s = { t | t ∈ ∏ R-S (r) ∧ ∀ u ∈ s ( tu ∈ r ) }

Where tu means the concatenation of tuples t and u to produce a single tuple

**Assignment Operation**

The assignment operation (←) provides a convenient way to express complex queries

**Aggregation function** takes a collection of values and returns a single value as a result.

**avg**: average value  
 **min**: minimum value  
 **max**: maximum value  
 **sum**: sum of values  
 **count**: number of values



**Aggregate operation** in relational algebra E is any relational-algebra expression

* 1. G1, G2 …, Gn is a list of attributes on which to group (can be empty)
  2. Each Fi is an aggregate function
  3. Each Ai is an attribute name
  4. The schema of result is (G1, G2 …, Gn , a1, a2 …, am ).where ai is the result of applying the aggregate function Fi on the multiset of values for attribute Ai in the group

**Chapter 3**

**Domain Types in SQL**

1. **char(n).** Fixed length character string, with user-specified length n.
2. **varchar(n).**  Variable length character strings, with user-specified maximum length n.
3. **int.** Integer (a finite subset of the integers that is machine-dependent).
4. **smallint.** Small integer (a machine-dependent subset of the integer domain type).
5. **numeric(p,d).** Fixed point number, with user-specified precision of p digits (including a sign), with d of the p digits are to the right of decimal point.
6. **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
7. **float(n).** Floating point number, with user-specified precision of at least n digits.

An SQL relation is defined using the **create table** command:

**create table** r (A1 D1, A2 D2, ..., An Dn,  
 (integrity-constraint1),  
 ...,  
 (integrity-constraintk))

* 1. r is the name of the relation
  2. each Ai is an attribute name in the schema of relation r
  3. Di is the data type of values in the domain of attribute Ai

1. The **drop table** command deletes all information about the dropped relation from the database.
2. The **alter table** command is used to add attributes to an existing relation:

**alter table** r **add** A D

where A is the name of the attribute to be added to relation r and D is the domain of A.

All tuples in the relation are assigned null as the value for the new attribute.

1. The **alter table** command can also be used to drop attributes of a relation:

**alter table** r **drop** A

where A is the name of an attribute of relation r

Dropping of attributes not supported by many databases

To force the elimination of duplicates, insert the keyword **distinct**  after select**.**

The **select** clause can contain arithmetic expressions involving the operation, +, –, \*, and /, and operating on constants or attributes of tuples.

The **where** clause specifies conditions that the result must satisfy

The **from** clause lists the relations involved in the query

The SQL allows renaming relations and attributes using the **as** clause:

old-name **as** new-name

The operator “**like**” uses patterns that are described using two special characters.

**Multiset** versions of some of the relational algebra operators – given multiset relations r1 and r2:

1. **σθ (r1):** If there are c1 copies of tuple t1 in r1, and t1 satisfies selections σθ,, then there are c1 copies of t1 in σθ (r1).

2. **ΠA (r ):** For each copy of tuple t1 in r1, there is a copy of tuple ΠA (t1) in ΠA (r1) where ΠA (t1) denotes the projection of the single tuple t1.

3. **r1  x r2 :** If there are c1 copies of tuple t1 in r1 and c2 copies of tuple t2 in r2, there are c1 x c2 copies of the tuple t1. t2 in r1  x r2

**Relevant Nested Subqueries:** the query condition of subquery depends on outer query

The **unique** construct tests whether a subquery has any duplicate tuples in its result.

The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

**Views Defined Using Other Views**

1. One view may be used in the expression defining another view
2. A view relation v1 is said to depend directly on a view relation v2 if v2 is used in the expression defining v1
3. A view relation v1 is said to depend on view relation v2 if either v1 depends directly to v2  or there is a path of dependencies from v1 to v2
4. A view relation v is said to be recursive if it depends on itself.

**Chapter 4**

**User-Defined Types**

1. **create type** construct in SQL creates user-defined type

**create type** Dollars **as numeric (12,2) final**

1. **create domain** construct in SQL-92 creates user-defined domain types

**create domain** person\_name **char**(20) **not null**

1. Types and domains are similar. Domains can have constraints, such as **not null**, specified on them. Domains are not strongly typed.

**Domain constraints** (约束)are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.

**Referential Integrity**

Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:

* 1. The primary key clause lists attributes that comprise the primary key.
  2. The unique key clause lists attributes that comprise a candidate key.
  3. The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

**Authorization**

Forms of authorization on parts of the database:

1. **Read** - allows reading, but not modification of data.
2. **Insert** - allows insertion of new data, but not modification of existing data.
3. **Update** - allows modification, but not deletion of data.
4. **Delete** - allows deletion of data.

**Authorization Specification in SQL**

1. The grant statement is used to confer authorization

**grant <privilege list> on <relation name or view name> to <user list>**

1. <user list> is:
   1. a user-id
   2. **public**, which allows all valid users the privilege granted
   3. A role (more on this in Chapter 8)
2. Granting a privilege on a view does not imply granting any privileges on the underlying relations.
3. The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

**with grant option**

**grant select on** branch **to** U1 **with grant option**

**Revoking Authorization in SQL**

The revoke statement is used to revoke authorization.

**revoke <privilege list> on <relation name or view name> from <user list>**

**Chapter 6**

**Design Process**

1. Characterize fully the data needs of the prospective data users
   1. A specification of user requirement
2. **Conceptual-design** --- Chooses a data model, translates the requirements into a conceptual schema of database
   1. **Redundancy**
   2. **Incompleteness**
3. Review the schema to ensure it meets functional requirements
4. Implementation
   1. **Logical-design**
   2. **Physical-design**

**Modeling**

1. A database can be modeled as:

a collection of entities,

relationship among entities.

1. An **entity** is an object that exists and is distinguishable from other objects.
2. Entities have **attributes**

Example: people have names and addresses

1. An **entity set** is a set of entities of the same type that share the same properties.

A **relationship** is an association among several entities

**Relationship Sets:**A relationship is an association among several entities

1. The association between entity sets is referred to as **participation**
   1. The entity sets E1,E2,…,En participate in relationship set R
   2. **Relationship instance** represents an association between the named entities in the real-world enterprise that is being modeled
2. **Role**---The function that an entity plays in a relationship
3. **Recursive relationship set**---the same entity set participate in a relationship set more than once

**Descriptive attribute** --An **attribute** can also be property of a relationship set.

**Degree of a Relationship Set**

**Degree**-Refers to number of entity sets that participate in a relationship set.

Relationship sets that involve two entity sets are **binary** (or degree two). Generally, most relationship sets in a database system are binary.

**Attributes:**

**Domain –** the set of permitted values for each attribute

**Mapping Cardinality Constraints**

**Keys:**

1. A **super key** of an entity set is a set of one or more attributes whose values uniquely determine each entity.
2. A **candidate key** of an entity set is a minimal super key
3. Although several candidate keys may exist, one of the candidate keys is selected to be the **primary key**.

**Participation of an Entity Set in a Relationship Set**

1. **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
2. **Partial participation:** some entities may not participate in any relationship in the relationship set

**Weak Entity Sets**

1. An entity set that does not have a primary key is referred to as a **weak entity set**. e.g. Employee and Family member, Contract and Attachment
2. The existence of a weak entity set depends on the existence of a **identifying entity set**
   1. it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
   2. **Identifying relationship** depicted using a double diamond
3. The **discriminator** (or partial key) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.

**Extended E-R Features: Specialization**

**Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

**Extended ER Features: Generalization**

**A bottom-up design process –** combine a number of entity sets that share the same features into a higher-level entity set.

The ISA relationship also referred to as **superclass - subclass** relationship

**Design Constraints on a Specialization/Generalization**

1. Constraint on which entities can be members of a given lower-level entity set.
   1. condition-defined

Example: all customers over 65 years are members of senior-citizen entity set; senior-citizen ISA person.---**attribute-defined**

* 1. user-defined

1. Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
   1. **Disjoint**
      1. an entity can belong to only one lower-level entity set
      2. Noted in E-R diagram by writing disjoint next to the ISA triangle
   2. **Overlapping**
      1. an entity can belong to more than one lower-level entity set

**Redundancy of Schemas**

1. For one-to-one relationship sets, **either side** can be chosen to act as the “many” side
   1. That is, extra attribute can be added to either of the tables corresponding to the two entity sets
2. If participation **is partial** on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could **result in null values**
3. The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
   1. Example: The payment schema already contains the attributes that would appear in the loan\_payment schema (i.e., loan\_number and payment\_number).

**Chapter 7**

Denote as a **functional dependency**:

loan\_number → amount

A functional dependency is a generalization of the notion of a **key**.

We use functional dependencies to:

* 1. test relations to see if they are legal under a given set of functional dependencies.
     1. If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F.
  2. specify constraints on the set of legal relations
     1. We say that F **holds on** R if all legal relations on R satisfy the set of functional dependencies F.

A functional dependency is **trivial** if it is satisfied by all instances of a relation

**First Normal Form**

1. Domain is **atomic** if its elements are considered to be indivisible units
2. A relational schema R is in **first normal form** if the domains of all attributes of R are atomic

**Closure of a Set of Functional Dependencies**

1. The set of all functional dependencies logically implied by F is the closure of F.
2. We denote the closure of F by F+.
3. Given a relational schema R, a functional dependency f on R is **logically implied** by a set functional dependencies F on R if every relation instance r(R) that satisfies F also satisfies f.

**Closure of Attribute Sets:**B is **functionally determined** by a if a→ B

**Extraneous Attributes:**

Consider a set F of functional dependencies and the functional dependency α → β in F.

* 1. Attribute A is **extraneous** in α if A ∈ α   
      and F logically implies (F – {α → β}) ∪ {(α – A) → β}.
  2. Attribute A is **extraneous** in β if A ∈ β   
      and the set of functional dependencies   
      (F – {α → β}) ∪ {α →(β – A)} logically implies F.

**Testing if an Attribute is Extraneous:**

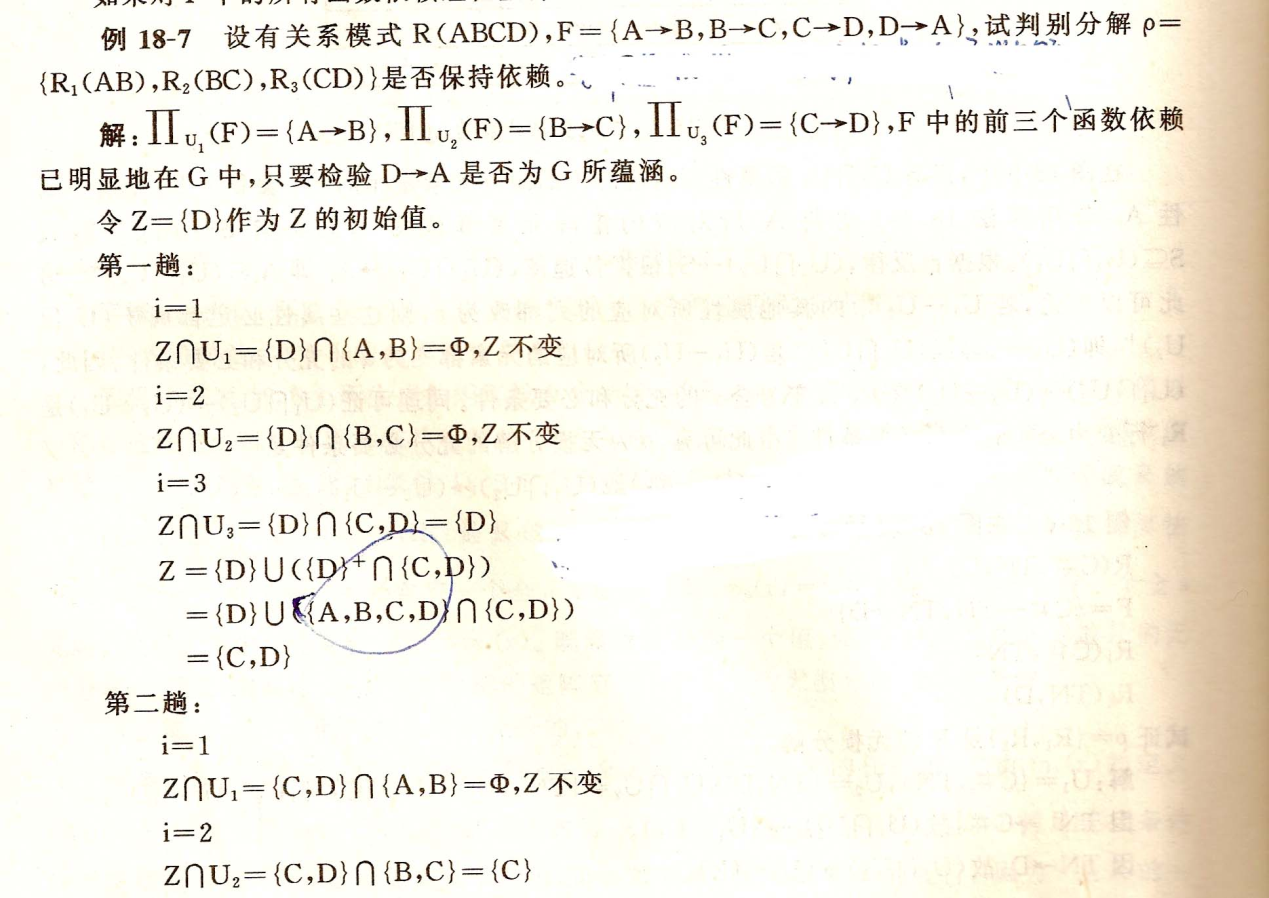
To test if attribute A ∈ β is extraneous in β

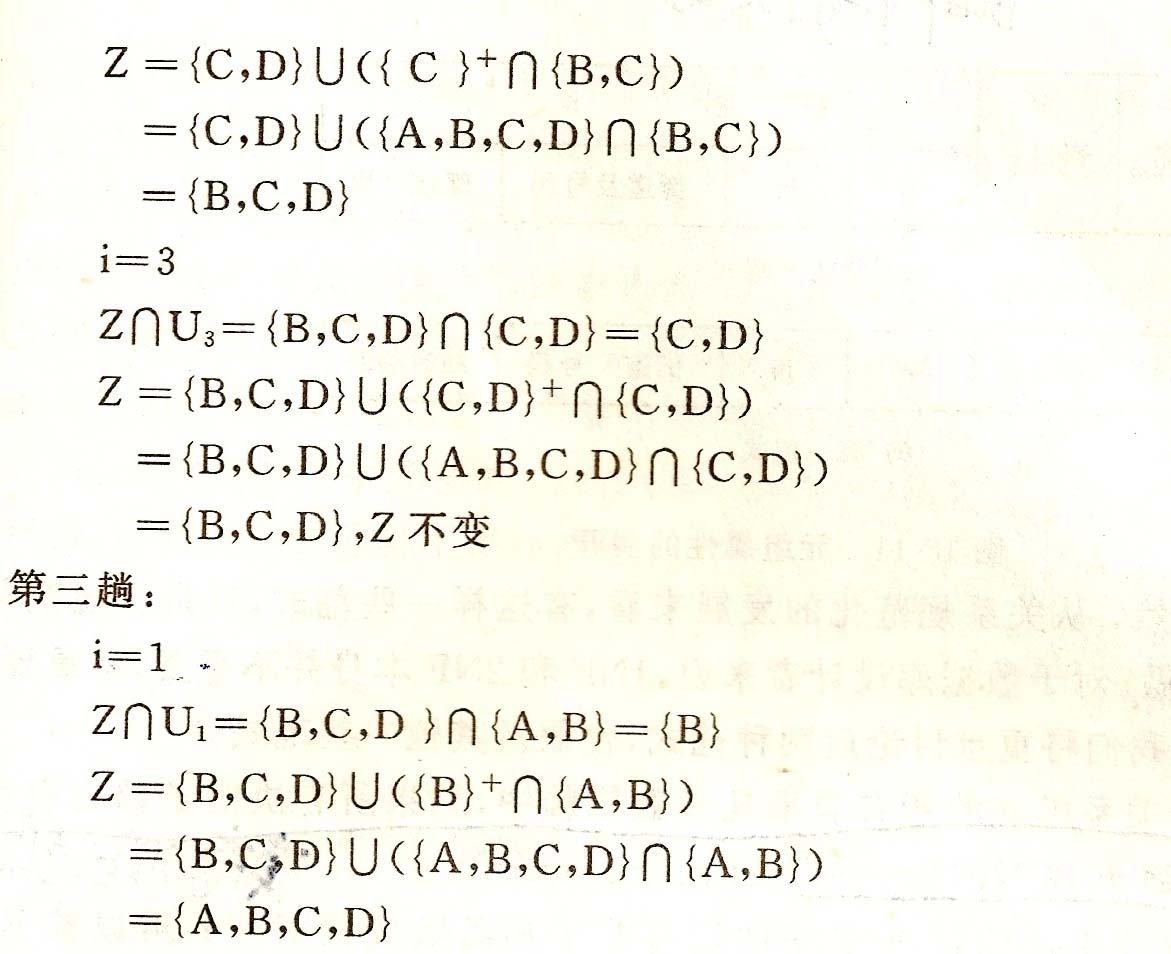
* 1. compute α+  using only the dependencies in   
      F’ = (F – {α → β}) ∪ {α →(β – A)},
  2. check that α+  contains A; if it does, A is extraneous

**A canonical cover(正则覆盖) for F** is a set of dependencies Fc such that

* 1. F logically implies all dependencies in Fc, and
  2. Fc logically implies all dependencies in F, and
  3. No functional dependency in Fc contains an extraneous attribute
  4. Each left side of functional dependency in Fc is unique.

**Dependency Preservation**





**Chapter 12**

**File Organization:**

1. **Heap** – a record can be placed anywhere in the file where there is space
2. **Sequential** – store records in sequential order, based on the value of the search key of each record

if no free space, insert the record in an **overflow block**

1. **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

**Basic Concepts of Index**

1. Indexing mechanisms used to speed up access to desired data.
   1. E.g., author catalog in library
2. **Search Key** - attribute to set of attributes used to look up records in a file.
3. An **index file** consists of records (called **index entries**) of the form |search key|pointer|
4. Index files are typically much smaller than the original file
5. Two basic kinds of indices:
   1. **Ordered indices:** search keys are stored in sorted order
   2. **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

**Ordered Indices:**

1. In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
2. **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
   1. Also called **clustering index**
   2. The search key of a primary index is usually but not necessarily the primary key.
3. **Secondary index**:an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
4. Index-sequential file**:** ordered sequential file with a primary index.

**Dense Index Files:** **Dense index** — Index record appears for every search-key value in the file (not every record).

**Sparse Index Files:** **Sparse Index:**  contains index records for only some search-key values.

**Static Hashing:**

1. A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
2. In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash** **function.**
3. Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
4. Hash function is used to locate records for access, insertion as well as deletion.
5. Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

**Hash Functions:**

1. An ideal hash function is **uniform***,* i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
2. Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.

**Handling of Bucket Overflows:** Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow bucket****s.*

**Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.

Above scheme is called **closed hashing.**

An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.

A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.

**Index Definition in SQL**

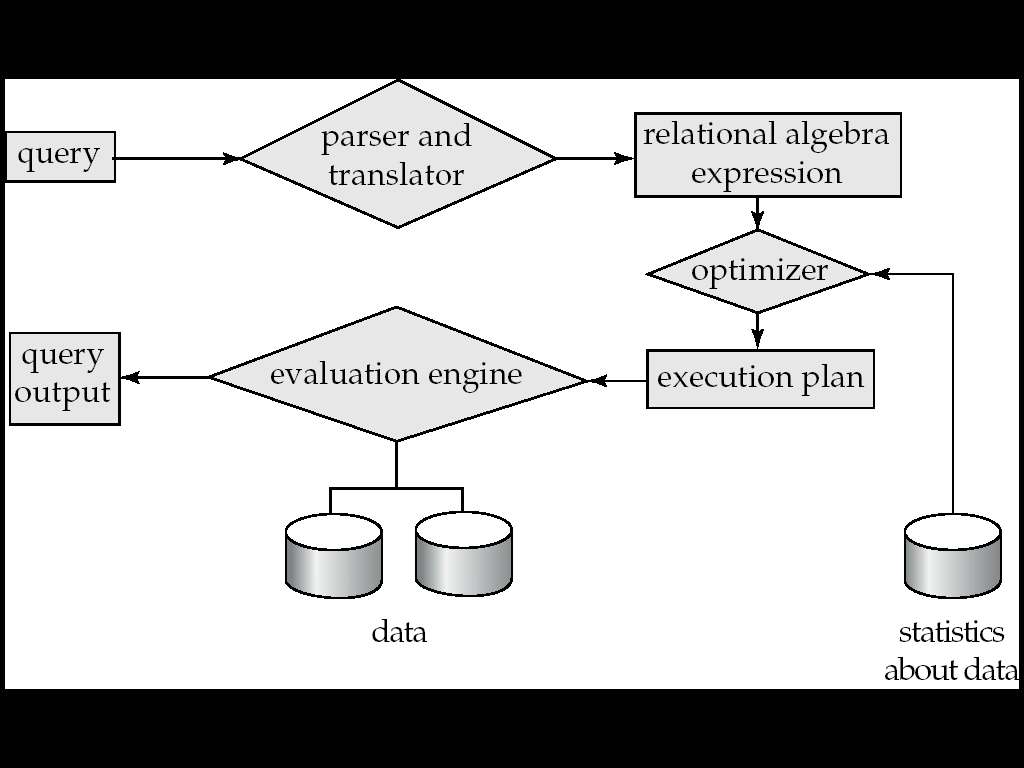
1. Create an index
   * 1. **create index** <index-name> **on** <relation-name> (<attribute-list>)
2. E.g.: **create index**  *b-index* **on** *branch(branch\_name)*
3. Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key .
   1. Not really required if SQL **unique** integrity constraint is supported
4. To drop an index
   * + 1. **drop index** <index-name>

**Chapter 13**

**BASIC \_STEPS:**

1. Parsing and translation
2. Optimization
3. Evaluation

图：



Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan.**

**Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.

**选择操作:**

Basic Algorithms:

A1 Linear sreach

A2 Binary search

**Nested-Loop Join:**

*r* is called the **outer** **relation** and *s* the **inner relation** of the join.

**Hash-Join algorithm:**

**Recursive partitioning**required if number of partitions *n* is greater than number of pages *M* of memory.

Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others

**Hash-table overflow** occurs in partition *si* if *si* does not fit in memory. Reasons could be

* 1. Many tuples in s with same value for join attributes
  2. Bad hash function

**Chapter 14**

**Introduction**

1. Generation of query-evaluation plans for an expression involves several steps:
   1. Generating logically equivalent expressions using **equivalence rules**.
   2. Annotating resultant expressions to get alternative query plans
   3. Choosing the cheapest plan based on **estimated cost**
2. The overall process is called **cost based optimization.**

**Transformation of Relational Expressions**

1.Two relational algebra expressions are said to be **equivalent** if on every legal database instance the two expressions generate the same set of tuples

2.An **equivalence rule** says that expressions of two forms are equivalent

Can replace expression of first form by second, or vice versa

Equivalence Rules P571-574

**Cost-Based Optimization:**

A Cost-Based Optimizer generates a range of query-evaluation plans from the given query by using the equivalence rules, and chooses the one with the least cost.

**Heuristic Optimization:**

1. Cost-based optimization is expensive, even with dynamic programming.
2. Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
3. Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
   1. Perform selection early (reduces the number of tuples)
   2. Perform projection early (reduces the number of attributes)
   3. Perform most restrictive selection and join operations before other similar operations.
   4. Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

**Chapter 15**

1. **The concept of transactions:（ACID）**

**A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:**

**1.Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.

**2.Consistency.** Execution of a transaction in isolation preserves the consistency of the database.

**3.Isolation。**Although multiple transactions may execute concurrently，the system guarantees that, for every pair of transactions *Ti* and *Tj,* it appears to *Ti* that either *Tj,* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished. Thus, each transaction is unaware of other executing concurrently in the system.

**4.Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

(ACID这一缩写来自四条性质的第一个字母)

1. **Transaction State**

**1.Active** –the initial state; the transaction stays in this state while it is executing

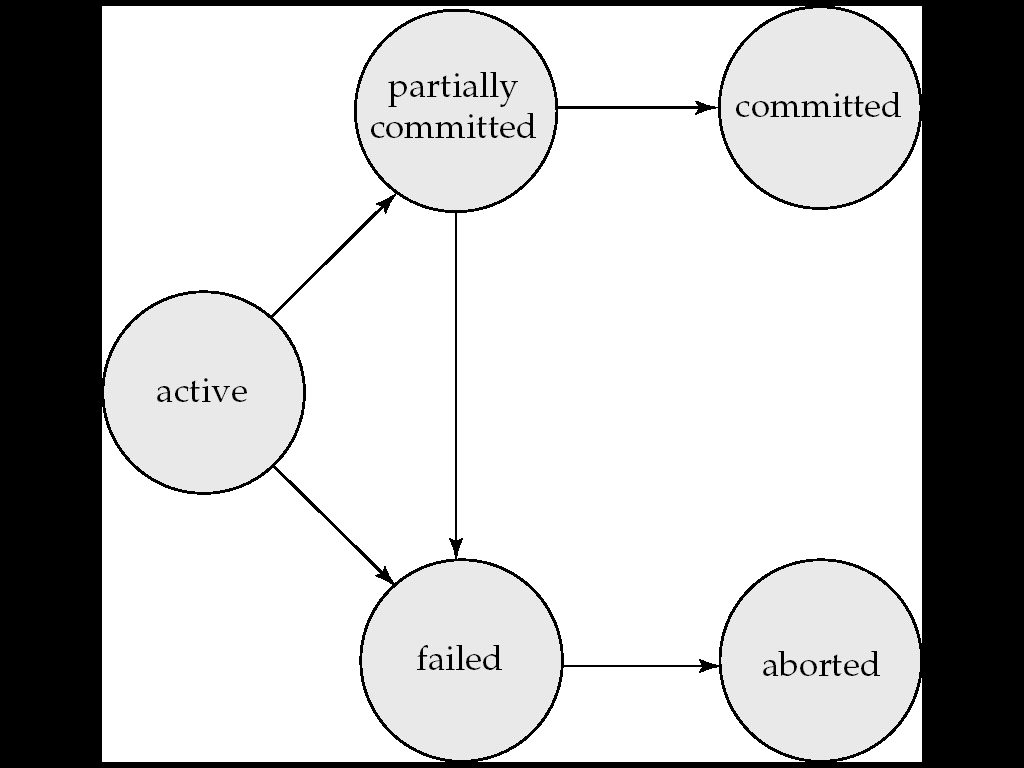
**2.Partially committed** –after the final statement has been executed.

**3.Failed --** after the discovery that normal execution can no longer proceed.

**4.Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.

**5.Committed** – after successful completion.

**Rolled back-** after failed

事务相应的状态图： 

1. **Serializability**

**1).conflict serializability**

1. *li* = read(*Q), lj =* read(*Q*). *li* and *lj* don’t conflict.  
 2. *li* = read(*Q), lj =* write(*Q*). They conflict.  
 3. *li* = write(*Q), lj =* read(*Q*). They conflict  
 4. *li* = write(*Q), lj =* write(*Q*). They conflict

**If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are conflict equivalent*.***

A schedule is conflict serializable if it is conflict equivalent to a serial schedule.

**2).view serializability**

**Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are view equivalentif the following three conditions are met:**

1. For each data item *Q,* if transaction *Ti* reads the initial value of *Q* in schedule *S,* then transaction *Ti*  must, in schedule *S´*, also read the initial value of *Q.*

*2.* For each data item *Q* if transaction *Ti* executes **read**(*Q)* in schedule *S*, and that value was produced by a **write(Q)** operation executed by transaction *Tj* (if any), then transaction *Ti* must, in schedule *S´* also read the value of *Q* that was produced by the same **write(Q)** operation of transaction *Tj* .

3. For each data item *Q*, the transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must perform the final**write**(*Q*) operation in schedule *S´*.

A schedule is view serializable if it is view equivalent to a serial schedule.

Every view serializable schedule that is not conflict serializable has **blind writes.**

1. **Recoverable/ Cascadeless schedule 区别与联系**

**Recoverableschedule** — if a transaction *Tj* reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti*  appears before the commit operation of *Tj.*

**Cascadelessschedules** — cascading rollbacks cannot occur; for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti*  appears before the read operation of *Tj*.

**Chapter 16**

**基于锁的协议：**

**Lock-Base Protocols**

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

**两种加锁模式：**

1. ***Exclusive mode*.**

Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.

If a transaction Ti has obtained an **exclusive-mode lock** (denoted by X) on item Q, then Ti can both read and write Q.

1. ***Shared mode***.

Data item can only be read. S-lock is requested using **lock-S** instruction.

If a transaction Ti has obtained a **shared-mode lock** (denoted by S) on item Q, then Ti can read, but cannot write, Q.

**两阶段提交协议：**

**The Two-Phase Locking Protocol**

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phase:

1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

**Chapter 17**

**Data Access(数据访问)**

**(物理块)Physical blocks are those blocks residing on the disk.**

**(缓冲块)Buffer blocks are the blocks residing temporarily in main memory.**

**Log-Based Recovery(基于日志的恢复)**

A **log** is kept on stable storage.

The log is a sequence of **log records(日志记录)**, and maintains a record of update activities on the database.

1. **Deferred Database Modification(延迟的数据库修改)**

The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.

**2.Immediate Database Modification(立即的数据库修改)**

The **immediate database modification** scheme allows database modification to be output to the database while the transaction is still in active state.

database modifications written by active transactions are called **uncommitted modifications**.(未提交修改)