**第二章: 词法分析**

**Keys**

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

*K* is a ***candidate key(候选码)*** if *K* is minimal(no subset)

**Relational-Algebra Operations**

**Select** *σp*(*r*) = {*t* | *t* ∈ *r* and *p(t)*}

**Project** ∏A1, A2, …, *Ak* (*r*)

**Union** *r* ∪ *s* = {*t* | *t* ∈ *r* or *t* ∈ *s*}

**Set difference** *r – s* = {*t* | *t* ∈ *r* and t ∉ *s*}

**Cartesian product** *r* x *s* = {*t q* | *t* ∈ *r* and *q* ∈ *s*}

**Rename** *ρx[*(*A1, A2, …, An*)] (*E*)

**Set-Intersection** *r* ∩ *s* ={ *t* | *t* ∈ *r* and *t* ∈ *s* }

**Natural-Join** r s

**Theta join** r θs= σθ(r x s)

**Division**

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

*q = r* ÷ *s,* *q* is the largest rel. satisfying *q* x *s* ⊆ *r*

**Assignment** *temp*1 ← ∏*R-S* (*r*)

**Chapter 3: SQL**

**Data Definition**

**Domain Types in SQL**

char(n). varchar(n). int. smallint. numeric(p,d). real, double precision. float(n).

**Create Table Construct**

create table*r* (*A*1 *D*1, *A*2 *D*2, ..., *An Dn,*

integrity-constraint1,…  
integrity-constraintk)

**Integrity Constraints**

not null

primary key (*A*1, ..., *An)*

check *(P),* where *P* is a predicate

unique(*A*1, ..., *An)*

**Drop and Alter Table Constructs**

drop table r

alter table *r* add *A D*

alter table *r* drop *A*

**Basic Structure of SQL Queries**

**Basic Structure**

select *A*1, *A*2, ..., *An*  
from *r*1, *r*2, ..., *rm* where *P*

∏*A*1*, A2, ..., An*(σ*P (r*1 x *r*2 x ... x *rm*))

**The select Clause**

corresponds to the ***projection***

∏branch-name(*loan)*

operation of relational algebra

select [ distinct | all ] *attrlist | \** from *r1,r2,…rn*

all is default.

**Select with operators**

select *amount* \*100 from *loan*

would return a relation that *amount* is multiplied by 100.

**The from Clause**

select A as B

can rename resultant field A to B.

**Tuple Variables**

select *T.loan-number, S.amount*from *borrower* as *T, loan* as *S*where *T.loan-number = S.loan-numbe****r***

**String Operations**

% character matches any substring.

\_ character matches any character.

where *customer-street* like ‘%Main%’

转义符: \ (\\, \%, \\_...)

连接: using “||” or concat()  
大小写: upper(), lower()

其它: length(), substr()…

**Ordering the Display of Tuples**

order by *customer-name* desc , *amount* asc

default: asc,升序

**Duplicates**

In relations with duplicates, SQL can define how many copies of tuples appear in the result.

*Multiset* (多重集) versions of some of the relational algebra operators:

\* σ*θ(r*1*):* If there are *c*1 copies of tuple *t*1 in *r*1, and *t*1 satisfies selections σ*θ* , then there are *c*1 copies of *t*1 in σ*θ (r*1*)*

\* Π*A(r*1*)*: For each copy of tuple *t1* in *r*1*,* there is a copy of tupleΠ*A(t*1*)* in Π*A(r*1*)* where Π*A(t*1*)* denotes the projection of the single tuple *t1.*

\* *r*1 x *r*2 : If there are *c*1 copies of tuple *t1* in *r*1 and *c*2 copies of tuple *t*2 in *r*2, there are *c*1 x *c*2 copies of the tuple *t1. t*2 in *r*1 x *r*2

**Set Operations**

union, intersect, except: **∪,∩,−**

The relations must be compatible

***eliminates duplicates*** by default, to get duplicates, use the following:

union all, intersect all and except all

**Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:**

*m + n* times in *r* union all *s*

min(*m,n)* times in *r* intersect all *s*

max(0, *m – n)* times in *r* except all *s*

**Aggregate Functions**

**Aggregate Functions (均不计算NULL!)**

avg min max sum count

input to avg and sum must be numeric. E.g.

select avg *(balance)*

select count (distinct *customer-name)*

**Grounp By Clause**

select *branch-name,* avg *(balance)*from *account* group by *branch-name*

**Having Clause**

select *branch-name,* avg *(balance)*from *account* group by *branch-name*having avg *(balance) >* 1200

！predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups！

**Null Values**

where *amount* is null | is not null

*5 + null* returns *null*

*5 < null* or *null <> null* or *null = null*

returns *unknown*

(*unknown* or *true*) = *true*, (*unknown* or *false*) = *unknown,* (*unknown* or *unknown) = unknown*

*(true* and *unknown) = unknown, (false* and *unknown) = false, (unknown* and *unknown) = unknown*

*(*not *unknown) = unknown*

！Result of where clause predicate is treated as *false* if it evaluates to *unknown*！

**Nested Subqueries**

**Examples**

where *customer-name* in | not in (select *customer-name* fromdepositor)

where *customer-name* not in ( ‘Smith’, ‘Jones’ )

**Set Comparison**

E.g. Find all branches that have greater assets than some branch located in Brooklyn

where *assets >* some | any  
(select *assets* from *branch*where *branch-city =* ‘Brooklyn’)

any == some,只要和后面集合中的某些项成立就可以！

all, 要和集合中所有的成立！

**Test for Empty Relations**

exists  *r* ⇔ *r* ≠ *Ø*

not exists *r* ⇔ *r* = *Ø*

！We can write “relation A contains relation B” as “not exists (B except A)” (Note that *X – Y = Ø* ⇔ *X* ⊆ *Y)*！

The unique construct tests whether a subquery has any duplicate tuples.

**Complex Queries**

**Derived Relations**

select … from (subquery) as T(r1, r2)

**With Clause**

with T(r1, r2) as (subquery)

**Views(隐藏某属性)**

create view *V* as (subquery)

select … from V …

**Modification of the Database**

**Deletion**

delete from *account* where …

**Insertion** (with NULL)

insert into *account* values (‘A-777’,‘Perryridge’, *null*)

insert into *account (account-number, branch-name)* values (…)

insert into *table* (subquery)

subquery is evaluated at first.

**Updates**

update T set … where P

update T set case

when case1 then up1

else up2

end

**Chapter 4: Advanced SQL**

**SQL Data Types and Schemas**

**Built-in Data Types in SQL**

date ‘2001-7-27’

time ’09:00:30’

timestamp ‘2001-7-27 09:00:30.75’

cast <string-valued-expression> as …

interval ‘1’ day

extract (year | date ... from r.starttime)

**User-Defined Types**

create domain *Dollars* numeric(12, 2)

also drop domain and alter domain

**Constraints in User-Defined Domain**

create domain *…* [constraint *test*]check(*value* > = 4.00)

‘constraint *test’* is optional

check (*branch*-*name* in (subquery))

**Integrity Constraints**

**Integrity constraints** 防止因意外损坏造成数据库数据越界、缺失

**Domain constraints** are the most elementary form of integrity constraint.

**Referential Integrity**

**Ensures** that a value that appears in one relation for a given set of attributes also appears in another relation.

**The** subset α of R2 is a *foreign key* referencing *K*1 in relation *r*1, if for every *t*2 in *r*2 there must be a tuple *t*1 in *r*1 such that *t*1[*K*1] = *t*2[α].

**Referential Integrity on Modification**

**If** t1 is deleted from r1, referenced tuples in t2: **σα = *t*1[K] (*r*2)**, return error or cascading deletion.

**Insertion** and **update** in r2 must check the referential integrity.

**Update** of K in r1, similar to deletion: return error or cascading update.

**Referential Integrity in SQL**

Primary key – primary key

Foreign keys

foreign key (*account-number*) references *account*

*account-number* char (10) references *account*

foreign key (*account-number*) references *account*(*account-number*)

**Cascading Actions in SQL**

foreign key *(branch-name)* references *branch*

on delete cascade  
on update cascade

**or:** on delete set null

**or:** on delete set default

！best prevented using null

Referential integrity is only checked at the end of a transaction. If a cascading causes a constraint violation, the system aborts the transaction. ！

**Assertions (e.g. ppt4.22)**

create assertion <assertion-name> check <predicate>

test it on every update that may violate the assertion, may case a ***significant amount of overhead***

**Authorization**

**Security**

DB/OS/Network/Physical/Human

**Forms of Authorization**

**To data**: insert/update/delete/read

**To schema**:

Index: create/delete indicies

Resource: create relations

Alteration: addition/deletion attr in rel.

Drop: deletion of relations

！A user with resource authorization who creates a new relation is given all privileges on that relation automatically

**Authorization and Views**

可给予用户View权限而不给予relation权限，对部分用户隐藏部分数据的方法

The creator of a view gets only those privileges that provide no additional authorization beyond that he had.

E.g. if creator of view *cust-loan* had only read auth on *borrower* and *loan*, he gets only read auth on *cust*-*loan*

**Granting of Privileges**

权限图(authorization graph)

图根为 the database administrator.

权限级联收回(不存在其他与DBA通路时被收回)

循环授权在环中任意点与DBA无通路时，环自动删除

**Roles**

Roles: 一类用户，授权时与用户类似，可指定用户的Role

create role *teller | manager*

grant *teller* to *manager*

grant *teller* to *alice, bob*

**Authorization Specification in SQL**

grant [<privilege list>

on <relation | view >] | [roles] to <user list> | public | <role list>

**Privileges in SQL**

select/insert/update[attr]/delete/ references/usage/all privileges

usage: 允许自定义类型

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

允许继续grant权限

**Revoking Authorization in SQL**

revoke <privilege list> on <relation name or view name> from <user list> [restrict|cascade]

With restrict, the revoke command fails if cascading revokes are required

**<privilege-list>** may be all to revoke all privileges.

If **<revokee-list>** includes public, all users lose the privilege **except** those granted it *explicitly*.

If the same privilege was granted twice to the same user by **different** grantees, the user may **retain** the privilege.

revoke grant option for select on *branch* from *U1*

只收回select的grant option.

**Ch6. DB Design and the E-R Model**

**Design Phases**

Spec. of data requirements

Conceptual design

Spec. of functional requirements

Logical Design

Physical Design

防止信息冗余和不完整

**The Entity-Relationship Model**

**Entity Sets**

A *database* can be modeled as

a collection of ***entities****(objects实体)*,

***Relationship****(联系)* among entities.

An ***entity set(实体集)*** is a set of entities of the same type that share the same properties. (用属性描述)

**Attributes**

简单属性/复合属性|单值属性/多值属性|派生属性

**Relationship Sets**

A ***relationship***(联系) is an association among several entities

A ***relationship*** ***set***(联系集) is a mathematical relation among *n* ≥ 2 entities, each taken from entity sets

Mike - lives(relationship) - Addr

！A rel. instance in a given relationship set must be uniquely identifiable from its participating entities！

！There can be more than one relationship set involving the same entity sets！

**Degree:** Refers to number of entity sets that participate in a relationship set. Mostly binary.

**Constraints**

**Mapping Cardinalities(影射数)**

For binary relationships:

One to one / one to many

Many to one / many to many

**Keys**

Superkeys/candidatekeys/primarykeys

**Entity-Relationship Diagram (E-R)**

**See diagrams later for details.**

**directed line (→),** signifying “one,” undirected line (—), signifying “many,”

**Roles** are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles. Role labels are optional, and are used to clarify semantics of the relationship

**Participation of an Entity Set**

Total participation (double line):

实体集的每个实体都有关系集的一个关系

Partial participation:

可以不包含任何关系

**三元关系E-R图**

三元关系及以上的，禁止从关系中引出1个以上箭头(多对1)

**Design Issues**

Common mistakes:

**1)** to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship

**2)** to designate the primary key attributes of the related entity sets as attributes of the relationship set

**Weak Entity Sets (弱实体集)ppt439**

\* Entity set do not have a primary key

\* The existence of a weak entity set depends on the existence of a *identifying or owner entity set(标识/属主实体集)*

\* It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set

\* The *discriminator(分辨符) (or partial key部分码)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set that depend on one particular strong entity.

*primarykey=primarykey0+discriminator*

graph: double rect/diamond & dashed underlines

**Properties**

\* The identifying relationship set should have no descriptive attributes

\* A weak entity set can participate in relationships other than the identifying relationship

\* A weak entity set may participate as owner in an identifying relationship with another weak entity set

\* May express a weak entity set as a multivalued composite attribute of the owner entity set

\* A weak entity set may be more appropriately modeled as an attribute if it participates in only the identifying relationship, and if it has few attributes

**Extended E-R Features**

**Specialization(特殊化)/Top-down**

IS A – C++中单继承。

**Generalization/Bottom-up**

同特殊化互逆,表示法一致可互换

**Constraints on Specialization/Generalization**

Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.

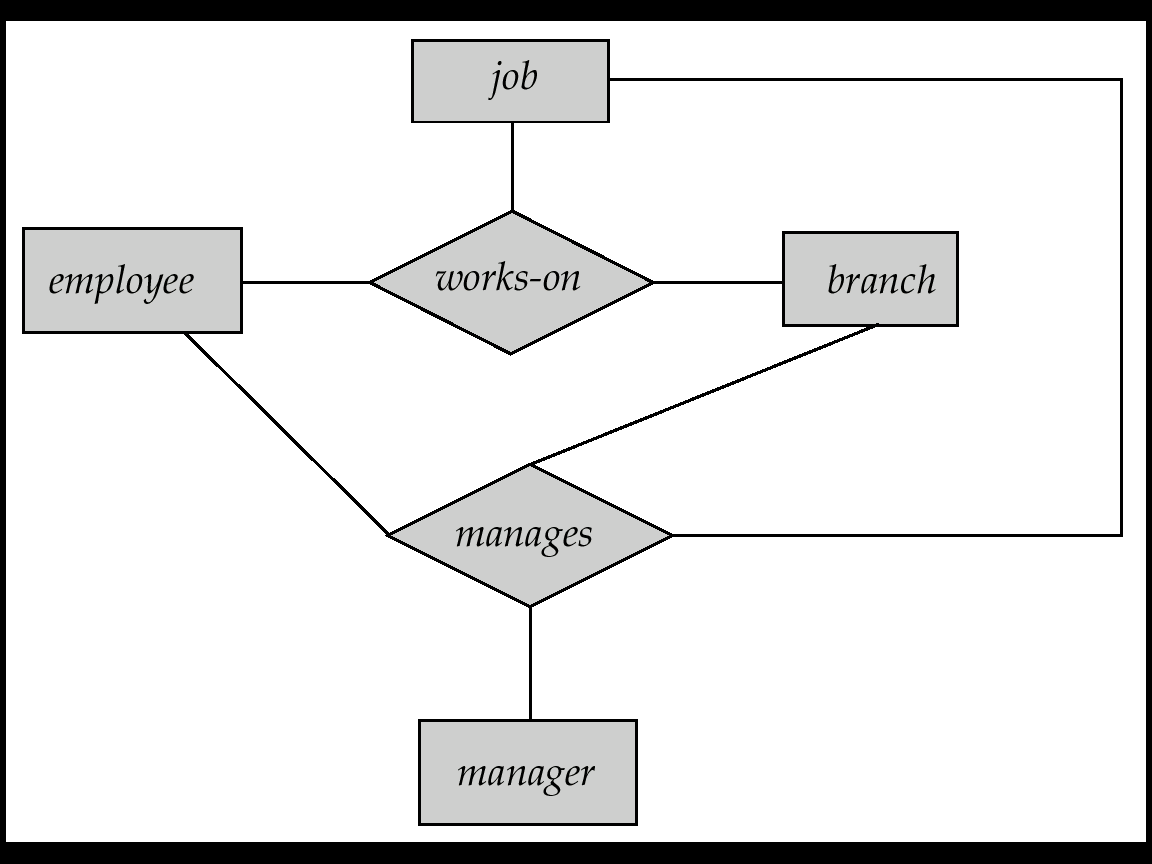
*Disjoint* (by writing *disjoint* beside ISA)

*Overlapped*, default(more than one).

Completeness constraint(完备性约束) -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization**.**

Total(double line)/partial(default)

**Aggregation(聚集)**

去除冗余

Treat relationship as an abstract entity Allows relationships between relationships

Abstraction of relationship into new entity

**Design of an E-R Database Schema**

**E-R Design Decisions**

\* The use of an attribute or entity set to represent an object.

\* Whether a real-world concept is best expressed by an entity set or a relationship set.

\* The use of a ternary relationship versus a pair of binary relationships.

\* The use of a strong or weak entity set

\* The use of specialization/ generalization – contributes to modularity in the design.

\* The use of aggregation – groups a part of an E-R diagram into a single entity set, can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

**(See last of card for graphs)**

**Reduction of an E-R Schema to Tables**

\* A strong entity set reduces to a table with the same attributes

\* Composite attributes are flattened(变平) out by creating a separate attribute for each component attribute

**\*** A multivalued attribute M of an entity E is represented by a separate table EM

\* A weak entity set becomes a table that includes columns for the primary key of the identifying strong entity set

\* A relationship set is represented as a table with columns for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.

**Redundancy of Tables**

**\* Many-to-one** and **one-to-many** relationship sets that are **total** on the **many-side** can be represented by adding an extra attribute to the many side, containing the primary key of the one side to eliminate redundancy.

\* For one-to-one relationship sets, either side can be chosen to act as the “many” side

\* If participation is *partial* on the many side, we can also combine tables by using null values

\* The table corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.

**Representing Specialization as Tables**

M1: Form a table for the higher level entity and a table for each lower level entity set, include primary key of higher level entity set and local attributes.

*More tables*

M2: Form a table for each entity set with all local and inherited attributes

*More redundant data*

*If spec. is total, table for generalized entity (person) isn’t required*

*If the generalization is disjoint and total, the method is appropriate*

**Relations Corresponding to Aggregation**

To represent aggregation, create a table containing

\* primary key of the aggregated relationship,

\* the primary key of the associated entity set

\* Any descriptive attributes

**Ch7. Relational Database Design**

**Design Pitfalls ppt7.3**

**Atomic Domains and First Normal Form**

Domain is atomic if its elements are considered to be indivisible units

**A relational schema R is in first normal form(第一范式) if the domains of all attributes of R are atomic**

*Atomicity is treated diff. in diff. cases.*

**Decomposition Using Functional Dependencies**

**Functional Dependencies**

有属性(集) X, Y,以及tuples t1, t2:

*t*1[X] = *t*2 [X] ⇒ *t*1[*Y* ] = *t*2 [*Y* ]

则称Y函数依赖X记X->Y

\* A functional dependency is a generalization of the notion of a *superkey.*

\* Functional dependencies allow us to express constraints that cannot be expressed using superkeys.

\* A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.

\* Afunctional dependency is ***trivial*** if it is satisfied by all instances of a relation

**In general, α → *β* is trivial if *β* ⊆ α**

**General Faults in decomposing 7.21**

**Lossless-join Decomposition**

**(无损连接分解)**

A decomposition {*R*1*, R*2*, ..., Rn*} of R is a LJD if, for all relations r on schema R that are legal,  
r= ∏R1 (*r*) ∏R2 (*r*) …

R = *R*1∪*R*2∪…∪*Rn*

简言之，分解再自然连接后不变

A decomp of R into *R*1 and *R*2 is lossless join if and only if at least one of the following dependencies is in F+:

***R*1 ∩ *R*2 → *R*1**

***R*1 ∩ *R*2 → *R*2**

**Boyce-Codd Normal Form(BC范式)**

A relation schema *R* is in ***BCNF*** with respect to a set *F* of functional dependencies if for all functional dependencies in *F*+ of the form α → *β,* where α ⊆ *R* and *β* ⊆ *R*,at least one of the following holds:

α → *β* is trivial (i.e., *β* ⊆ α)

α is a superkey for *R*

BCNF要求所有可能α *β*都满足

**Testing for BCNF**

可在F下判别R是否违反BCNF, 但必须在F+下判别R的分解式是否违反BCNF.

**BCNF Decomposition (e.g. ppt7.27)**

Suppose we have a schema *R* and a non-trivial dependency α →*β* causes a violation of BCNF. We decompose *R* into:

* + **(α U β)**
  + **( *R*** – **( *β*** –**α ) )**

**Dependency Preservation**

If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is ***dependency preservin****g.*

*It is not always possible to get a BCNF decomposition that is dependency preserving*

**BCNF一定无损，不一定DP**

**Third Normal Form: Motivation**

There are some situations where

BCNF is not dependency preserving,

efficient checking for FD violation on updates is important

**Solution: define a weaker normal form, called Third Normal Form.**

\* Allows some redundancy (with resultant problems; we will see examples later)

\* But FDs can be checked on individual relations without computing a join.

\* There is ***always*** a lossless-join, dependency-preserving decomposition into 3NF.

**Def. of Third Normal Form**

A relation schema *R* is in third normal form (3NF) if for all α → *β* in *F*+, at least one of the following holds:

\* α → *β* is trivial (i.e., *β* ∈ α)

\* α is a superkey for *R*

\* Each attribute *A* in *β* – α is contained in a candidate key for *R.*

*(NOTE:* each attribute may be in a different candidate key)

\* If a relation is in BCNF it is in 3NF.

\* Third condition is a minimal relaxation of BCNF to ensure dependency preservation

**Testing for 3NF**

Optimization: Need to check only FDs in F, need not check all FDs in F+.

\* Use attribute closure to check for each dependency α → β, if α is a superkey.

\* If α is not a superkey, we have to verify if each attribute in β – α is contained in a candidate key of *R*

**3NF Decomposition Algorithm**

Let Fc be a canonical cover for F;

i := 0;

for each functional dependency α  β in Fc do

if none of the schemas Rj, 1  j  i contains α β then begin

i := i + 1;

Ri := (α β)

end

if none of the schemas Rj, 1  j  i contains a candidate key for R then begin

i := i + 1;

Ri := any candidate key for R;

end

return (R1, R2, ..., Ri)

**Comparison of BCNF and 3NF**

A schema that is in 3NF but not in BCNF has the problems

\* repetition of information (e.g., the relationship banker-branch)

\* need to use null (e.g., to represent the relationship banker-branch where there is no corresponding value for customer).

**Design Goals**

**Goals:** BCNF. Lossless join. Dependency preservation.

**If cannot achieve this, accept one of**

\* Lack of dependency preservation

\* Redundancy due to use of 3NF

*SQL does not provide a direct way of specifying functional dependencies other than superkeys.*

*Can specify FDs using assertions, but they are expensive to test*

**Testing for FDs Across Relations** **(ppt7.37)**

**Relationship of NFs**

5NF ⊆4NF ⊆BC ⊆3NF ⊆2NF ⊆1NF

**Functional Dependency Theory**

**Closure of Functional Dependencies**

The set of all functional dependencies logically implied by *F* is the *closure(闭包)* of *F*. We denote the *closure* of *F* by F*+.*

We can find all ofF*+* by applying **Armstrong’s Axioms(公理)**

if *β* ⊆ α, then α → *β*

(reflexivity自反律)

if α → *β,* then γ α → γ *β*

(augmentation增广律)

if α → *β,* and *β* → γ, then α → γ

(transitivity传递律)

**公理产生的F+正确且完备**

We can further simplify manual computation of *F*+ by using the following additional rules.

\* If α → *β* holds *a*nd α→ γ holds, then α → *β* γ holds

(union rule 合并规则)  
\* If α → *β* γ holds, then α → *β* holds and α→ γ holds (decomposition分解规则)

\* If α → *β* holds *a*nd γ *β* → δ holds, then α γ → δ holds (pseudo transitivity伪传递规则)

**Closure of Attribute Sets**

**Given a set of attributes a, define the *closure* of a under *F* (denoted by a+) as the set of attributes that are functionally determined by a under *F.***

a → βis in *F*+ ⬄ β ⊆ a+

**Uses of Attribute Closure**

Testing for superkey:

To test if α is a superkey, we compute α+, and check if α+ contains all attributes of *R*.

Testing functional dependencies

To check if a functional dependency α → β holds (or, in other words, is in *F*+), just check if β ⊆ α+.

Is a simple and cheap test, and very useful

Computing closure of F

For each γ ⊆ *R,* we find the closure γ+, and for each *S* ⊆ γ+, we output a functional dependency γ → *S.*

**Canonical Cover** *(正则覆盖)*

A canonical cover for F is a set of dependencies Fc such that

*\* F* logically implies all dependencies in *Fc,* and

*\* Fc* logically implies all dependencies in *F,* and

\* No functional dependency in *Fc* contains an extraneous attribute, and

\* Each left side of functional dependency in *Fc* is unique.

Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F

*\* it does not contain extraneous attributes,* having no redundant dependencies or redundant parts of dependencies

*\* it combines functional dependencies with the same left side. It is cheaper to test Fc than it is to test F itself*

**Extraneous Attributes**

\* Attribute A is **extraneous** in α if *A* ∈ α and *F* logically implies (*F* – {α → β}) ∪ {(α – *A*) → β}.

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

***Note****:* implication in the opposite direction is trivial in each of the cases, since a “stronger” FD always implies a weaker one

**Testing if an Attribute is Extraneous**

if attribute A ∈ α is extraneous in α

**1** compute ({α} – A)+ using the dependencies in *F*

**2** check that ({α} – A)+ contains A; if it does, *A* is extraneous

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

**Computing Canonical Cover**

*R* = (*A, B, C), F = {A* → *BC, B* → *C, A* → *B, AB* → *C*}

**1** Combine *A* → *BC* and *A* → *B* into *A* → *BC*

Set is now *{A* → *BC, B* → *C, AB* → *C*}

**2** *A* is extraneous in *AB* → *C*

\* Check if the result of deleting A from *AB* → *C* is implied by the other dependencies

Yes: in fact, *B* → *C* is already present!

\* Set is now *{A* → *BC, B* → *C*}

**3** *C* is extraneous in *A* → *BC*

\* Check if *A* → *C* is logically implied by *A* → *B* and the other dependencies

Yes*:* using transitivity on *A* → *B and B* → C.

**4** The canonical cover is: *{* *A* → *B, B* → *C }*

**Decomposition Using Functional Dependency**

Decide whether a particular relation *R* is in “good” form.

In the case that a relation *R* is not in “good” form, decompose it into a set of relations {*R*1*, R*2*, ..., Rn*} such that

\* each relation is in good form: each *R*ishould be in either Boyce-Codd Normal Form or Third Normal Form

\* Lossless-join decomposition: Otherwise would result in information loss

\* Dependency preservation(保持依赖): Let *Fi* be the set of dependencies *F+* that include only attributes in *Ri.*

\* Preferably the decomposition should be dependency preserving, that is, (*F*1 *∪ F*2 *∪ … ∪ F*n)+ = *F+*

\* Otherwise, checking for violation of functional dependencies may require join, which is expensive.

**Testing for dependency preservation**

To check if a dependency αβ is preserved in a decomposition of R into R1, R2, …, Rn, we apply the following simplified test:

result = α  
while (changes to result ) do

for each Ri in the decomposition

t = (result  Ri )+  Ri

result = result  t

If result contains all attributes in β, then the functional dependency αβ is preserved.

Apply the test on all dependencies in F to check if a decomposition is dependency preserving.

**Decomposition Using Multivalued Dependencies**

**Multivalued Dependencies**

Let *R* be a relation schema and let α ⊆ *R* and β ⊆ *R.* The *multivalued dependency* **α →→ β** holds on *R* if in any legal relation *r(R),* for all pairs for tuples *t*1 and *t2* in *r* such that *t*1[α] = *t2* [α], there exist tuples *t3* and *t*4 in *r* such that:

*t*1[α] = *t2* [α] = *t*3 [α] = *t*4[α]

*t*3[β] = *t*1 [β]

*t*3[*R –* β] = *t*2[*R –* β]

*t*4 [β] = *t*2[β]

*t*4[*R –* β] = *t*1[*R –* β]

If β  α, or α  β = R, then α  β is trivial.

If α  β, then α  β

**Fourth Normal Form (4NF)**

A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D+ of the form α  β, where α  R and β  R, at least one of the following hold:

α  β is trivial (i.e., β  α or α  β = R)

α is a superkey for schema R

If a relation is in 4NF, it is in BCNF.

**4NF Decomposition Algorithm**

Like BCNF decomposition.

**Chapter 8: Application Design and Development**

**Triggers**

*A trigger(触发器) is a statement that is executed automatically by the system as a side effect of a modification to the database.*

**Example**

create trigger *overdraft-trigger* after update on *account*referencing new row as *nrow* for each row  
when *nrow.balance* < 0  
begin atomic

(update actions)

end

**Triggering Events and Actions in SQL**

\* Triggering event can be insert, delete or update

\* Triggers on update can be restricted to specific attributes

create trigger *overdraft-trigger* after update of *balance* on *account*

\* Values of attributes before and after an update can be referenced. (transition variables)

referencing old row as

for deletes and updates

referencing new row as

for inserts and updates

**Triggers Before an Event**

create trigger *setnull-trigger* before update on *r*

referencing new row as *nrow*

for each row

when *nrow.phone-number = ‘ ‘*

set *nrow.phone-number* = null

**Statement Level Triggers**

Use for each statement instead of for each row

Use referencing old table or referencing new table to refer to temporary tables (called *transition tables*) containing the affected rows

Can be more efficient when dealing with SQL statements that update a large number of rows

**External World Actions**

\* Triggers can be used to record actions-to-be-taken in a separate table

\* Have an external process that repeatedly scans the table, carries out external-world actions and deletes action from table

**When Not To Use Triggers**

\* Triggers should be written with great care, since a trigger error detected at run time causes the failure of the insert/delete/update statement that set off the trigger

\* Can even lead to an infinite chain of triggering

**Chapter 10 XML**

**Introduction**

XML: **Extensible Markup Language**

**Comparison with Relational Data**

*Inefficient*: tags, which in effect represent schema information, are repeated

Better than relational tuples as a data-exchange format

\* not like relational tuples, XML data is self-documenting due to presence of tags

\* Non-rigid format: tags can be added

\* Allows nested structures

\* Wide acceptance, not only in database systems, but also in browsers, tools, and applications

**Structure of XML Data**

**Basic Structure**

\* Mixture of text with sub-elements is legal in XML. Useful for document markup, but discouraged for data representation

**Attributes**

<account acct-type = “checking” monthly-fee=“5”>

**Attributes Vs. Subelements**

\* In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents

\* In the context of data representation, the difference is unclear and may be confusing

\* **Suggestion:** use attributes for identifiers of elements, and use subelements for contents

**More on XML Syntax**

\* Elements without subelements

<account number=“A-101” balance=“200” />

\* To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below.

<![CDATA[<account> … </account>]]>

**Namespaces**

unique-name:element-name

Avoid using long unique names all over document by using XML Namespaces

<bank Xmlns:FB=“http://www.FB.com”>

<FB:branch> …

**XML Document Schema**

**DTD**: Widely used

**XML Schema**: Newer, increasing use

**Document Type Definition (DTD)**

**DTD constraints:** What elements can occur/ What attributes can/must an element have/ What subelements can/must occur inside each element, and how many times.

**DTD does not constrain:** data types. All values represented as strings in XML

**DTD syntax:**

<!DOCTYPE bank [

<!ELEMENT element (subelements-specification) >

<!ATTLIST element (attributes) >

]>

**Element Specification in DTD**

**Subelements can be specified as**

\* names of elements

\* **#PCDATA** (parsed character data), i.e., character strings

\* **EMPTY** (no subelements) or **ANY** (anything can be a subelement)

Subelement specification may have regular expressions

<!ELEMENT bank (( account | depositor)+)>

<!ELEMENT message (to+ from, cc\*, sub, text)>

<!ELEMENT to (#PCDATA )>

“|” - alternatives, “+” - 1 or more occurrences

“\*” - 0 or more occurrences, “?” - 0 or 1 occurrences

**Attribute Specification in DTD**

For each attribute:

<!ATTLIST name1 type1 default1|#REQUIRED|#IMPLIED… >

Type of attribute

**CDATA**

**ID** (identifier) or **IDREF** (ID reference) or **IDREFS** (multiple IDREFs)

Whether

mandatory ( #**REQUIRED** )

has a default value (value),

or neither ( #**IMPLIED** )

<!ATTLIST account acct-type CDATA “checking”>

<!ATTLIST customer customer-id ID #REQUIRED accounts IDREFS #REQUIRED>

**IDs and IDREFs**

\* An element can have at most one attribute of type ID

\* The ID attribute value of each element in an XML document must be distinct

\* An attribute of type IDREF must contain the ID value of an element in the same document

\* An attribute of type IDREFS contains a set of (0 or more) ID values…

**Limitations of DTDs**

\* All values are strings, no integers, reals, etc.

\* Difficult to specify unordered sets of subelements

\* IDs and IDREFs are untyped

**XML Schema**

\* XML Schema addresses the faults of DTDs.

\* XML Schema is itself specified in XML syntax

\* XML Schema is integrated with namespaces

\* XML Sche is more complicated than DTDs

<xs:schema xmlns:xs=… >

<xs:element name=“bank” type=“BankType”/>

<xs:element name=“account”>  
<xs:complexType> <xs:sequence>  
 <xs:element name=“” type=“xs:string”/> …  
</xs:squence> </xs:complexType> </xs:element>

….. definitions of customer and depositor ….

<xs:complexType name=“BankType”>  
</xs:complexType> </xs:schema>

<xs:keyref name = “AK” refer=“aKey”>

<xs:selector xpath = “/bank/account”/>

<xs:field xpath = “account\_number”/>

</xs:keyref>

**Querying and Transformation**

**Standard XML querying/translation lang**

XPath / XSLT / XQuery

**Tree Model of XML Data**

\* Element nodes have children nodes, which can be attributes or subelements

\* Text in an element is modeled as a text node child of the element

\* The root node has a single child, which is the root element of the document

**XPath**

Result of path expression: set of values that along with their containing elements/attributes match the specified path.

\* Path( ) returns without the enclosing tags

\* Selection predicates may follow any step in a path, in [ ]

/bank-1/customer/account[balance>400] /bank-1/customer/account[balance]

returns account containing a balance subele

Attributes are accessed using “@”

/bank-1/cust/acc [bal>400]/@account-number

**Functions in XPath**

count(), and, or, not(),

IDREFs can be referenced using function id()

**More XPath Features**

\* Operator “|” used to implement union, cannot be nested inside other operators

“//” in path can be used to skip levels of nodes

“//” alone, specifying “all descendants” (of last step)

“..” alone specifies the parent(of last step)

doc(name) returns the root of a named doc

**XQuery**

XQuery uses a **for…let…where…order by…result** syntax

**for**: SQL from

**let:** allows temporary variables

**where**: SQL where

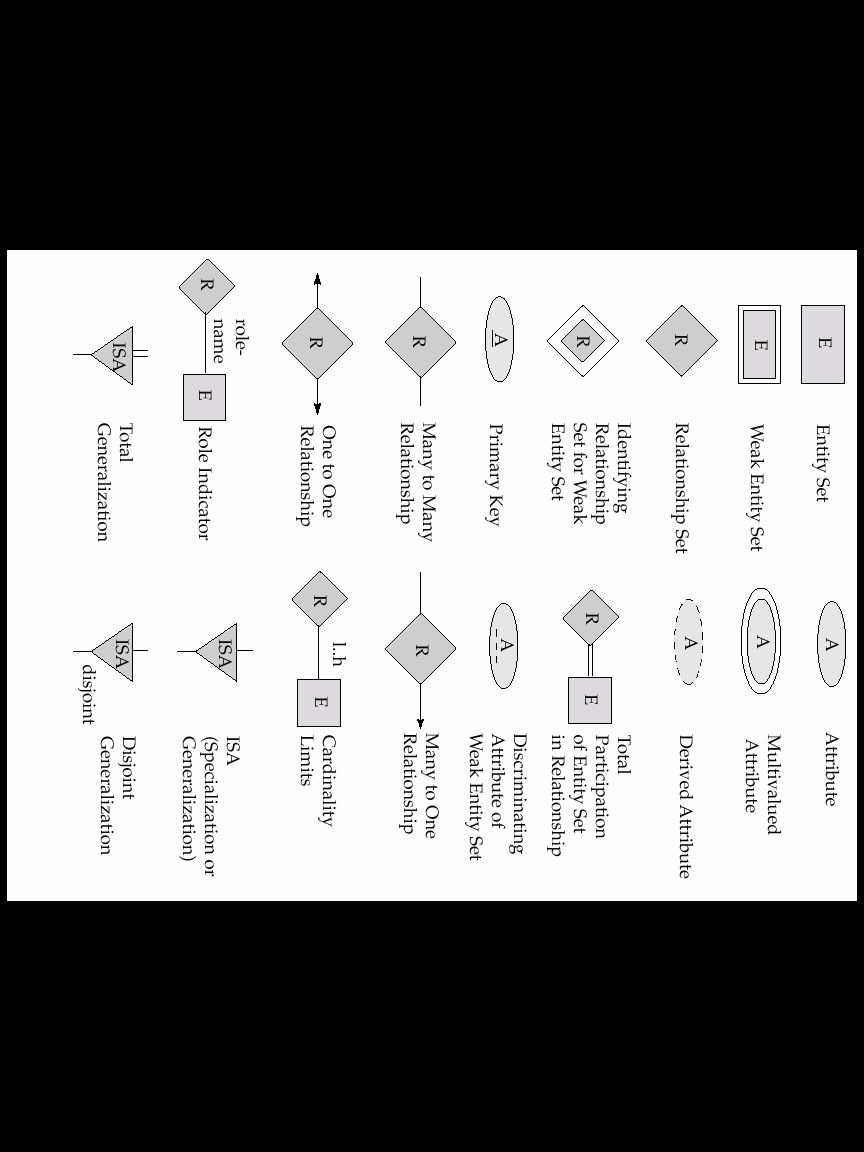
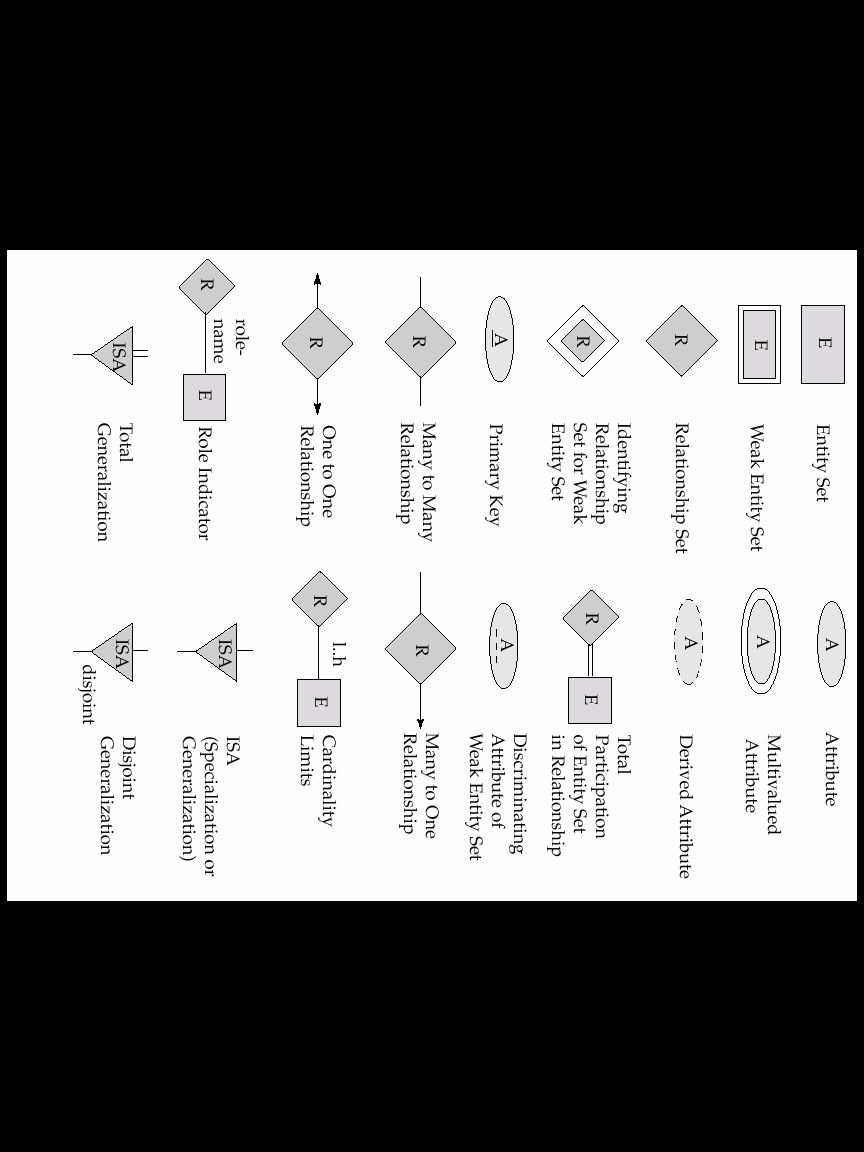
**order by**: SQL order by

**result**: SQL select

**for** $x in /bank-2/account  
**let** $acctno := $x/@account-number   
**where** $x/balance > 400   
**return** <account-number> $acctno </account-number>

**Joins & Nested Queries…**

**Storage of XML Data**

**E-R Graphs:**