Subject: اصول طراحی پایگاه داده ها
Year. 86 Month. 8 Date. 24 (161)

جلسه هشتم 86.8.24

Modification of the Database:

تغییر Data Base ؛ یعنی چگونگی (How) اضافه کردن، حذف کردن و یا تغییر اطلاعات در دیتابیس. دراین بحث تغییری روی ساختمان D.B نخواهیم داشت

← برای تغییر schema از projection استفاده می شود و

← برای تغییر tuple ها از سه عمل زیر استفاده می شود:

1, Deletion 2, Insertion 3, Updating

۱, حذف (Deletion): برای حذف یک tuple ابتدا آنها را انتخاب کرده (E) از کل کم می کنیم و حاصل را در همان رابطه می ریزیم ←

$r \leftarrow r - E$

توجه داشته باشید که نمی توانیم تنها مقادیر att های آن را پاک کنیم.

مثال: تمام حسابهای smith را پاک کنید.

$$depositor \leftarrow depositor - \sigma_{customer\text{-}name = "smith"}(depositor)$$

مثال: وام هایی با مقادیر بین ۰ تا 50 را پاک کنید.

$$loan \leftarrow loan - \sigma_{amount \geq 0 \wedge amount \leq 50}(loan)$$

مثال: همه حسابهای شعبه Brooklyn را پاک کنید.

$$r_1 \leftarrow \prod_{branch\text{-}name,\ account\#,\ balance}\left(\sigma_{branch\text{-}city = "Brooklyn"}(account \bowtie branch)\right)$$

$$account \leftarrow account - r_1$$

۲, درج (Insertion): عمل درج را با عملگر Union اجتماع انجام می دهیم، باید توجه داشته باشیم که r, E حتماً schema هم را داشته باشند.

$r \leftarrow r \cup E$

P4PCO

Subject:
Year. Month. Date. ( )

مثال: یک حساب برای smith با شماره حساب A-973 با 1200 موجودی در شعبه perry اضافه کنید

account ← account ∪ {(A-973, "perry", 1200)}
depositor ← depositor ∪ {("smith", A-973)}

مثال: برای تمام کسانی که در شعبه میرداماد وام گرفته‌اند به عنوان هدیه حساب باز کنید و درون حساب 200$ باشد (فرض: loan# همان account# حساب جدید باشد)
اول کسانی که وام گرفته‌اند را پیدا می‌کنیم (برای k اضافه کردن حساب باید account تغییر کند)

$r_1 \leftarrow \sigma_{branch\text{-}name = "Mirdamad"}(borrower \bowtie loan)$ ← کسانی که از میرداماد وام گرفته‌اند

account ← account ∪ ($\Pi_{loan\#, branch\text{-}name}(r_1) \times \{(200)\}$) ← ستون balance را با ضرب اضافه می‌کنیم

ترتیبشان هم مهم است باید قبل ترتیب attهای account باشد

depositor ← depositor ∪ $\Pi_{customer\_name, loan\#}(r_1)$

3. بهنگام‌سازی (updating): ممکن بخواهیم مقادیر یک tuple را بدون تغییر همهٔ مقادیر tuple ها تغییر دهیم این کار را با عملگر Generalized-projection می‌توان انجام داد.

$r \leftarrow \Pi^{(r)}_{F_1, F_2, \ldots, F_n}$

مثال: موجودی تمام مشتریها را 5٪ افزایش دهید

account ← $\Pi^{(account)}_{account\#, branch\_name, balance * 1.05}$

P4PCO

مثال: موجودی کلیه ی که در شعبه میرداماد حساب دارند را 5٪ افزایش دهید.

ابتدا کلیه ی که در میرداماد حساب دارند را از کل کم می کنیم و سپس اجتماع این مجموعه را با مجموعه حسابهای شعبه میرداماد با افزایش 5٪ موجودی در account می زنیم:

$$account \leftarrow \left[account - \sigma_{branch\_name = "Mirdamad"}(account) \right] \cup \prod_{account\#,\ branch\_name,\ balance * 1.05} \left(\sigma_{branch\_name = "Mirdamad"}(account) \right)$$

فصل (5)

Query lang. ← procedural → عبارتهای (How)
Non-procedural → DRC, TRC (what)

نتیجه هر query یک رابطه است؛ و هر رابطه مجموعه‌ای از تاپل هاست بنابراین
(جواب هر query مجموعه‌ای از tuple هاست)

DRC (Domain Relational Calculus)
TRC (Tuple Relational Calculus)

حال به بررسی DRC می پردازیم:

فرمت DRC: $query_{جواب} = \{ \langle x_1, x_2, \ldots, x_n \rangle \mid \underbrace{P(x_1, x_2, \ldots, x_n)}_{\text{شرط}} \}$ (دامنه)

مثال: شماره وام، نام شعبه و میزان وام را برای وام های بیش از 1200 به دست آورید

از آنجا که query سه چیز را خواسته بنابراین DRC ما سه تاپل خواهد بود که آنها را a: amount ، b: branch_name ، L: loan# می نامیم

$$DRC: \{ \langle L, b, a \rangle \mid \langle L, b, a \rangle \in loan \wedge a > 1200 \}$$

P4PCO

Subject:
Year. Month. Date. ( )

مثال: تمام شماره وام هایی که مقدارشان بزرگتر از 1200 باشد را پیدا کنید.
پاسخ تک تاپل رابطه:

$$\{ \langle L \rangle \mid \exists b,a (\langle L,b,a \rangle \in loan \wedge a > 1200) \}$$

نکته: بعد از هر $\exists$ و $\forall$ باید یک پرانتز باز وجود داشته باشد که توضیحات لازم درون آن پرانتز داده شود. و متغیرهای قبل از هر پرانتز تا قبل از بسته شدن پرانتز معنی دارند.

مثال: نام مشتریهایی که از شعبه میرداماد وام گرفته اند و همچنین مقدار وامشان را بیابید.
c : customer_name → borrower , a : amount → loan

$$\{ \langle c,a \rangle \mid \exists l (\langle c,l \rangle \in borrower \wedge \exists b (\langle L,b,a \rangle \in loan \wedge b = \text{"Mirdamad"})) \}$$

همانطور که می بینیم l در هر دو شرط استفاده شده اما نمی تواند یک بار تعریف شده و دلیلی ندارد بار دوم پرانتز قرمز است؛ یعنی چنانچه پرانتز قرمز را قبل از تعریف $\exists$ می بستیم باید دوباره l را تعریف می کردیم که البته این l با l قبلی تفاوت دارد و پاسخ سوال غلط خواهد بود.

l_1 → balance
a → account #
b → branch_name

مثال: شماره حساب شخصی که Max موجودی را دارد بیابید.

$$\{ \langle a \rangle \mid \exists b, l_1 (\langle a,b,l_1 \rangle \in account \wedge \forall x,y,l_2 (\langle x,y,l_2 \rangle \in account \wedge L_1 \geq L_2)) \}$$
$$\equiv L_1 > L_2 \wedge a \neq x$$

مثال: نام مشتریهایی که حساب دارند یا وام دارند و یا هردو ؟

$$\{ \langle c \rangle \mid \exists a (\langle c,a \rangle \in depositor) \vee \exists l (\langle c,l \rangle \in borrower) \}$$

مثال: نام مشتریهایی که در شعبه میرداماد حساب دارند و یا از میرداماد وام گرفته اند؟

P4PCO

$\{\langle c\rangle \mid \exists l(\langle c,l\rangle \in borrower \wedge \exists b,a(\langle l,b,a\rangle \in loan \wedge b = \text{"Mirdamad"}) \vee \exists a(\langle c,a\rangle \in depositor \wedge \exists b,l(\langle a,b,l\rangle \in account \wedge b = \text{"Mirdamad"})\}$

مثال: نام تمام مشتریهایی که در تمام شعب تهران حساب دارند؟ (÷)

$\{\langle c\rangle \exists s,t(\langle c,s,t\rangle \in customer) \wedge \forall x,y,z(\langle x,y,z\rangle \in branch \wedge y = \text{"Tehran"} \Rightarrow \exists a,l(\langle a,x,l\rangle \in account \wedge \langle c,a\rangle \in depositor))\}$

جلسه نهم 86.8.26

SQL (structure Query Language):

تعریف شمای رابطه، حذف رابطه، اصلاح شمای رابطه

دستورات SQL تشکیل شده از:

- Data Definition Language : DDL
- Data Manipulation Language : DML (Edit) دستکاری — درج، حذف، اصلاح جزئی

انواع داده ها (Data type):

- char(n) : کاراکتر یا رشته با طول ثابت n.
- varchar(n) : کاراکتر با طول متغیر و با حداکثر طول n.
- numeric(p,d) : عدد اعشاری به طول p (یعنی تعداد کل ارقام = p) و تعداد ارقام بعد از ممیز اعشار = d. مثال □□.□□□ ⇒ (5,3)

دستورات DDL: مربوط به Schema و Relation هستند یعنی می توانیم با دستورات DDL یک رابطه جدید (Relation) ایجاد کنیم و یا Schema یک رابطه را تغییر دهیم (یعنی ستون اضافه و کم کنیم) اما با خود Data کاری نداریم

دستورات DML: به کمک دستورات DML می توانیم رکوردی را به یک رابطه اضافه کرده و یا رکوردی را از یک schema حذف کنیم. (یعنی سطر اضافه و کم می کنیم)

P4PCO

Subject:
Year. Month. Date. ( )

دستور ایجاد یک رابطه:

Creat table r ($A_1 D_1$, ... , $A_n D_n$,
<integrity Constraint$_1$> ,
<integrity Constraint$_k$>)

رابطه، att، دامنه، محدودیت

A_i : att.
D_i : Domain

توجه: محدودیت ها همیشه نمی آیند؛ مثلاً جایی که می خواهیم primary key را نشان دهیم باید از Constraint استفاده کنیم.

مثال: فرض کنید می خواهیم رابطه account را بسازیم:

account = (account-number, branch-name, balance)

```
Creat table account (
        account-number    char(5) ,
        branch-name       varchar(20) ,
        balanc            numeric(12,2),
*محدودیت ← PRIMARY KEY   (account-number)
        );
```

محدودیت * نمی گذارد که account # مقدار نداشته باشد (یعنی NULL باشد) و همواره باید مقدار واحد (unic) را بپذیرد.

مثال:

depositor (customer-name, account #)

```
Creat table depositer (
    customer-name    varchar(20),
    account-#        numeric(12,2),
  PRIMARY KEY        (customer-name, account #),
  FOREIGN KEY        (customer-name) REFERENCES customer,
  FOREIGN KEY        (account #) REFERENCES account
    );
```

Subject: اصول طراحی پایگاه داده ها
Year. 86 Month. 8 Date. 26 19

در اینجا چون دو primary key هستند دو ویژگی (1) اجازه نمی دهد که هیچ کدام NULL شوند و همچنین اجازه تکرار به آنها نمی دهد یعنی نباید هر دو با هم تکرار شوند)

← کلمه کلیدی REFERENCES به primary key مراجعه می کند

دراین جا شناسه customer-name چون نمی تواند بی معنی باشد بنابراین باید بگوییم customer.name فقط می تواند از جدول customer باشد و این کار را با کلید کلیدی references انجام می دهیم.
(زیرا هر دو کلید خارجی هستند)

نکته: هرگاه در schema ای که تعریف می کنیم کلید خارجی داشته باشیم باید از الگوی زیر استفاده کنیم:

FOREIGN KEY (نام کلید خارجی) REFERENCES (نام رابطه ای که شامل کلید خارجی است)

* دستور INSERT ← (DML) اضافه کردن یک رکورد به schema

INSERT INTO ... VALUES (..., ..., ...)

مثال: INSERT INTO customer VALUES ("AFSANEH", "Ghobadian", "Tehran")
name, street, city

باید تغییر به ترتیب مقادیر دهی کنیم.

* دستور DROP TABLE ← (DDL) دستوری است که یک table را به کلی پاک می کند یعنی یک schema را به کلی حذف می کند.

DROP TABLE ...

مثال: DROP TABLE account

* دستور DELET ← (DML) رکوردهای را پاک می کند که در شرط نوشته شده

DELET FROM ... WHERE ...

PqPCO

Subject:
Year. Month. Date. ( )

مثال: DELET FROM account ← کل رکوردهای account پاک می‌کند

مثال: DELET FROM account WHERE balance = 0
← رکوردهایی از رابطه account را حذف می‌کند که موجودی آنها صفر باشد.

* دستور ALTER TABLE ← (DDL) schem: رابطه‌ای که به وجود آورده‌ایم را تغییر می‌دهد (یعنی ستون حذف یا اضافه می‌کند)

ALTER TABLE ... — ADD (اضافه) / DROP (حذف) —

مثال: ALTER TABLE customer ADD tel-number varchar(10)

* دستور SELECT ← (DML)

SELECT $A_1, A_2, \dots A_n$ — A_i = attribute
FROM $r_1, r_2, \dots r_k$ — r_i = relation
[WHERE p] — p = شرط

خروجی‌های دستور select همیشه به صورت Multiset است یعنی تکراری‌ها را حذف نمی‌کند و دستور SELECT DISTINCT تکراری‌ها را حذف می‌کند (یعنی از هر تکراری یکی را می‌نویسیم)

معادل جبر رابطه‌ای دستور select:

$$\Pi_{A_1, A_2, \dots, A_n}\left(\sigma_{p}(r_1 \times r_2 \times \dots \times r_n)\right)$$

مثال: SELECT DISTINCT branch-name
FROM loan

$\Pi_{\text{branch-name}}(\text{loan})$ = SELECT DISTINCT ✓
تکراری‌ها را حذف می‌کند

SELECT branch_name
FROM loan → (تکراری‌ها را حذف نمی‌کند)

P4PCO

مثال:

$\prod_{branch\_name,\ amount * 100}(loan)$

SELECT branch_name, amount * 100 → Gen. porj.
FROM loan

مثال: WHERE CLAUSE : شماره وام گیرنده‌ای که از شعبه میرداماد وام گرفته‌اند و مقدار وامشان بیش‌تر از 1200 است.

SELECT loan#
FROM loan
WHERE branch_name = "Mirdamad" and amount > 1200

معادل جبر رابطه‌ای $\prod_{loan\#}\left(\sigma_{branch\_name = "Mirdamad" \wedge amount > 1200}(loan)\right)$

مثال: مقدار وام بین 10,000 و 90,000 باشد.

WHERE amount ≥ 10000 and amount ≤ 90000
or
WHERE amount between 10000 and 90000
بین دو عدد

مثال: معادل SQL بنویسید:

$\prod_{customer\_name,\ loan\#,\ amount}(borrower \bowtie loan)$

SELECT customer_name, loan#, amount
FROM borrower, loan
WHERE borrower.loan# = loan.loan#

$\prod_{customer\_name,\ loan\#,\ amount}(\sigma_{branch\_name = "Mirdamad"}(borrower \bowtie loan))$

WHERE " " and branch_name = "Mirdamad"

فعل‌های مقایسه‌ای شرط

Subject.
Year. Month. Date. ( )

Rename: می‌توانیم از att در SELECT و از relation ها در بخش FROM با دستور as تغییر دهیم

```
SELECT A1 as A1', A2, —, An
FROM   r1 as r1', r2, —, rn
```

نکته: برای اجتماع، اشتراک کافی است کلمات کلیدی Union (اجتماع) و Intersect (اشتراک) را بین دو مجموعه به کار ببریم

مثال: اسم مشتری‌هایی که وام یا حساب دارند.

```
(SELECT customer-name FROM borrower) Union
(SELECT customer-name FROM depositor)
```

* Union ، Intersect تکرارها را حذف می‌کند (برخلاف select) و چنانچه نخواهیم تکرارها حذف گردند از Union All ، intersect All استفاده می‌کنیم.

مثال: شماره حساب کسی که max مقدار موجودی را دارد بیابید؟

$$\Pi_{account\#}(account) - \Pi_{account.account\#}(\sigma_{account.account\# < x.account\#}(account \times \rho_x\ account))$$

```
(SELECT account#
 FROM   account)
except
(SELECT account.account#
 FROM   account, account as x
 WHERE  account.account# < x.account#)
```

مثال: متوسط موجودی حساب‌های شعبه میردامادی

$$\sigma_{branch\text{-}name = "Mirdamad"}\left(G_{avg(balance)}(account)\right)$$

P4PCO

```
SELECT avg(balance)
FROM account
WHERE branch_name = "Mirdamad"
```

مثال: متوسط موجودی هر شعبه؟

$$\mathcal{G}_{\text{branch\_name}}^{(\text{account})}\ \text{avg(balance) as avgbal}$$

```
SELECT branch_name, avg(balance)
FROM account
GROUP BY branch_name
```

گروه بندی

$\mathcal{G}_{\max(\text{avgbal})}^{(r_1)}$ ← بیشترین موجودی شعبه‌ها

مثال: میانگین موجودی شعبی که از 10000 بیشتر است؟

$$r_1 \leftarrow \mathcal{G}_{\text{branch\_name}}^{(\text{account})}\ \text{avg(balance) as avgbal}$$

$\mathcal{G}_{\text{avgbal}>10000}(r_1)$ ← شرط پایداری avg

اجرا شود

چون شرط بر اساس کار G باید اعمال گردد، نباید بنویسیم where بلکه از having استفاده می‌کنیم

```
SELECT branch_name, avg(balance) as avgbal
FROM account
GROUP BY branch_name
HAVING avgbal > 10000
```

→ SELECT بعد از اینکه کامل اجرا شد این تست را اعمال می‌کند

ORDER BY: اجرای شود SORT کردن یا دستور

```
ORDER BY ___ desc/asc
```

نزولی / صعودی

گاهی از وقتی چند چیز مرتب کردن صورت می‌گیرد این کار را به کمک استفاده از کامادی اجرا می‌کنیم

Subject:
Year. Month. Date. ( )

مثال: موجودی‌ها را براساس میزان و شماره حسابشان از بزرگ به کوچک مرتب کنید (ترتیبی)

```
SELECT account# , balance
FROM account
ORDER BY balance desc , account# desc
```

جلسه دهم 86 . 9 . 3

Null: یا مقدار ندارد / یا مقدارش معلوم نیست

هر مقایسه‌ای که حاوی مقدار تهی (null) باشد نتیجه‌اش ناشناخته (unknown) است

And: T and unknown → unkown
F and unkown → F

Or: T or unknown → T
F or unknown → unknown

نکته: عملگر sum در ورودی خود از مقادیر تهی صرف نظر می‌کند یعنی مقادیر null را نمی‌شمارد
اما عملگر count مقادیر null را هم می‌شمارد. (فقط count ها null ها را نمی‌شمارد)

مثال: {2, 4, null}

sum = 6 | Count = 3 | , avg = $\frac{6}{2}$ = 3 |

Subquery (Query های تودرتو)

query های فرعی، عبارت select-from-where است که داخل query های دیگر قرار دارد. استفاده متداول از query های فرعی، انجام تست‌هایی برای عضویت مجموعه، مقایسه مجموعه‌ها و کاردینالیتی مجموعه‌است.

عضویت مجموعه: (Set Membership)

عملگر "in" عضویت مجموعه را تست می‌کند و عملگر "not in" عدم عضویت را تست می‌کند

مثال: نام مشتریانی که وام گرفته اند و حساب دارند.

```
select distinct customer-name
from borrower
where customer-name in (select customer-name
                        from depositor)
```

* باید تضمین به تضمین باشد یعنی یکی - دوتایی ...

مثال: نام مشتریانی که در یک شعبه خاص هم وام دارند و هم حساب.

```
select distinct customer-name
from borrower, loan
where borrower.loan# = loan.loan# and
(branch-name, customer-name) in
        (select branch-name, customer-name
         from depositor, account
         where depositor.account# = accoun.account#)
```

توجه: چنانچه در این مثال به جای "in" بنویسیم "not in" عکس را برمی گرداند یعنی نام مشتریانی که از یک شعبه وام گرفته اند اما در آنجا حساب ندارند.

معادلسازی مجموعه ها (Set Compration):

مثال: اسامی مشتریانی که دارایی های آنها بیش از حداقل یک شعبه مستقر در تهران است.

```
select branch-name
from branch
where assets > some (select assets
                     from branch
                     where branch-city = "Tehran")
```

> some ← بیش از حداقل یک
در SQL

Subject:
Year. Month. Date. ( )

به عبارت های زیر که SQL مورد استفاده قرار می گیرد توجه کنید:

> some ← «بیشتر از حداقل یک»

<> some ، = some ، >= som ، <= some

↓ in معادل

> all ← «بیشتر از همه»

<> all ، = all ، >= all ، <= all ، < all

↓ not in معادل

جلسه یازدهم 86.9.10

تست رابطه ی خالی: (تست تهی بودن)

ساختار exists در صورت خالی نبودن یک query مقدار True را برمی گرداند یعنی اگر در یک query مقداری وجود داشته باشد exists، true خواهد بود.

مثال: تمام مشتریانی را پیدا کنید که هم وام و هم حساب دارند.

```
Select customer-name
from borrower
where exists ( Select *
                 from depositor
                 where depositor.customer-name =
                       borrower.customer-name )
```

↓ وجود دارد

ساختار not exists ← وجود ندارد، در صورت خالی بودن یک query مقدار True را برمی گرداند یعنی اگر در یک query مقداری وجود نداشته باشد not exists، true خواهد بود.

مثال: تمام مشتریانی که وام دارند ولی حساب ندارند.

← کافی است در حل مثال قبل به جای exists از not exists استفاده کنیم

P4PCO

Subject: اصول طراحی پایگاه داده
Year 86 Month 9 Date 10 23

مثال: تمام مشتریانی را پیدا کنید که در تمام شعب "Brooklyn" حساب دارند.

```
select distinct S.customer-name
from depositor as S
where not exists ((select branch-name
                   from branch
                   where branch_city = "Brooklyn")
                  except
                  (select R.branch_name
                   from depositor as T, account as R
                   where T.account# = R.account# and
                   S.customer-name = T.customer-name))
```

در SQL تقسیم نداریم ⇐

query اول: تمام شعب شهر "Brooklyn" را پیدا می‌کند.

query دوم: (depositor ⋈ account) را حساب کرده و مشتریانی که (در ابتدا تعریف شده بودند) و حساب دارند را پیدا می‌کند.

query 1 − query 2 → مشتریانی که حساب دارند / شعب تهران / مشتریانی که در شعب تهران حساب ندارند

تست: حکم وجود ندارد. (تست Multiset یا Set بودن مجموعه)

ساختار "unique" در صورتی که در query فکری نداشته باشیم مقدار true را برمی‌گرداند. (یعنی unique برای مجموعه‌های set مقدار T را برمی‌گرداند)

مثال: تمام مشتریانی که حداکثر یک حساب در شعبه میرداماد دارند را پیدا کنید.

```
Select T.customer-name
From depositor as T
where unique ( select R.customer-name
               from account, depositor as R
               where R.account# = account.account#
               and T.customer-name = R.customer-name
               and account.branch-name = 'Mirdamad')
```

فکری ندارد ↓

P4PCO

Subject:
Year. Month. Date. ( )

تقاضاهای پیچیده (Complex Query):

روشش برای ترکیب بلوک‌های SQL، جهت بیان تقاضاهای پیچیده وجود دارد که عبارتند از:

۱- رابطه‌های مشتق (Derived Relation)

۲- بخش with

۱- رابطه‌های مشتق: می‌توانیم در بخش FROM یک query از subquery استفاده کنیم و باید برای آن subquery ای انتخاب کنیم و حتی می‌توانیم نام صفاتش را هم تغییر دهیم (با استفاده از as)

مثال: میانگین موجودی حسابهای شعبی را بیابید که میزان آنها بیش از 1200 دلار باشد

```
select branch_name, avg_balance
from ( select branch_name , avg(balance)
       from account
       group by branch_name )
       as branch_avg (branch_name, avg_balance)
where avg_balance > 1200
```

همین مثال را با having حل کنید

```
select branch_name, avg(balance) as X
from account
group by branch_name
having X > 1200
```

G (account)
branch_name avg(balance) as X

۲- بخش with: همان طور رابطه‌های مشتق را انجام می‌دهد با این تفاوت که رابطه ایجاد شده توسط with، ذخیره می‌شود. (جوری with یک رابطه مشتق تعریف کرده و جایی هم برایش اسم می‌گذاریم)

```
with R( ... , ... ) as
```

P4PCO

مثال: حساب‌هایی با حداکثر موجودی را بیابید.

نام رابطه نام صفت رابطه

```
with max_balance (value) as
    select max (balance)
    from account
```


```
select account_number
from account, max_balance
where account.balance = max_balance.value
```

۸) دیدها (View)

در SQL با create view تعریف می‌شود. رابطه ایجاد می‌شود که از ابتدا در DB ذخیره و با این عملگر (view) این رابطه ایجاد شده save می‌شود

```
create view v as <query expression>
```

← با این دستور یک table می‌سازیم و ذخیره می‌کنیم که افراد اجازه دسترسی به این table را دارند اما از قسمت‌های داخلی آن (مثلاً union) خبر ندارند

سیستم‌هایی که بخش view را ذخیره می‌کنند materialized view نام دارند و با ذخیره این بخش سرعت سیستم بالا می‌رود

مثال: دیدی به نام all_customer ایجاد کنید که شامل شعب و مشتریان آنهاست.

```
creat view all_customer as
    (select branch_name, customer_name
     from depositor, account
     where depositor.account# = account.account#)
  union
    (select branch_name, customer_name
     from borower, loan
     where borower.loan# = loan.loan#)
```

Subject:
Year. Month. Date. ( )

محیط‌های DB
- OLTP (One Line Transiational Pro...ing) ← محیط‌هایی که داده‌ها دائم در حال تغییر هستند (مثل شبکه بانکداری)
- OLAP (One Line Analysis pro...) ← محیط‌هایی که Data در آن‌ها بیشتر آنالیز می‌شود و کمتر تغییر می‌کند

اصلاح پایگاه داده:

حذف (delete): این دستور می‌تواند tuple‌هایی با شرایط خاص را پاک کند ولی نمی‌تواند از یک رکورد فقط مقادیر صفات خاص را حذف کند. (حذف فقط روی یک رابطه انجام می‌شود)

```
delete from r
where p
```

r ← رابطه یا جدول
p ← شرط

مثال: حذف رکوردهایی که در شعب شهر تهران حساب دارند.

```
Delete from account
where branch_name IN( Select branch_name
                      from branch
                      where branch_city = 'Tehran')
```

نکته: اگر بخواهیم تمام چندتایی‌های یک رابطه را حذف کنیم، کافی فقط اسم رابطه را در بخش r مربوط بگذاریم (یعنی بخش where نداریم) مثلاً دستور زیر تمام چندتایی‌های (رکوردهای) رابطه loan را حذف می‌کند

```
delete from loan
```

درج (insert): این دستور tuple‌هایی که مدنظر ماست را به یک رابطه اضافه می‌کند. این دستور به دو شکل زیر به کار می‌رود.

1. `insert into r VALUES ( ,..., )`
به همان ترتیبی که Attr‌های r در DB تعریف شده‌اند می‌نویسیم

2. `insert into r( ..., ..., )` → Attr‌های r را به دلخواه تعریف می‌کنیم
`VALUES ( ..., ... )` → به همان ترتیبی که خودمان تعریف کردیم مقداردهی می‌کنیم

Subject: اصول طراحی پایگاه داده‌ها
Year. 86 Month. 9 Date. 10 25

مثال: یک حساب با شماره حساب A1 در شعبه تهران (اسم شعبه) و با موجودی 1000 دلار به مجموعه حسابها اضافه کنید:

```
1. insert into account
        values ("A1", "Tehran", 1200)
```

مقادیری که در values به ترتیب attهای account در DB اضافه شده اما چنانچه ترتیب account را فراموش کرده بودیم از فرم دوم استفاده می‌کنیم:

```
2. insert into account (branch_name, account#, balance)
        values ("Tehran", "A1", 1200)
```

توجه: دستور زیر به ازای r1 یک ستون اضافه می‌کند.

```
select A1, A2, "Shayan"
from r1
```

r1

A1	A2
1	ali
2	reza
3	nima

⇒

1	ali	Shayan
2	reza	shayan
3	nima	shayan

جلسه دوازدهم 86.9.17

فصل (7)

طراحی پایگاه داده‌ای رابطه‌ای: Relational D.B Designer

anomaly: بی‌نظمی، طراحی خوب باعث کاهش بی‌نظمی می‌شود ← افزونگی کاهش یافته و حافظه مورد نیز هم کاهش می‌یابد (از طرفی وجود افزونگی باعث افزایش سرعت می‌گردد) یعنی وجود افزونگی اگرچه حجم حافظه را افزایش می‌دهد اما باعث افزایش سرعت هم می‌شود.

مثال: فرض کنید می‌خواهیم اطلاعات loan و borrower در یک جدول نشان داده شود و بدین منظور داریم حالت بهتر است.

```
loan(loan#, amount)
borrower(customer-id, loan#)
```

⋈ → bor-loan = (customer-id, loan#, amount)

این ادغام برای ما طراحی خوبی را حاصل نخواهد کرد زیرا Redundency ایجاد می‌شود مثلاً چنانچه

Subject:
Year. Month. Date. ( )

دو مشتری C_1 و C_2 هردو وام L_1 را گرفتند مقدار وام دوبار تکرار می‌شود که احتیاج به تکرار آن نیست.

bor-loan → $C_1, L_1, 10,000\$$ ⟹ redundency
$C_2, L_1, 10,000\$$

راه‌حل (شکستن): Decomposition: گاهی در طراحی به شکستن یک DB نیاز پیدا می‌کنیم مانند مثال بالا که bor-loan باعث redundency شده و بهتر است شکسته شود. (فرض می‌کنیم که borrower و loan را داریم و خودمان می‌خواهیم از روی bor-loan آن را بسازیم)

decomposition ← راه‌حلی که می‌خواهد یک رابطه (R) را به جدول‌هایی که ایجاد کند جدا کرده و برای R بعدی هر چیزی که مانند + کلید R قبلی را وارد می‌کنیم و این کار را تا جایی انجام می‌دهیم که قرار نباشد اطلاعات از بین برود.

bor-loan ↗ (loan#, amount)
↘ (customer-id, loan#)

مثال: loan ⋈ branch
r (branch-name, loan#, amount)

میرداماد	L_1	10,000$	⟹ دارد Redundency
میرداماد	L_2	20,000$	

توجه: اگر جدول‌هایی که به وسیله‌ی آن table‌ها را به هم جسبانیدیم برای هردو کلید table باشد ⟸ نخواهیم داشت Redundency، اما اگر کلید یک جدول هم نباشد ⟸ خواهیم داشت Redundency.

وابستگی تابعی: یعنی به ازای هر x تنها یک y وجود دارد ($x \longrightarrow y$)
و از روی x، مقدار y مشخص است. (تعریف تابع) (x نباید تکرار شود و اگر تکرار شد y هم باید تکرار شود)
مثال: loan # ⟶ amount : Functional Dependency

تعریف طبیعی: به یک یا مجموعه att هایی گفته می‌شود که تمام att های دیگر رابطه به آن وابستگی تابعی داشته باشند طبیعی است.

مثال: branch_name ⟶ branch_city, assets

مثال: روی $r(A,B,C,D)$ وابستگی‌های تابعی را مشخص کنید.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_3
a_2	b_3	c_2	d_4

$A \longrightarrow C$

$C \longrightarrow A$

$D \longrightarrow B$ ✓

(در این مثال 3 وابستگی تابعی وجود دارد (یعنی هر سه رابطهٔ نوشته شده تابع (تعریف ریاضی) هستند)

توجه: وابستگی تابعی را نمی‌توان از روی داده‌های table تشخیص داد زیرا در اثر اضافه شدن داده‌ها وابستگی تغییر خواهد کرد ← وابستگی تابعی را باید از روی ذات سیستم تشخیص دهیم.

تعریف ریاضی طبیعی:

$K \longrightarrow R \quad : \quad K \subseteq R$

$r(R)$

$t_1, t_2 : t_1 \neq t_2 \Longrightarrow t_1[k] \neq t_2[k]$

K یک super key خواهد بود هرگاه: (یک یا چند att)

or

$R : \alpha \subseteq R, \quad \beta \subseteq R$

$\alpha \longrightarrow \beta \quad : \forall t_1, t_2 \quad t_1[\alpha] = t_2[\alpha] \Longrightarrow t_1[\beta] = t_2[\beta]$

یعنی اگر در یک وابستگی تابعی دو تاپل مولفه اول (یا) یکسان داشتند آنگاه مولفه دوم آنها نیز باید یکسان باشد و در غیر این صورت $\alpha \to \beta$ وابستگی تابعی خواهد بود و α نیز نمی‌تواند طبیعی باشد.

سوال: چه وقت k طبیعی R است؟

1. $K \subseteq R$ ؛ 2. $K \longrightarrow R$ (یعنی k همهٔ att های R را بدهد)

Subject:
Year. Month. Date. ( )

فرم‌های نرمال (Normal Form) :

۱- 1NF ← (first normal form)
۲- 2NF ← normal form)
۳- BCNF ← (Boyce Codd NF)
۴- 3NF ← normal form)

- فرم نرمال اول (1NF) : اگر دامنه‌های تمام att های شمای یک رابطه (R) اعیب باشند آنگاه رابطه R فرم نرمال اول خواهد بود (atomic ← عناصر دامنه متعدد و غیرنرمال تجزیه باشند) به بیان دیگر رابطه‌ای که صفات Multivalued و Compose نداشته باشد 1NF است.

← برای اینکه شمای رابطه‌ای 1NF شود باید تمای صفاتش اعیب باشند بنابراین:

۱- صفت Compose (مثل address) را حذف کرده و att هایش را مستقیماً به Entity وصل می‌کنیم

۲- به ازای کلید صفت Multivalued به تعداد که می‌خواهیم به table تاپل اضافه می‌کنیم مثلاً صفت tel که چند مقداری است به صورت صفت ساده در نظر گرفته و دو عضو تعداد تاپل‌ها را افزایش می‌دهیم

کلید	tel
A_1	~
A_1	~
A_1	—
A_2	

- فرم نرمال دوم (2NF) : فرمی بهتر از فرم 1NF است اما از آنجایی که کاربرد کمی دارد و جزو مواردی استعداد قرار نمی‌گیرد این فرم را بررسی نمی‌کنیم و BCNF را مورد بررسی قرار می‌دهیم.

P4PCO

Subject: اصول طراحی پایگاه داده‌ها
Year. 86 Month. 9 Date. 17 27

- فرم نرمال بویس ـ کد: شمای رابطه R در صورتی به فرم نرمال BCNF است که، برای تمام وابستگی‌های تابعی موجود در F^+ به شکل $\alpha \to \beta$، که $\alpha \subseteq R$, $\beta \subseteq R$ است، حداقل یکی از موارد زیر برقرار باشد:

1. $\alpha \to \beta$ Trivial : یعنی $\beta \subseteq \alpha$ یک وابستگی تابعی بدیهی است
2. α is a superkey of R : یعنی α تمام صفات R را مشخص (روشن) کند

مثال:

loan (loan#, amount)
loan# ⟶ amount
BCNF است (شرط دوم) ✓

borrower (customer-id, loan#)
customer-id ⟶ loan#
BCNF است (شرط اول) ✓

bor-loan (customer-id, loan#, amount)
loan# ⟶ amount
BCNF نیست (هیچ یک از دو شرط را ندارد) ✗

نکته: یک رابطه برای اینکه BCNF باشد باید حتماً 1NF باشد یعنی باید شرایط 1NF را هم داشته باشد تا یک BCNF باشد یعنی

$1NF \subseteq BCNF \subseteq 3NF$

توجه: اگر $\alpha \to \beta$ را پیدا کردیم که هیچ‌یک از شروط مذکور را نداشت باید آن رابطه decompose شود به این صورت که رابطه‌ای به صورت $(\alpha \cup \beta)$ درنظر گرفتیم و رابطه دوم به صورت $R - (\beta - \alpha)$ تعریف می‌گردد.

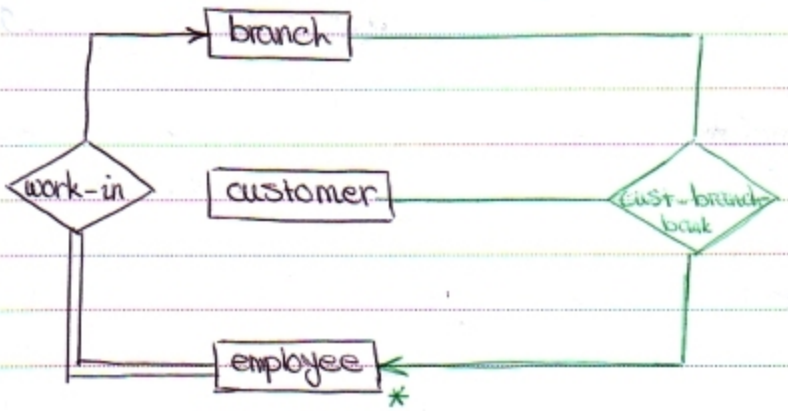
- فرم نرمال سوم (3NF): در این فرم شرط سومی به دو شرط فرم BCNF اضافه می‌شود:

1. $\alpha \to \beta$ trivial
2. α be a superkey
3. each att A in $\beta - \alpha$ in contained in a candidate key

↳ P4PCO اگر β جزء از کلید بود R نباید decompose شود $(\alpha \to \beta)$

یعنی در فرم 3NF محدودیت BCNF را ندارد

مثال: می خواهیم هر customer در هر شعبه فقط با یک employee سروکار داشته باشد.
(شکل کتاب غلط است زیرا بین customer و branch هیچ رابطه ای در نظر نگرفته)



بدیهی رابطه سه تایی
تعریف کنیم که هر
شعبه و مشتری
فقط باید یک کارمند
دارا باشد

cust-branch-bank(branch-name, customer-id, employee-id, type)

employee-id → branch-name

رابطه * BCNF نیست زیرا $\alpha \rightarrow \beta$ در آن پیدا کردیم که هیچ یک از دو شرط را ندارد.
بنابراین باید رابطه را بشکنیم.

r_1(employee-id, branch-name)
r_2(customer-id, employee-id, type)
$r_2 = R - (\beta - \alpha)$

← همانطور که می بینیم کلید رابطه ی اصلی با شکستن آن از بین می رود بنابراین تجزیه ما غلط بوده و نباید تجزیه کنیم.

← وقتی اجازه داریم تجزیه کنیم که وابستگی های تابعی حفظ شود.
← بنابراین 3NF طراحی شد که کامل تر از 2NF و BCNF است و در واقع سخت گیری BCNF را ندارد و بیان می کند که اگر هم جزئی از کلید بود دیگر تجزیه لازم نیست.

Subject: اصول طراحی پایگاه داده ها
Year. 86 Month. 9 Date. 24 28

جلسه دوازدهم 86.9.24

بستار F ← F^+: عبارتست از تمام وابستگی هایی که میشود از F نتیجه گرفت ← یعنی F^+ نسبت به استتاج F بسته است و هر نتیجه ای از F در F^+ قرار دارد.

مثال: $F = \{A \rightarrow B, B \rightarrow C\}$

$\Rightarrow \{A \rightarrow C\} \in F^+$

تئوری وابستگی تابعی (Functional dependency Theory):

* data redundancy ← تکرار بیهوده داده ها
* اصل ← چیزی است که فرض میکنیم درست است اما آن را ثابت نمی کنیم.

اصول آرمسترانگ (Armstrong Axioms):

ثابت کرد با سه قانون زیر می توانیم از F به F^+ برسیم: Armstrong

1- Reflexivity Rule: $\alpha \rightarrow \beta : \beta \subseteq \alpha$ مثال $\{AB \rightarrow A, AB \rightarrow B\}$

2- Augmentation Rule: $\alpha \rightarrow \beta \Rightarrow \gamma\alpha \rightarrow \gamma\beta$

3- Transitivity Rule: $\alpha \rightarrow \beta \ \& \ \beta \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma$

اصول زیر را از اصول Armstrong نتیجه می گیریم:

اجتماع → Union Rule: $\alpha \rightarrow \beta \ \& \ \alpha \rightarrow \gamma \Rightarrow \alpha \rightarrow \beta\gamma$

اثبات: $\alpha \rightarrow \beta \Rightarrow \alpha \rightarrow \alpha\beta$ ①

$\alpha \rightarrow \gamma \Rightarrow \alpha\beta \rightarrow \gamma\beta$ ②

①,② $\rightsquigarrow \alpha \rightarrow \alpha\beta$ and $\alpha\beta \rightarrow \gamma\beta \Rightarrow \alpha \rightarrow \gamma\beta$ ✓

تجزیه → Decomposition Rule: $\alpha \rightarrow \beta\gamma \Rightarrow \alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$

اثبات: $\beta\gamma \rightarrow \beta$ and $\beta\gamma \rightarrow \gamma$

$\alpha \rightarrow \beta\gamma$ and $\beta\gamma \rightarrow \beta \Rightarrow \alpha \rightarrow \beta$ ✓

$\alpha \rightarrow \beta\gamma$ and $\beta\gamma \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma$ ✓

P4PCO

Subject:
Year Month Date ( )

→ Pseudotransitivity Rule: $\alpha \to \beta$ and $\gamma\beta \to \delta \Rightarrow \gamma\alpha \to \delta$

اثبات $\alpha \to \beta \Rightarrow \gamma\alpha \to \gamma\beta$

$\gamma\alpha \to \gamma\beta$ and $\gamma\beta \to \delta \Rightarrow \gamma\alpha \to \delta$ ✓

الگوریتم به دست آوردن F^+ از روی F:

این الگوریتم پرهزینه بوده و خیلی مورد استفاده قرار نمی‌گیرد.

* در ابتدای کار فرض می‌کنیم F^+ همان F است ←

$F^+ = F$

REPEAT

FOR EACH functional dependency f in F^+:

apply reflexivity and augmentation rules on f

add the resulting ... into F^+

FOR EACH pair of functional dep. f_1, f_2 in F^+

if f_1, f_2 can be combined using 3 add resulting F^+

UNTIL F^+ doesn't change any further.

این الگوریتم می‌گوید که 3 قانون آرمسترانگ را آنقدر روی F انجام بده تا F^+ دیگر تغییر نکند. وقتی از حلقه خارج می‌شویم F^+ کامل شده.

Clouser of att. Set:

از آنجایی که الگوریتم تولید F^+ بسیار پرهزینه است بهتر است α^+ را تعریف کنیم:

α^+ ← α : مجموعه تمام صفاتی که از α به صورت Functionally نتیجه می‌شود را α^+ می‌نامند؛ از روی α^+ موارد زیر نتیجه گرفته می‌شود:

۱- اگر β در α^+ باشد آنگاه $\alpha \to \beta$ صحیح است. (یعنی اگر بخواهیم ببینیم که $\alpha \to \beta$ متعلق به F^+ است یا نه، کافی است ببینیم β در α^+ است یا نه ← کم هزینه‌تر)

۲- می‌توانیم بفهمیم که α سوپر کلید است یا نه.

۳- بدون اجرای الگوریتم F^+ می‌توانیم F^+ را به دست آوریم

Subject: اصول طراحی پایگاه داده‌ها
Year 86 Month 9 Date 24 29

الگوریتم به دست آوردن α^+ از روی α :

result = α → به دلیل برقراری قانون 1 α همواره در α^+ است.
while (change to result) do
  For each functional dependency $\beta \rightarrow \gamma$ F do
    begin
      if $\beta \subseteq$ result then result = result $\cup$ γ
    end
این if قانون transitivity را بیان می‌کند

مثال:
$F = \{AB \rightarrow C, A \rightarrow D\}$
A را داریم D را هم اضافه می‌کنیم — A, B را داریم C را اضافه می‌کنیم
$AB^+ = \{A, B, C, D\}$
در این مثال اگر $R = \{A, B, C, D\}$ باشد آنگاه AB^+ سوپر کلید است زیرا بقیه R را تولید می‌کند.

* همانطور که قبلاً اشاره کردیم یکی از مزایای α^+ به دست آوردن F^+ (بدون استفاده از الگوریتم F^+) است که در زیر این الگوریتم هم بررسی می‌کنیم:
۱- به ازای هر γ که زیر مجموعه R است (بدون در نظر گرفتن تهی) γ^+ را به دست می‌آوریم:
مثلاً اگر $R = (A, B, D, C)$ ← $2^4 - 1$ ← γ^+ تا (15 =) داریم.
۲- به ازای هر زیر مجموعه γ^+ که در مرحله قبل γ را به دست آوردیم) داریم:

$S \subseteq \gamma^+ \Rightarrow \gamma \rightarrow S$

یعنی $\gamma \rightarrow S$ را به مجموعه F^+ اضافه می‌کنیم ← F^+ کامل می‌شود

مثال:
$ABC^+ = \{A, B, C\}$ → $2^3 - 1 = 7$
7 عضو باید به F^+ اضافه شود

1. $ABC \rightarrow A$
2. $ABC \rightarrow B$
3. $ABC \rightarrow C$
4. $ABC \rightarrow AB$
5. $ABC \rightarrow AC$
6. $ABC \rightarrow BC$
7. $ABC \rightarrow ABC$
8. $ABC \rightarrow \phi$ ~~~ نمی‌نویسیم

P4PCO

Subject:
Year. Month. Date. ( )

Canonical Convert :

کانونی کردن یعنی اضافی ها را حذف کرده و F^+ را مختصر و مفید کنیم.

- هر چه تعداد اعضای مجموعه F کمتر باشد Canonical Convert سریع تر انجام می شود

آرگ- از روی F ، F_c را ایجاد می کنیم به طوری که هیچ محدودیتی (قانونی) از F حذف نشود

. یعنی باید $F^+ = F_c^+$ باشد ← در این صورت از F_c به مختصر تر است استفاده می کنیم.

مثال :

$F = \{ A \to B,\ B \to C,\ A \to C \}$

می توانیم $A \to C$ را حذف کنیم زیرا دو قانون اول ، نوی را نتیجه می دهند ←

$F_c = \{ A \to B,\ B \to C \}$ ⇝ هیچ محدودیتی از F را نبردیم.

* اگر $\alpha \to \beta$ را از F انتخاب کرده باشیم آرگ- داریم :

مثال (A اضافی) $AB \to C$
$\Rightarrow B \to C$

۱- A در α برای extranet است ← A را حذف می کنیم

$A \in \alpha : F' = \{ (F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\} \} \Rightarrow F'^+ = F^+$

مثال (A اضافی) $B \to AC$
$B \to C$

۲- A در β برای extranet است ← A را حذف می کنیم

$A \in \beta : F' = \{ (F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\} \} \to F'^+ = F^+$

مثال : در موارد زیر F' را به دست آورید.

(1) $F = \{ AB \to CD,\ A \to CD \}$

$F' = \{ A \to CD \}$

در F ، A به تنهایی CD را نتیجه می دهد بنابراین B در $(AB \xrightarrow{\alpha} CD)$ برای extranet است

بنابراین B را حذف می کنیم ← آنچه باقی می ماند یعنی $A \to CD$ است که دیگر احتیاج به تکرار آن نخواهد بود و مجموعه F' یک عضو خواهد داشت :

$F' = \{ A \to CD \}$

(2) $F = \{ AB \to CD,\ A \to E,\ E \to CD \}$

$F' = \{ A \to CD,\ A \to E,\ E \to CD \}$

در این مثال از دو قانون $CD \to A$ نتیجه می شود بنابراین B در $AB \to CD$ اضافی و قابل حذف است.

PqPCO

(3) $F = \{AB \rightarrow CD, A \rightarrow C\}$

$F' = \{AB \rightarrow D, A \rightarrow C\}$ ← A همیشه C را می‌دهد ← C اضافی است

اثبات: $AB \rightarrow A$ ⇒ $AB \rightarrow C$ ①

نتیجه $A \rightarrow C$

① and $AB \rightarrow D \Rightarrow AB \rightarrow CD$

F_c مجموعه‌ای است S.t به ازای هر $\alpha \rightarrow \beta$ هیچ att چه در α و چه در β extranet نباشد و سمت چپ هیچ دو قانونی یکسان نباشد.

$\alpha_1 \rightarrow \beta$ and $\alpha_2 \rightarrow \gamma \Rightarrow \alpha_1 \neq \alpha_2$

الگوریتم به دست آوردن F_c از روی F:

$F_c = F$ * F_c منحصر به فرد نیست *

repeat

- use the union rule to replace any dependencies in F_c of the form $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$
- Find a Functional dependency $\alpha \rightarrow \beta$ in F_c with extraneous att. either in α or in β
- If an extraneous att. is fo n, delet it from $\alpha \rightarrow \beta$

until F_c dose not change.

مثال: F_c بیابید.

(کتاب این مثال را مفصل انویتم بالا حل کرده.)

$A \rightarrow BC$

$B \rightarrow C$

$A \rightarrow B$ ← A به تنهایی B را می‌دهد

$A \rightarrow BC$ ⟶ (اضافی B) $A \rightarrow C$

$AB \rightarrow C$ ⟶ $A \rightarrow C$ and $B \rightarrow C$ ⟶ $AB \rightarrow C$ حذف

$\Rightarrow B \rightarrow C, A \rightarrow B, A \rightarrow C$ (اضافی) $\Rightarrow F_c = \{A \rightarrow B, B \rightarrow C\}$

P4PCO

Subject:
Year. Month. Date.

$\{A \xrightarrow{*} BC,\ B \longrightarrow AC,\ C \longrightarrow AB\}$

- در قدم اول دنبال تابعی هایی می گردیم که سمت چپ یکسان دارند ⇐ نداریم
- در مرحله بعدی extranet ها را پیدا کرده و حذف می کنیم

$\xrightarrow{*}$ $A \to B$; $B \to AC$; $C \to AB$ → $A \to B$; $B \to AC$; $C \to A$ → $A \to B$, $B \to C$, $C \to A$ F_c

$\xrightarrow{*}$ $A \to C$; $B \to AC$; $C \to AB$ → $A \to C$; $B \to A$; $C \to AB$ → $A \to C$, $B \to A$, $C \to B$ F_c

$\xrightarrow{*}$ $A \to C$; $B \to AC$; $C \to AB$ → $A \to C$, $B \to C$, $C \to AB$ F_c

$\xrightarrow{*}$ $A \to B$; $B \to AC$; $C \to AB$ → $A \to B$, $B \to AC$, $C \to B$ F_c

همانطور که قبلاً هم ذکر کرده بودیم F_c منحصر به فرد نیست و در این مثال هم می بینیم که 4 ، F_c وجود دارد

نتیجه: تجزیه به منظور کاهش data redundancy انجام می شود و وقتی تجزیه صحیح است که از نظر اطلاعاتی داده ای را از دست ندهیم (یعنی تجزیه باید lossless باشد)

lossy decomposition ← تجزیه باز دست دادن اطلاعات داده ای

lossless decomposition ← تجزیه بدون از دست دادن اطلاعات داده ای

نکته: فرض کنید رابطه r را روی شمای R تعریف کرده و عمل تجزیه را انجام دهیم: $r(R)/(R_1, R_2)$

$\Rightarrow \prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$ ⇝ lossless decomposition

$\neq r$ ⇝ lossy decomposition

وباتوجه به نکته قبلی وقتی عمل تجزیه lossless است که یکی از شرایط زیر برقرار باشد:

a) $R_1 \cap R_2 \longrightarrow R_1$

b) $R_1 \cap R_2 \longrightarrow R_2$

یعنی اشتراک R_1, R_2 (صفت مشترک) باید یا کلید اولی باشد یا کلید دومی و یا کلید هردو تا lossless decomposition داشته باشیم و درغیراین صورت تجزیه lossy خواهد بود.

جلسه سیزدهم 86.10.1