

# 数据类型

结构化数据 (Structured Data): 具备严格的模式，可被组织进关系型数据库中。

非结构化数据 (Unstructured Data): 无需预先定义，数据格式不固定。

文件系统与数据库系统的对比 (File Systems vs Database Systems)

文件系统缺点包括：数据冗余、不一致性、数据隔离、完整性问题、并发访问问题。

## 1. 数据冗余 (Data Redundancy)

- 专业定义：同样的数据在多个文件中重复存储

- 造成的问题：存储空间浪费、数据更新困难、可能导致数据不一致

## 2. 数据不一致性(Data Inconsistency)

- 专业定义：同一数据在不同位置的多个副本之间存在矛盾

- 例如：同一客户信息在不同文件中的地址不同

## 3. 数据隔离(Data Isolation)

- 专业定义：数据分散在不同文件中，难以访问和组合查询

- 影响：难以生成综合报表或进行复杂查询

## 4. 完整性问题(Data Integrity Problems)

- 专业定义：难以在文件系统层面实施统一的完整性约束

- 包括:
  - 实体完整性 (Entity Integrity)
  - 参照完整性 (Referential Integrity)
  - 域完整性 (Domain Integrity)

## 5. 并发访问问题(Concurrent Access Problems)

- 专业定义: 多用户同时访问和修改数据时的同步控制问题
- 英文表述: Concurrent Access Control
- 相关概念:
  - 并发控制 (Concurrency Control)
  - 死锁 (Deadlock)
  - 事务处理 (Transaction Processing)

这些问题在现代数据库管理系统(DBMS)中通过以下机制得到解决:

1. 数据字典(Data Dictionary): 控制冗余
2. 事务管理(Transaction Management): 保证一致性
3. 统一管理(Centralized Management): 解决隔离问题
4. 约束机制(Constraint Mechanism): 保证完整性
5. 并发控制(Concurrency Control): 处理并发访问

数据库管理系统的特性

数据独立性 (Data Independence)

高效的数据访问 (Efficient Data Access)

数据完整性与安全性 (Data Integrity and Security)

数据管理 (Data Administration)

并发访问与崩溃恢复 (Concurrent Access and Crash Recovery)

数据库设计 (Database Design)

概念设计 (Conceptual Design): 利用工具进行需求表示，便于维护和转化为数据库实现。

逻辑设计 (Logical Design): 通过概念设计转换为数据模型并实现于 DBMS。

物理设计 (Physical Design): 进一步规范数据库的存储与访问。

数据库语言 (Database Languages)

数据定义语言 (DDL): 用于定义概念模式。

数据操作语言 (DML): 用于请求和操作数据，有非过程性 DML (如 SQL) 和过程性 DML。

数据库用户类型:

数据库管理员 (Database Administrator, DBA): 负责安全、授权、数据恢复和数据库调优。应用程序员 (Application Programmer): 实现具体需求。最终用户 (End User): 实际使用数据库的用户。

数据库模型:

数据模型的级别包括: 高层次或概念模型 (如 ER 模型)、实现模型 (如关系模型) 和物理模型 (低层次)。

数据库管理系统类型:

主要包括关系型 (Relational)、键值型 (Key/Value)、图形型 (Graph)、文档型 (Document)、列族型 (Column-family)。

数据库历史 (History of Database Systems)

1960s: 通用数据库管理系统 (Integrated Data Store)。

1970s: 关系模型的提出 (Edgar Codd)。

1980s: SQL 标准化及事务概念提出 (Jim Gray)。

1990s 至今: 关系型数据库、NoSQL、大数据分布式处理的发展。

数据建模 (Data Modelling):

概念模型 (Conceptual Model): 抽象的高层数据模型, 例如 ER 模型和 ODL (Object Data Language), 用户友好。

逻辑模型 (Logical Model): 为具体的数据库管理系统(DBMS)实现的模型, 例如关系模型。

物理模型 (Physical Model): 在具体 DBMS 内部的文件存储方式。

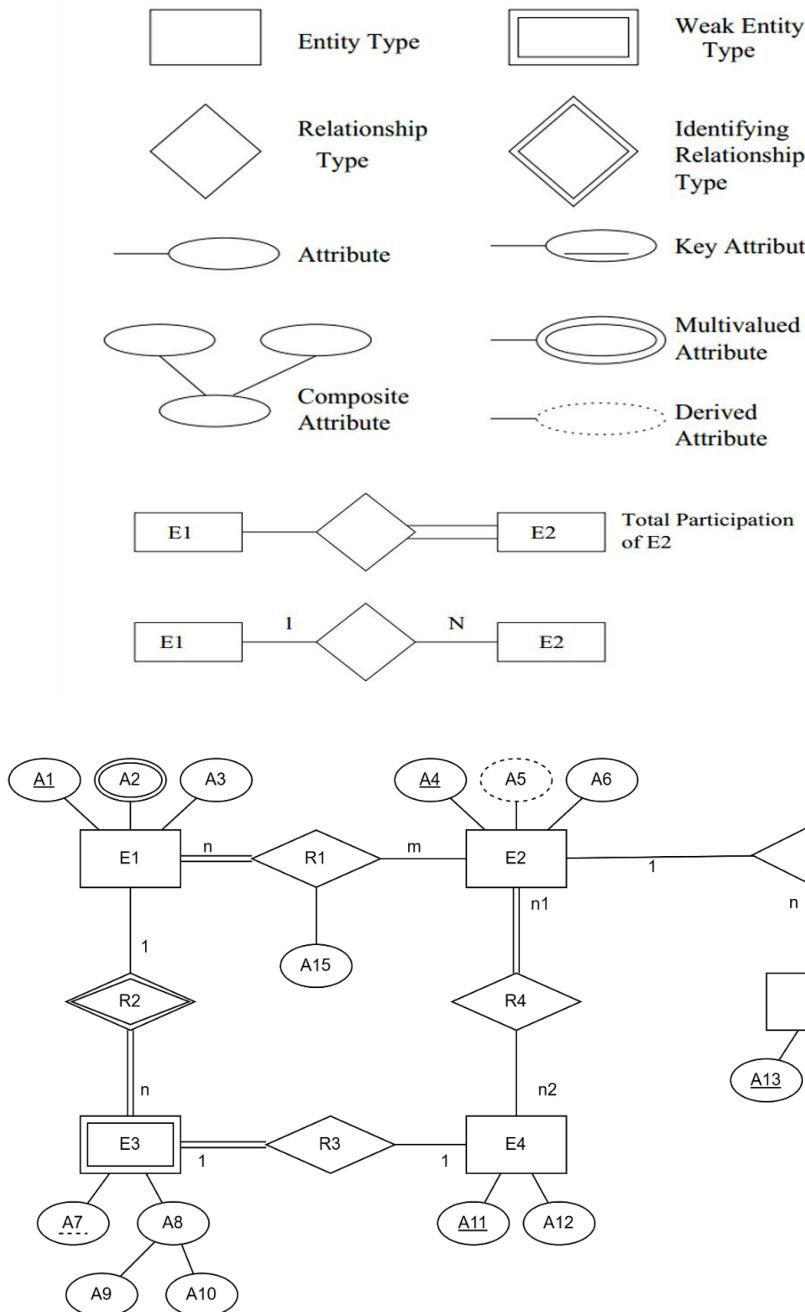
设计阶段 (Design Stages): 概念设计 (Conceptual Design)、逻辑设计 (Logical Design)、物理设计 (Physical Design)

# 实体-关系模型 (Entity-Relationship Model, ER):

实体 (Entity): 描述感兴趣对象的属性集合。

关系 (Relationship): 实体之间的关联。

属性 (Attribute): 描述实体某种特性的具体数据项。



属性类型 (Attribute Types):

简单属性 (Simple Attribute): 不可再分的属性。

复合属性 (Composite Attribute): 可分为更小的子属性，例如地址 (Address) 可以分为街道 (Street) 和郊区 (Suburb)。

多值属性 (Multi-Valued Attribute): 可以有多个值的属性，例如衬衫的颜色。

派生属性 (Derived Attribute): 可以从其他存储属性推导得出，例如年龄可以从出生日期推导。

实体类型 (Entity Type): 定义具有相同属性的实体集合。在 ER 图中用矩形框表示。

键 (Keys):

超键 (Super Key): 能唯一标识实体实例的一组属性。

候选键 (Candidate Key): 最小的超键，即没有子集能成为超键。

主键 (Primary Key): 由数据库设计者选择的候选键，用于唯一标识实体。

弱实体 (Weak Entity): 没有自己的主键，通常通过部分键（与所有者实体关联的关系）进行标识。

关系约束 (Relationship Constraints):

基数约束 (Cardinality Constraint): 实体参与关系实例的数量限制，如一对一 (1:1)、一对多 (1:N)、多对多 (N:M)。

参与约束 (Participation Constraint): 实体实例是否必须参与某个关系，例如全部参与 (Total Participation) 和部分参与 (Partial Participation)。

关系属性 (Relationship Attributes): 关系可以有自己的属性，例如研究人员在项目上的工作时间。

关系的度 (Degree of Relationship): 参与关系的实体数量，例如二元关系 (Binary Relationship) 和三元关系 (Ternary Relationship)。

ER MODEL	RELATIONAL MODEL
Entity Type	<i>Entity</i> relation
1:1 or 1:N relationship type	Foreign key (or <i>relationship</i> relation)
M:N relationship type	<i>Relationship</i> relation and two foreign key
<i>n</i> -ary relationship type	<i>Relationship</i> relation and <i>n</i> foreign key
Simple Attribute	Attribute
Composite Attribute	Set of simple component attributes
Multivalued Attribute	Relation and foreign key

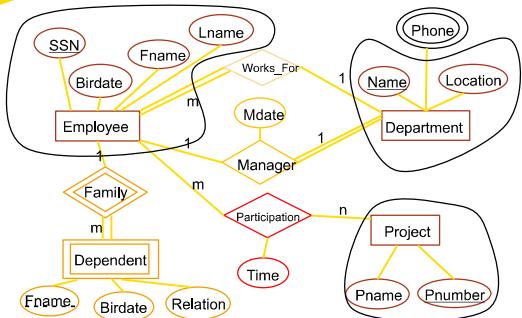
## Mapping Strong Entity Types

Step 1: For each ***strong entity*** (not weak entity) type E, create a new relation R with

- Attributes: all *simple attributes* (and simple components of composite attributes) of E.
- Key: key of E as the *primary key* for the relation.

36

## Mapping Strong Entity Types



37

## Mapping Strong Entity Types

Employee	[SSN   Fname   Lname   Birdate]
Department	[Name   Location]
Project	[Pnumber   Pname]

38

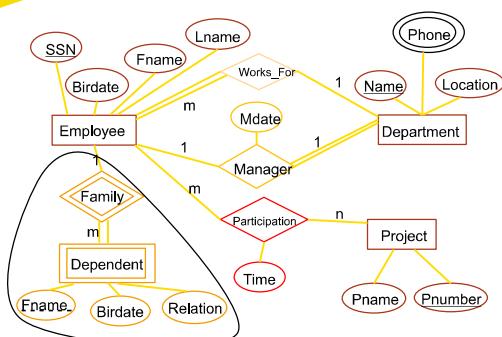
## Mapping Weak Entity Types

Step 2 : For each ***weak entity type*** W with the owner entity type E, create a new relation R with

- Attributes :
- all simple attributes (and simple components of composite attributes) of W,
- and include the primary key attributes of the relation derived from E as the foreign key,
- Key of R: foreign key to E and partial key of W.

39

## Mapping Weak Entity Types



40

## Mapping Weak Entity Types

Employee	[SSN   Fname   Lname   Birdate]
Department	[Name   Location]
Project	[Pnumber   Pname]
Dependent	[SSN   Fname   Birdate   Relation]

41

## Mapping 1:1 Relationship Types

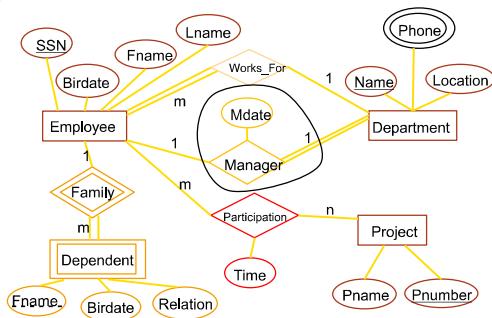
Step 3 : For each **1:1 relationship type** B. Let E and F be the participating entity types. Let S and T be the corresponding relations.

- Choose one of S and T (let S be the one that participates totally if there is one).
- Add attributes from the primary key of T to S as a foreign key.
- Add all simple attributes (and simple components of composite attributes) of B as attributes of S.

(Alternatively, merge the two entity types and the relationship into a single relation, especially if **both participate totally and do not participate in other relationships**).

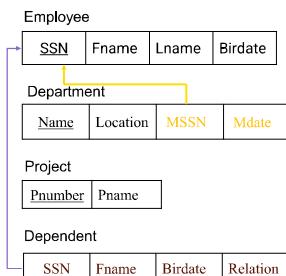
42

## Relationship Types



43

## Mapping 1:1 Relationship Types



44

## Mapping 1:N Relationship Types

Step 4 : For each **1:N relationship type** B. Let E and F be the participating entity types. Let S and T be the corresponding relations. Let E be the entity on the 1 side and F on the N side.

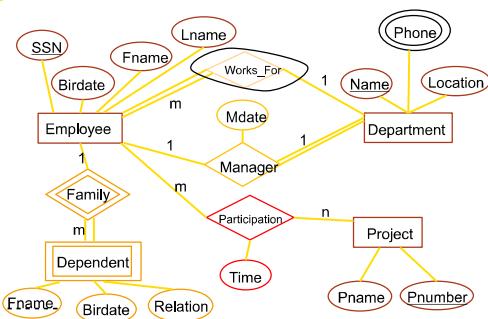
Add to the relation belonging to entity T,

- the attributes from the primary key of S as a foreign key.
- any simple attributes (or simple components of composite attributes) from relationship B.

(Notice that this doesn't add any new tuples, just attributes.)

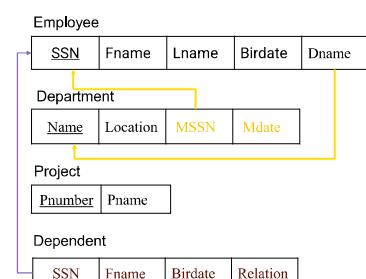
45

## Mapping 1:N Relationship Types



46

## Mapping 1:N Relationship Types



47

## Mapping M:N Relationship Types

Step 5: For each **N:M relationship type** B. Let E and F be the participating entity types. Let S and T be the corresponding relations

Create a new relation R (cross-reference) with

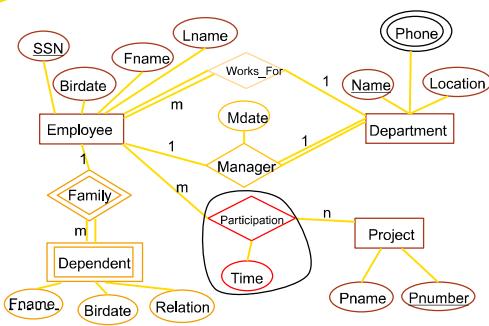
Attributes :

- Attributes from the key of S as a foreign key,
- Attributes from the key of T as a foreign key,
- Simple attributes and simple components of composite attributes of relation B.

Key: All attributes from the key of S and T.

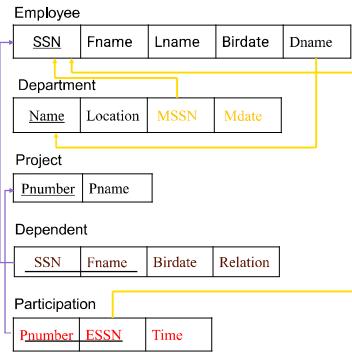
48

## Mapping M:N Relationship Types



49

## Mapping M:N Relationship Types



50

## Mapping Multivalued Attributes

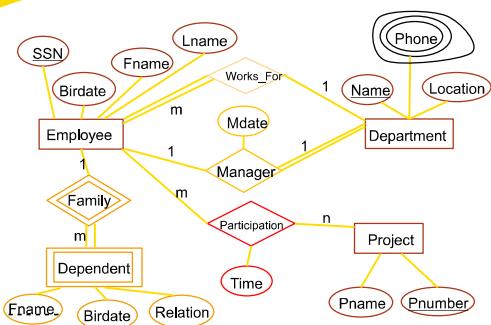
Step 6: For each **multivalued attribute** A, where A is an attribute of E, create a new relation R.

- If A is a **multivalued simple attribute**,  
Attributes of R = Simple attribute A, and key of E as a foreign key.
- If A is a **multivalued composite attribute**,  
Attributes of R = All simple components of A, and key of E as a foreign key.

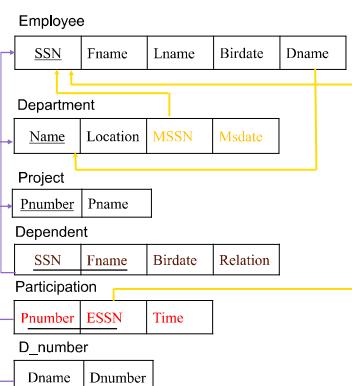
In both cases, the primary key of R is the set of all attributes in R.

51

## Mapping Multivalued Attributes



52



53

## Mapping N-ary Relationship Types

Step 7: For each **N-ary relationship type** ( $n > 2$ ), create a new relation with

➤ Attributes: same as Step 5.

➤ Key: same as Step 5

(Advice: *binary relationships are simpler to model.*)

# 关系代数 (Relational Algebra)

## 1. 选择操作 (SELECT)

- 选择满足特定条件的元组。
- 记号:  $\sigma$ <选择条件>(R), 如  $\sigma \text{ age} \leq 24(\text{Student})$  表示选择年龄小于等于 24 的学生。

## 2. 投影操作 (PROJECT)

- 用于选择关系中的部分属性 (列), 并删除重复的元组。
- 记号:  $\pi$ <属性列表>(R), 如  $\pi \text{ name, age}(\text{student})$  表示只选择学生关系中的姓名和年龄属性。

## 3. 并集操作 (UNION)

- 将两个关系中的元组合并, 结果包含 R 或 S 或者两者都包含的元组。
- 记号:  $R \cup S$ , 要求 R 和 S 具有相同的属性集。

## 4. 交集操作 (INTERSECTION)

- 返回两个关系中都包含的元组。
- 记号:  $R \cap S$ , 要求 R 和 S 具有相同的属性集。

## 5. 差集操作 (DIFFERENCE)

- 返回在关系 R 中存在但不在关系 S 中的元组。
- 记号:  $R - S$ , 要求 R 和 S 具有相同的属性集。

## 6. 笛卡尔积操作 (CARTESIAN PRODUCT)

- 产生 R 和 S 中所有元组的组合, 结果的属性包含 R 和 S 的所有属性。
- 记号:  $R \times S$ , 如  $R \times S$  表示 R 和 S 的所有可能组合。

## 7. 连接操作 (JOIN)

- 连接是用于将两个关系中相关的元组合并为单个“更长”元组。
- **θ 连接 (Theta-join):** 条件连接, 记号:  $R \bowtie_{\theta} S$ , 如  $Student \bowtie_{CourseID=CourseID} Course$ 。
- **等值连接 (Equi-join):** 仅使用等于比较的连接, 记号同 θ 连接, 如  $Student \bowtie_{StudentID=CourseID} Course$ 。
- **自然连接 (Natural Join):** 基于属性名和域相同的属性进行等值连接, 重复的属性只保留一个, 如  $Student \bowtie Enrolment$ 。

## 8. 除法操作 (DIVISION)

- 用于找出 R 中与 S 中的所有元组关联的元组。
- 记号:  $R \div S$ , 通常用于回答“哪些课程被所有部门开设”之类的问题。

假设我们有以下两个关系:

1. **R (学生, 课程):** 表示每个学生选修的课程。

学生 课程

---- ----

张三 数学

张三 物理

李四 数学

李四 物理

王五 数学

2. **S (课程):** 表示所有的课程。

课程

----

数学

物理

**问题:** 找出选修了所有课程的学生。

### 计算除法 $R \div S$

在这个例子中, 我们想找到那些选修了所有课程 (即数学和物理) 的学生。

- $R \div S$  的结果:

学生

----

张三

李四

### 9. 重命名操作 (RENAME)

- 用于更改关系或属性的名称。
- 记号:  $\rho<\text{新名称}>(R)$ , 如  $\rho(\text{监护人}, \text{孩子})(\text{家人})$  表示将关系“家人”的属性“父母”重命名为“监护人”。

### 10. 聚合操作 (Aggregate Operators)

- 用于求取诸如“员工工资总和”或“学生平均年龄”等聚合信息。
- 使用记号  $\gamma$ , 例如  $\gamma(\text{SUM}(\text{工资}))(\text{员工})$  表示计算员工工资的总和。

## 例题描述

我们有以下四个关系：

### 1. 学生 (Student)

学生ID	姓名	年龄
1	张三	20
2	李四	21
3	王五	20
4	赵六	22

### 2. 课程 (Course)

课程ID	课程名
101	数学
102	物理
103	化学

### 3. 选课 (Enrolment)

学生ID	课程ID	成绩
1	101	85
1	102	90
2	101	78
3	103	88
4	102	82
4	101	95

### 4. 导师 (Advisor)

导师ID	导师名	学生ID
201	陈教授	1
202	王教授	2
203	李教授	3
201	陈教授	4

## 要求和操作

通过这些关系，回答以下复杂的问题，尽量综合使用各种运算。

## 1. 找出所有学生及其所选课程的详细信息，并显示学生的导师姓名

运算步骤：

- **连接操作 (JOIN) :**

1. 首先通过 学生 和 选课 关系进行连接，找出学生的选课信息。

```
Student ⋈(学生ID=学生ID) Enrolment
```

2. 再将上一步的结果与 课程 关系进行连接，得到课程名。

```
(Student ⋈ Enrolment) ⋈(课程ID=课程ID) Course
```

3. 最后，再与 导师 关系连接，得到每个学生的导师信息。

```
((Student ⋈ Enrolment) ⋈ Course) ⋈(学生ID=学生ID) Advisor
```

- **结果：**

学生ID	姓名	年龄	课程ID	课程名	成绩	导师ID	导师名
1	张三	20	101	数学	85	201	陈教授
1	张三	20	102	物理	90	201	陈教授
2	李四	21	101	数学	78	202	王教授
3	王五	20	103	化学	88	203	李教授
4	赵六	22	102	物理	82	201	陈教授
4	赵六	22	101	数学	95	201	陈教授

## 2. 找出平均年龄超过 20 岁的导师所指导的学生名单

运算步骤：

- 1. **选择操作 (SELECT) :**

- 首先选择所有年龄超过 20 的学生。

```
σ(年龄 > 20)(Student)
```

- 2. **连接操作 (JOIN) :**

- 将上一步的结果与 导师 关系进行连接，以找到这些学生的导师信息。

$$\sigma(\text{年龄} > 20)(\text{Student}) \bowtie (\text{学生ID}=\text{学生ID}) \text{ Advisor}$$

### 3. 投影操作 (PROJECT):

- 投影出导师的名字和学生的姓名。

$$\pi(\text{导师名}, \text{姓名})(\sigma(\text{年龄} > 20)(\text{Student}) \bowtie \text{Advisor})$$

- 结果：

导师名	姓名
王教授	李四
陈教授	赵六

### 3. 找出所有学生的课程成绩及其对应导师，且只保留与所有课程都有关的学生

运算步骤：

#### 1. 连接操作 (JOIN):

- 首先将 选课 和 课程 进行连接，以便得到学生选修的课程名称。

$$\text{Enrolment} \bowtie (\text{课程ID}=\text{课程ID}) \text{ Course}$$

#### 2. 连接操作 (JOIN):

- 再将上一步结果与 学生 关系连接，以便找到每个学生的姓名和成绩。

$$(\text{Enrolment} \bowtie \text{Course}) \bowtie (\text{学生ID}=\text{学生ID}) \text{ Student}$$

#### 3. 除法操作 (DIVISION):

- 现在我们需要找出那些选修了所有课程的学生。将包含所有学生的 (学生ID, 课程ID) 投影部分除以包含所有课程的 课程ID。

$$\pi(\text{学生ID}, \text{课程ID})(\text{Enrolment}) \div \pi(\text{课程ID})(\text{Course})$$

#### 4. 连接操作 (JOIN):

- 将除法结果中的学生ID与之前的详细连接结果结合，以找到相关详细信息。

$$(\pi(\text{学生ID}, \text{课程ID})(\text{Enrolment}) \div \pi(\text{课程ID})(\text{Course})) \bowtie (\text{学生ID}=\text{学生ID}) \\ ((\text{Enrolment} \bowtie \text{Course}) \bowtie \text{Student})$$

## 5. 结果：

学生ID	姓名	课程ID	课程名	成绩
1	张三	101	数学	85
1	张三	102	物理	90
4	赵六	101	数学	95
4	赵六	102	物理	82

## 4. 统计每个导师所指导学生的平均成绩

运算步骤：

### 1. 连接操作 (JOIN)：

- 将 **导师** 和 **选课** 连接，以获得每个学生及其课程成绩，以及导师信息。

$$\text{Advisor} \bowtie (\text{学生ID}=\text{学生ID}) \text{ Enrolment}$$

### 2. 聚合操作 (AGGREGATE)：

- 使用聚合操作计算每个导师所指导学生的**平均成绩**。

$$\gamma(\text{导师名}, \text{AVG}(\text{成绩}))(\text{Advisor} \bowtie \text{Enrolment})$$

## 3. 结果：

导师名	平均成绩
陈教授	88
王教授	78
李教授	88

## 总结

通过以上的例子，我们用到了几乎所有关系代数的主要操作：

- 选择 (SELECT)** 用于从关系中筛选符合条件的元组。
- 投影 (PROJECT)** 用于从关系中选择特定属性。
- 并集、交集、差集** 用于处理集合之间的数据操作。
- 笛卡尔积 (CARTESIAN PRODUCT)** 和 **连接 (JOIN)** 用于合并不同关系中的相关数据。

- **除法 (DIVISION)** 用于找出满足所有子集条件的数据。
- **重命名 (RENAME)** 可以用于消除属性名称上的歧义。
- **聚合 (AGGREGATE)** 用于计算统计信息，如平均值、总和等。

这些运算结合起来，帮助我们解决复杂的数据查询和分析问题。

我将综合使用 SQL 的更多语法特性来回答问题，尽量涵盖更多的功能，包括子查询、集合操作（UNION、INTERSECT、EXCEPT）、聚合函数、HAVING、DISTINCT、视图（VIEW）、嵌套查询、IN、EXISTS、ANY、ALL、ORDER BY 等。

## 例题描述

继续使用前面的数据库表：

### 1. 学生 (Students)

学生ID	姓名	年龄
1	张三	20
2	李四	21
3	王五	19
4	赵六	22

### 2. 课程 (Courses)

课程ID	课程名
101	数学
102	物理
103	化学
104	生物

### 3. 选课 (Enrolments)

学生ID	课程ID	成绩
1	101	85
1	102	90
2	101	78
3	103	88
4	102	82
4	104	91

### 4. 导师 (Advisors)

导师ID	导师名	学生ID
201	陈教授	1
202	王教授	2
203	李教授	3
204	张教授	4

## 综合问题和解答

### 1. 找出所有选修过 "数学" 的学生的姓名，并按年龄降序排序

运算步骤：

#### 1. 选择 (SELECT)、连接 (JOIN) 和 排序 (ORDER BY)：

- 使用连接操作查询选修数学的学生，并按年龄降序排序。

```
SELECT DISTINCT s.姓名, s.年龄
FROM Students s
JOIN Enrolments e ON s.学生ID = e.学生ID
JOIN Courses c ON e.课程ID = c.课程ID
WHERE c.课程名 = '数学'
ORDER BY s.年龄 DESC;
```

- 结果：

姓名	年龄
李四	21
张三	20

### 2. 查找那些选修了所有课程的学生（使用除法操作的变通方法）

运算步骤：

#### 1. 使用 NOT EXISTS 和 EXCEPT 来实现除法操作：

```
SELECT DISTINCT e.学生ID, s.姓名
FROM Enrolments e
JOIN Students s ON e.学生ID = s.学生ID
WHERE NOT EXISTS (
    (SELECT 课程ID FROM Courses)
    EXCEPT
    (SELECT 课程ID FROM Enrolments WHERE Enrolments.学生ID = e.学生ID)
);
```

- 结果：没有学生选修了所有课程。

### 3. 查找哪些导师指导的学生选修了 "化学" 并且得分超过 80 分

运算步骤：

## 1. 选择、连接和子查询：

- 查找学生选修化学且成绩超过 80 分的记录，再找到他们对应的导师。

```
SELECT DISTINCT a.导师名, s.姓名
FROM Advisors a
JOIN Students s ON a.学生ID = s.学生ID
JOIN Enrolments e ON s.学生ID = e.学生ID
JOIN Courses c ON e.课程ID = c.课程ID
WHERE c.课程名 = '化学' AND e.成绩 > 80;
```

- 结果：

导师名	姓名
李教授	王五

## 4. 创建一个视图，包含所有导师及其学生的选课成绩

运算步骤：

### 1. 创建视图 (VIEW)：

- 使用 `CREATE VIEW` 创建一个视图，包含导师及其学生的详细选课信息。

```
CREATE VIEW 导师学生选课视图 AS
SELECT a.导师名, s.姓名 AS 学生姓名, c.课程名, e.成绩
FROM Advisors a
JOIN Students s ON a.学生ID = s.学生ID
JOIN Enrolments e ON s.学生ID = e.学生ID
JOIN Courses c ON e.课程ID = c.课程ID;
```

### 2. 查询视图：

- 从刚创建的视图中查询信息。

```
SELECT * FROM 导师学生选课视图 WHERE 成绩 > 80;
```

- 结果：

导师名	学生姓名	课程名	成绩
陈教授	张三	数学	85
陈教授	张三	物理	90
李教授	王五	化学	88

张教授	赵六	生物	91
张教授	赵六	物理	82

## 5. 使用聚合函数查找每个学生的总成绩和平均成绩，按总成绩降序排序

运算步骤：

### 1. 聚合 (AGGREGATE) 和 分组 (GROUP BY)：

- 对每个学生进行成绩的总和和平均值计算，并按总成绩降序排序。

```
SELECT e.学生ID, s.姓名, SUM(e.成绩) AS 总成绩, AVG(e.成绩) AS 平均成绩
FROM Enrolments e
JOIN Students s ON e.学生ID = s.学生ID
GROUP BY e.学生ID, s.姓名
ORDER BY 总成绩 DESC;
```

- 结果：

学生ID	姓名	总成绩	平均成绩
4	赵六	173	86.5
1	张三	175	87.5
3	王五	88	88.0
2	李四	78	78.0

## 6. 使用 EXISTS 查找指导至少一名学生且该学生选修了 "生物" 的导师

运算步骤：

### 1. 使用 EXISTS：

- 查找指导至少一名选修了 "生物" 课程的导师。

```
SELECT DISTINCT a.导师名
FROM Advisors a
WHERE EXISTS (
    SELECT 1
    FROM Enrolments e
    JOIN Courses c ON e.课程ID = c.课程ID
    WHERE a.学生ID = e.学生ID AND c.课程名 = '生物'
);
```

- 结果：

导师名  
----  
张教授

## 7. 查找年龄大于 20 岁并且成绩超过平均成绩的学生

运算步骤：

### 1. 使用子查询和比较：

- 查找年龄大于 20 岁的学生，且他们的成绩高于全体学生的平均成绩。

```
SELECT s.学生ID, s.姓名, e.成绩
FROM Students s
JOIN Enrolments e ON s.学生ID = e.学生ID
WHERE s.年龄 > 20 AND e.成绩 > (
    SELECT AVG(成绩)
    FROM Enrolments
);
```

- 结果：

学生ID	姓名	成绩
1	张三	85
1	张三	90
4	赵六	82
4	赵六	91

## 8. 使用集合操作查找选修了 "数学" 或 "物理" 的学生，但未选修 "化学"

运算步骤：

### 1. 使用集合操作 (UNION 和 EXCEPT)：

```
(SELECT 学生ID FROM Enrolments e JOIN Courses c ON e.课程ID = c.课程ID
WHERE c.课程名 = '数学'
UNION
SELECT 学生ID FROM Enrolments e JOIN Courses c ON e.课程ID = c.课程ID
WHERE c.课程名 = '物理')
EXCEPT
(SELECT 学生ID FROM Enrolments e JOIN Courses c ON e.课程ID = c.课程ID
WHERE c.课程名 = '化学');
```

- 结果：

学生ID

----

1

2

4

## 总结

在以上问题中，我综合使用了多种 SQL 语法特性，包括：

- **选择 (SELECT)、投影 (PROJECT)、连接 (JOIN)。**
- **子查询 (Subquery)、EXISTS 和 NOT EXISTS、IN、ANY 和 ALL。**
- **除法操作的变通方法 (NOT EXISTS 和 EXCEPT)。**
- **集合操作 (UNION、INTERSECT、EXCEPT)。**
- **聚合函数 (SUM, AVG, COUNT) 及其与 GROUP BY 和 HAVING 的结合。**
- **视图 (VIEW) 的创建和使用。**
- **排序 (ORDER BY) 和 DISTINCT 来消除重复数据。**

## PL/pgSQL

**PL/pgSQL** 是 PostgreSQL 中的过程化扩展语言，结合了 SQL 与过程化编程的特性，能够让用户在数据库中实现复杂的业务逻辑控制。它支持编写函数、存储过程、触发器等，提供了传统编程语言的控制结构，并与 SQL 紧密集成。

### 1. PL/pgSQL 的用途

PL/pgSQL 可以用于以下场景：

- **存储过程与函数**：编写复杂的数据库操作逻辑，存储于数据库中，能够通过函数调用进行复用。
- **触发器**：在特定数据库事件（如插入、更新、删除）发生时，自动触发执行自定义逻辑。
- **数据验证**：通过触发器或函数进行复杂的数据验证，确保数据一致性和完整性。
- **业务逻辑嵌入数据库**：将应用程序的业务逻辑直接嵌入到数据库中，减少应用程序与数据库之间的通信，提升性能。

### 2. PL/pgSQL 的语法结构

PL/pgSQL 提供了类似传统编程语言的语法结构，包括变量声明、条件判断、循环控制、错误处理等。以下是各个方面的详细介绍。

#### 2.1 函数和存储过程

- **存储过程 (Stored Procedure)** 是存储在数据库中的程序，它们可以直接调用并执行数据库操作。
- **函数 (Function)** 则类似于存储过程，但可以返回值（如单个值或多个元组）。

函数的定义语法：

```
CREATE OR REPLACE FUNCTION funcName(param1 type, param2 type, ...)
RETURNS returnType AS $$  
DECLARE
    variable declarations;
BEGIN
    -- 代码逻辑
    RETURN some_value;
END;
$$ LANGUAGE plpgsql;
```

- 参数可以有三种模式：
  - **IN**：输入参数，只能读取。
  - **OUT**：输出参数，只能写入。
  - **INOUT**：既能读取，也能写入。

#### 2.2 示例：银行取款操作

以下是一个简单的银行取款函数示例，模拟从银行账户中扣款：

```
CREATE FUNCTION withdraw(acctNum text, amount integer) RETURNS text AS $$  
DECLARE  
    bal integer;  
BEGIN  
    SELECT balance INTO bal FROM Accounts WHERE acctNo = acctNum;  
    IF (bal < amount) THEN  
        RETURN 'Insufficient Funds';  
    ELSE  
        UPDATE Accounts SET balance = balance - amount WHERE acctNo = acctNum;  
        SELECT balance INTO bal FROM Accounts WHERE acctNo = acctNum;  
        RETURN 'New Balance: ' || bal;  
    END IF;  
END;  
$$ LANGUAGE plpgsql;
```

- 逻辑解释:

- 首先查询账户余额 (SELECT balance INTO bal), 如果余额不足 (IF bal < amount), 则返回 "余额不足"。
- 否则, 扣减金额并更新账户余额。

### 3. 控制结构

PL/pgSQL 提供了多种控制结构, 类似于传统的编程语言, 用于实现复杂逻辑。

#### 3.1 条件控制

- IF ... THEN ... ELSE:

```
IF condition THEN  
    -- 语句块  
ELSE  
    -- 其他情况  
END IF;
```

- IF ... ELSIF ... ELSE:

```
IF condition1 THEN  
    -- 语句块1  
ELSIF condition2 THEN  
    -- 语句块2  
ELSE  
    -- 其他情况  
END IF;
```

#### 3.2 循环控制

- **基本循环:** LOOP ... END LOOP;

```
LOOP  
    -- 语句块  
    EXIT WHEN condition;  
END LOOP;
```

- **计数循环:** FOR i IN 1..10 LOOP ... END LOOP;

```
FOR i IN 1..10 LOOP  
    -- i 会依次取值 1 到 10  
END LOOP;
```

## 4. 游标 (Cursor)

游标是一个指针，用于逐行遍历查询结果集。

- **声明游标:**

```
<cursor_name> CURSOR FOR <query>;
```

- **操作游标:**

- **打开游标:** OPEN <cursor\_name>;
- **提取数据:** FETCH <cursor\_name> INTO <variable>;
- **关闭游标:** CLOSE <cursor\_name>;

示例：计算员工的总工资：

```
CREATE FUNCTION totalSalary() RETURNS real AS $$  
DECLARE  
    employee RECORD;  
    totalSalary REAL := 0;  
BEGIN  
    FOR employee IN SELECT * FROM Employees LOOP  
        totalSalary := totalSalary + employee.salary;  
    END LOOP;  
    RETURN totalSalary;  
END;  
$$ LANGUAGE plpgsql;
```

- **解释:** 该函数通过游标逐行遍历 Employees 表中的每个员工，累加其工资以计算总和。

## 5. 触发器 (Trigger)

触发器是数据库中的特殊程序，在特定的事件（如 `INSERT`、`UPDATE` 或 `DELETE`）发生时自动执行。

- **触发器的定义：**

```
CREATE TRIGGER TriggerName
AFTER/BEFORE Event ON TableName
FOR EACH ROW/STATEMENT
EXECUTE PROCEDURE FunctionName(args...);
```

- **触发器的作用：**例如在员工表中插入新员工时，自动更新部门的工资总和。

**触发器函数示例：**当新员工加入时更新部门工资总和：

```
CREATE FUNCTION totalSalary1() RETURNS trigger AS $$ 
BEGIN
    IF (NEW.dept IS NOT NULL) THEN
        UPDATE Department
        SET totSal = totSal + NEW.salary
        WHERE Department.id = NEW.dept;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER TotalSalary1
AFTER INSERT ON Employees
FOR EACH ROW EXECUTE PROCEDURE totalSalary1();
```

- **解释：**在 `Employees` 表中插入新员工后，触发器自动执行 `totalSalary1()` 函数，更新对应部门的工资总和。

## 6. 错误处理 (Exceptions)

PL/pgSQL 支持通过 `EXCEPTION` 处理运行过程中发生的异常。

- **示例：**处理除零异常：

```
BEGIN
    -- 代码块
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Caught division_by_zero';
END;
```

- 如果代码块中发生了除零错误，将会捕获并输出 `Caught division_by_zero` 的通知。

## 7. 用户自定义函数 (UDF)

PL/pgSQL 支持用户创建自定义函数（UDF），可以用于查询、复杂计算等。

- **类型：**

- **SQL 函数**：直接在 SQL 中调用的函数。
- **过程化语言函数**：如 PL/pgSQL 函数，支持复杂逻辑控制。

## 数据库规范化形式与实例详解

数据库规范化 (Normalization) 是数据库设计中的一种方法，用于消除冗余数据、提高数据一致性，并避免数据更新时的异常。通过逐步应用不同的规范化标准（范式），可以将数据库关系划分为结构更简单、依赖更明确的关系。以下将详细介绍从\*\*第一范式 (1NF) 到 Boyce-Codd 范式 (BCNF)\*\* 的定义与具体实例。

### 正规化范式 (Normal Forms)

1. **1NF (第一范式)**: 所有属性的值必须是原子的，即不可再分。
2. **2NF (第二范式)**: 非主属性不能对任何候选键部分依赖。
3. **3NF (第三范式)**: 对于所有非平凡的函数依赖 ( $X \rightarrow A$ )，要么 ( $X$ ) 是超键，要么 ( $A$ ) 是主属性 (没有传递依赖)。
4. **BCNF (Boyce-Codd 正规形)**: 对于所有非平凡的函数依赖 ( $X \rightarrow A$ )，( $X$ ) 必须是超键。

#### 1. 第一范式 (1NF)

- **定义**: 关系中的每个属性值必须是原子的 (Atomic)，即每个属性的值都是不可分割的，不能包含集合、列表或嵌套关系。
- **实例**: 假设有一个关系 **CRS\_PREF**，用于表示教授对不同课程的偏好：

<b>Prof</b>	<b>Course</b>	<b>Fac_Dept</b>	<b>Crs_Dept</b>
Smith	353	Comp Sci	Comp Sci
Smith	379	Comp Sci	Comp Sci
Turner	456	Chemistry	Mathematics

在这个表中，**每个单元格的值都是原子的**，没有多值或复杂类型，这意味着它满足 1NF。然而，这种形式存在一些问题：

- **数据重复**: 例如教授 Smith 和课程 353、379 的关联会导致部门信息多次重复。
- **插入异常**: 如果我们想插入一个新教授，但他还没有教授任何课程，那么我们无法插入记录，因为缺少课程信息。
- **删除异常**: 删除某个教授的最后一门课程可能导致该教授的部门信息也被删除。

#### 2. 第二范式 (2NF)

- **定义**: 一个关系模式 R 处于 2NF (Second Normal Form)，当且仅当 R 的每个非主属性完全依赖于 R 的**每个候选键 (Candidate Key)**。这里，依赖的左边是候选键，右边是非主属性。
- **部分函数依赖 (Partial Functional Dependency)** : 如果存在一个非主属性仅依赖于候选键的某个子集，则称其存在部分函数依赖。这里，依赖的左边是候选键的子集，右边是非主属性。这种情况下，该关系不符合 2NF，需要分解。
- **实例**: 在上面的 **CRS\_PREF** 表中，**Fac\_Dept** 只依赖于 **Prof**，而与 **Course** 无关。这意味着存在**部分依赖**，导致数据的重复存储。为了解决这个问题，我们可以将 **CRS\_PREF** 分解为以下三个关系：

1. **COURSE\_PREF** (教授与课程) :

Prof	Course
Smith	353
Smith	379
Turner	456

## 2. COURSE (课程与部门) :

Course	Dept
353	Comp Sci
379	Comp Sci
456	Mathematics

## 3. FACULTY (教授与部门) :

Prof	Dept
Smith	Comp Sci
Turner	Chemistry

通过这样的分解，消除了部分依赖 (Partial Dependency)，关系模式进入了 2NF，减少了数据冗余。

## 3. 第三范式 (3NF)

- 定义:** 一个关系模式 R 处于 3NF (Third Normal Form)，当且仅当对于每一个非平凡的函数依赖 (Functional Dependency)  $X \rightarrow A$ ，要么 X 是超键 (Superkey)，要么 A 是候选键的属性。
  - 传递函数依赖 (Transitive Dependency) :** 如果一个属性通过另一个属性间接依赖于候选键，则存在传递依赖。这会导致数据冗余和更新异常，需要消除。
- 实例:** 考虑以下 TEACHES 表:

Course	Prof	Room	Room_Cap	Enrol_Lmt
353	Smith	A532	45	40
456	Turner	B278	50	45

在这个关系中，Room\_Cap 是通过 Room 间接依赖于 Course，这是一种传递依赖。为了消除这种依赖，我们可以将 TEACHES 表分解为两个子关系：

### 1. COURSE\_DETAILS (课程、教授和房间) :

Course	Prof	Room
353	Smith	A532
456	Turner	B278

## 2. ROOM\_DETAILS (房间、容量和注册限制) :

Room	Room_Cap	Enrol_Lmt
A532	45	40
B278	50	45

通过分解，我们消除了传递依赖，使得每个非主属性直接依赖于候选键，从而达到了 3NF。

## 4. Boyce-Codd 范式 (BCNF)

- **定义：**一个关系模式处于 BCNF (Boyce-Codd Normal Form)，当且仅当对于每一个非平凡的函数依赖  $X \rightarrow A$ ,  $X$  必须是超键 (Superkey)。
  - BCNF 是 3NF (Third Normal Form) 的强化版本，进一步消除了由非超键决定的属性带来的冗余。
- **实例：**假设有一个关系 BOOKING：

Title	Theater	City
MovieA	Cineplex	NYC
MovieB	Cineplex	NYC

在这个表中， $\text{Theater} \rightarrow \text{City}$ ，但  $\text{Theater}$  不是超键，因此该关系不符合 BCNF。为了解决这个问题，我们可以将其分解为两个关系：

### 1. THEATER\_DETAILS (影院和城市) :

Theater	City
Cineplex	NYC

### 2. MOVIE\_SHOWING (电影和影院) :

Title	Theater
MovieA	Cineplex
MovieB	Cineplex

通过这样的分解，所有属性都由超键唯一决定，从而消除了冗余，使关系达到 BCNF。

## 5. 总结与判定示例

- 判断一个关系模式 R 最高符合的范式 (从高到低判断) :

### 1. BCNF 检查:

- 判断关系中的每一个非平凡函数依赖  $X \rightarrow A$ ，是否  $X$  是超键 (Superkey)。如果存在任意非平凡依赖的左边  $X$  不是超键，则该关系不符合 BCNF，继续检查是否符合 3NF。

### 2. 3NF 检查:

- 判断关系中的每一个非平凡函数依赖  $X \rightarrow A$ ，是否满足以下条件之一：

- $X$  是超键。
- $A$  是候选键的属性。
- 如果存在传递依赖 (Transitive Dependency)，即非主属性通过另一个非主属性间接依赖于主键，则关系不符合 3NF，继续检查是否符合 2NF。

### 3. 2NF 检查：

- 首先确认关系模式符合 1NF。
- 检查每一个非主属性是否完全依赖于整个候选键。如果存在部分依赖 (Partial Dependency)，即非主属性只依赖于候选键的某个子集，则关系不符合 2NF，继续判断是否符合 1NF。

### 4. 1NF 检查：

- 检查关系中的每个属性值是否都是原子的，即不可再分的基本类型。如果某个属性值包含集合、列表或嵌套关系，则不符合 1NF。

## 实用性提示：

- 从高到低逐步判断的方式更有效，因为一旦关系不符合某个高范式，它必然不符合更高的范式。因此从 BCNF 开始逐步降低，可以更快速确定最高符合的范式。

**例题：**假设有一个关系 STUDENT\_COURSE：

StudentID	CourseID	Instructor	Dept
1	C1	Prof. A	CS
2	C1	Prof. A	CS
3	C2	Prof. B	Math

- **BCNF 检查：**
  - 依赖  $\text{Instructor} \rightarrow \text{Dept}$  中， $\text{Instructor}$  不是超键，因此不符合 BCNF。
- **3NF 检查：**
  - 依赖  $\text{Instructor} \rightarrow \text{Dept}$  是传递依赖，因为  $\text{Dept}$  通过  $\text{Instructor}$  间接依赖于 ( $\text{StudentID}$ ,  $\text{CourseID}$ )，因此不符合 3NF。
- **2NF 检查：**
  - $\text{Dept}$  对  $\text{Instructor}$  存在部分依赖，因为它不依赖于整个候选键 ( $\text{StudentID}$ ,  $\text{CourseID}$ )，因此不符合 2NF。
- **1NF 检查：**
  - 该关系符合 1NF，因为每个属性值都是原子的。

通过分析，我们可以得出该关系模式只符合 **1NF**。

- **判断一个关系模式 R 最高符合的范式：**
  - 检查 R 是否符合 1NF：每个属性值是否都是原子的。
  - 检查 R 是否符合 2NF：是否存在部分函数依赖，如果有则不符合 2NF。
  - 检查 R 是否符合 3NF：是否存在传递依赖，如果有则不符合 3NF。
  - 检查 R 是否符合 BCNF：是否所有非平凡的函数依赖的左边都是超键，如果不是，则不符合 BCNF。

## 分解 (Decomposition)

- **分解的属性保留条件**: 在分解中, 所有属性必须在分解后的关系中保留。
- **分解的两个重要性质**:
  1. **依赖保持性**: 所有的原始函数依赖都能在分解后的关系中表达。
  2. **无损连接性质**: 在自然连接操作后, 分解后的关系能重建原始关系。

## 依赖保持 (Dependency Preserving)

- 分解 ( $D = \{R_1, \dots, R_n\}$ ) 被称为依赖保持的, 如果所有投影后的函数依赖的闭包等于原始函数依赖的闭包, 即  $((F_1 \cup \dots \cup F_n)^+ = F^+)$ 。

## 无损连接性质 (Lossless Join Property)

- 若分解 ( $D = \{R_1, \dots, R_m\}$ ) 满足无损连接性质, 则对于任意满足函数依赖的关系实例, 所有分解的自然连接的结果应与原始关系相同。
- 检查方法: 分解为 ( $R_1, R_2$ ) 的无损连接性质成立, 当且仅当交集 ( $R_1 \cap R_2$ ) 构成 ( $R_1$ ) 或 ( $R_2$ ) 的一个超键。

## BCNF 和 3NF 的分解算法

- **BCNF 分解算法 (TO\_BCNF)**:
  - 找到违反 BCNF 的函数依赖并分解关系, 直到所有关系都满足 BCNF。
  - 该分解确保无损连接, 但可能不保持依赖。
- **3NF 分解算法**:
  - 通过最小覆盖得到函数依赖的最简形式。
  - 创建包含候选键的关系以确保依赖保持, 并进行无损连接。
  - 3NF 分解总是能够找到依赖保持且无损的分解, 但可能保留一定的冗余。

## 最小覆盖 (Minimal Cover)

- **最小覆盖**: 函数依赖的最小集合, 具有相同的闭包且去除了冗余。
- 计算最小覆盖的步骤:
  1. **右侧简化**: 将右侧多个属性拆分为多个单属性的函数依赖。
  2. **左侧简化**: 去除左侧的冗余属性。
  3. **冗余去除**: 删除冗余的函数依赖。

## Test Lossless Join property

This previous test works on **binary** decompositions, below is the general solution to testing lossless join property

Algorithm TEST\_LJ:

1. Create a **matrix S**, each element  $s_{ij} \in S$  corresponds the relation  $R_i$  and the attribute  $A_j$ , such that:  $s_{ji} = a$  if  $A_i \in R_j$ , otherwise  $s_{ji} = b$ .
2. Repeat the following process until (1) S has no change OR (2) one row is made up entirely of "a" symbols.
  - i. For each  $X \rightarrow Y$ , choose the rows where the elements corresponding to X take the value a.
  - ii. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y.

Verdict: Decomposition is *lossless* if one row is entirely made up by "a" values.

27

## Testing lossless join property(cont)

**Example 1:**

$R = (A,B,C,D)$ ,  
 $F = (A \rightarrow B, A \rightarrow C, C \rightarrow D)$ .  
 Let  $R_1 = (A,B,C)$ ,  $R_2 = (C,D)$ .

	A	B	C	D
$R_1$	a	a	a	b
$R_2$	b	b	a	a

Note: rows 1 and 2 of S agree on {C}, which is the left-hand side of  $C \rightarrow D$ . Therefore, change the D value on rows 1 to a, matching the value from row 2.

**CHEAT SHEET: Algorithm TEST\_LJ**

1. Create a matrix S, each element  $s_{ij} \in S$  corresponds the relation  $R_i$  and the attribute  $A_j$ , such that:  $s_{ji} = a$  if  $A_i \in R_j$ , otherwise  $s_{ji} = b$ .
2. Repeat the following process till S has no change or one row is made up entirely of "a" symbols.
  1. For each  $X \rightarrow Y$ , choose the rows where the elements corresponding to X take the value a.
  2. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

28

## Testing lossless join property(cont)

**Example 1:**

$R = (A,B,C,D)$ ,  
 $F = (A \rightarrow B, A \rightarrow C, C \rightarrow D)$ .  
 Let  $R_1 = (A,B,C)$ ,  $R_2 = (C,D)$ .

	A	B	C	D
$R_1$	a	a	a	b a
$R_2$	b	b	a	a

Note: rows 1 and 2 of S agree on {C}, which is the left-hand side of  $C \rightarrow D$ . Therefore, change the D value on rows 1 to a, matching the value from row 2.

Now **row 1 is entirely a's**, so the decomposition is lossless.

29

## Testing lossless join property(cont)

**Example 2:**

$R = (A,B,C,D,E)$ ,  
 $F = \{AB \rightarrow CD, A \rightarrow E, C \rightarrow D\}$ .

Let  $R_1 = (A,B,C)$ ,  
 $R_2 = (B,C,D)$  and  
 $R_3 = (C,D,E)$ .

	A	B	C	D	E
$R_1$	a	a	a	b	b
$R_2$	b	a	a	a	b
$R_3$	b	b	a	a	a

**CHEAT SHEET: Algorithm TEST\_LJ**

1. Create a matrix S, each element  $s_{ij} \in S$  corresponds the relation  $R_i$  and the attribute  $A_j$ , such that:  $s_{ji} = a$  if  $A_i \in R_j$ , otherwise  $s_{ji} = b$ .
2. Repeat the following process till S has no change or one row is made up entirely of "a" symbols.
  1. For each  $X \rightarrow Y$ , choose the rows where the elements corresponding to X take the value a.
  2. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

30

## Testing lossless join property(cont)

**Example 2:**

$R = (A,B,C,D,E)$ ,  
 $F = \{AB \rightarrow CD, A \rightarrow E, C \rightarrow D\}$ .

Let  $R_1 = (A,B,C)$ ,  
 $R_2 = (B,C,D)$  and  
 $R_3 = (C,D,E)$ .

A	B	C	D	E
R <sub>1</sub>	a	a	a	b
R <sub>2</sub>	b	a	a	b
R <sub>3</sub>	b	b	a	a

Not lossless join

31

## Testing lossless join property(cont)

**Example 3:**

$R = (A,B,C,D,E,G)$ ,  
 $F = \{C \rightarrow DE, A \rightarrow B, AB \rightarrow G\}$ .  
 Let  $R_1 = (A,B)$ ,  $R_2 = (C,D,E)$  and  
 $R_3 = (A,C,G)$ .

	A	B	C	D	E	G
$R_1$	a	a	b	b	b	b
$R_2$	b	b	a	a	a	b
$R_3$	a	b	a	b	b	a

**CHEAT SHEET: Algorithm TEST\_LJ**

1. Create a matrix S, each element  $s_{ij} \in S$  corresponds the relation  $R_i$  and the attribute  $A_j$ , such that:  $s_{ji} = a$  if  $A_i \in R_j$ , otherwise  $s_{ji} = b$ .
2. Repeat the following process till S has no change or one row is made up entirely of "a" symbols.
  1. For each  $X \rightarrow Y$ , choose the rows where the elements corresponding to X take the value a.
  2. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

32

## Testing lossless join property(cont)

### Example 3:

$R = (A, B, C, D, E, G)$ ,  
 $F = \{C \rightarrow DE, A \rightarrow B, AB \rightarrow G\}$ .  
Let  $R_1 = (A, B)$ ,  $R_2 = (C, D, E)$  and  
 $R_3 = (A, C, G)$ .

A	B	C	D	E	G
$R_1$	a	a	b	b	b
$R_2$	b	b	a	a	b
$R_3$	a	b	a	b	a

### CHEAT SHEET: Algorithm TEST\_LJ

1. Create a matrix  $S$ , each element  $s_{ij} \in S$  corresponds the relation  $R_i$  and the attribute  $A_j$ , such that:  $s_{ij} = a$  if  $A_j \in R_i$ , otherwise  $s_{ij} = b$ .
2. Repeat the following process till  $S$  has no change or one row is made up entirely of "a" symbols.
  1. For each  $X \rightarrow Y$ , choose the rows where the elements corresponding to  $X$  take the value a.
  2. In those chosen rows (must be at least two rows), the elements corresponding to  $Y$  also take the value a if one of the chosen rows take the value a on  $Y$ .

33

## Testing lossless join property(cont)

### Example 3:

$R = (A, B, C, D, E, G)$ ,  
 $F = \{C \rightarrow DE, A \rightarrow B, AB \rightarrow G\}$ .  
Let  $R_1 = (A, B)$ ,  $R_2 = (C, D, E)$  and  
 $R_3 = (A, C, G)$ .

A	B	C	D	E	G
$R_1$	a	a	b	b	b
$R_2$	b	b	a	a	b
$R_3$	a	b	a	b	a

### CHEAT SHEET: Algorithm TEST\_LJ

1. Create a matrix  $S$ , each element  $s_{ij} \in S$  corresponds the relation  $R_i$  and the attribute  $A_j$ , such that:  $s_{ij} = a$  if  $A_j \in R_i$ , otherwise  $s_{ij} = b$ .
2. Repeat the following process till  $S$  has no change or one row is made up entirely of "a" symbols.
  1. For each  $X \rightarrow Y$ , choose the rows where the elements corresponding to  $X$  take the value a.
  2. In those chosen rows (must be at least two rows), the elements corresponding to  $Y$  also take the value a if one of the chosen rows take the value a on  $Y$ .

34

## Checkpoint

### Previous:

1. The test for lossless join property
2. The dependency preservation property

### Next:

1. The method to decompose to BCNF and 3NF
2. Minimal Cover and Equivalence
3. The method to decompose to 3NF

35

36

## Testing for BCNF

NOTE: We cannot use  $F$  to test relations  $R_i$  (decomposed from  $R$ ) for violation of BCNF. It may not suffice.

Consider  $R(A, B, C, D, E)$  with  $F = \{A \rightarrow B, BC \rightarrow D\}$ .

Suppose  $R$  is decomposed into  $R1 = (A, B)$  and  $R2 = (A, C, D, E)$ .

Neither of the dependencies in  $F$  contains only attributes from  $R_2$ . So  $R_2$  is in BCNF? No,  $AC \rightarrow D$  is in  $F^+$ .

Example above :  $X \rightarrow Y$  violating BCNF is not always in  $F$ . It passing with respect to the projection of  $F$  on  $R_1$ .

37

## Testing Decomposition for BCNF

An alternative BCNF test is sometimes easier than computing every dependency in  $F^+$ . To check if a relation schema  $R_i$  in a decomposition of  $R$  is truly in BCNF, we apply this test:

For each subset  $X$  of  $R_i$ , compute  $X^*$ .

- $X \rightarrow (X^*|_{R_i} - X)$  violates BCNF, if  $X^*|_{R_i} - X \neq \emptyset$  and  $R_i - X^* \neq \emptyset$ .
- This will show if  $R_i$  violates BCNF.

Explanation:

- $X^*|_{R_i} - X = \emptyset$  means each F.D with  $X$  as the left-hand side is trivial;
- $R_i - X^* = \emptyset$  means  $X$  is a superkey of  $R_i$

38

## Lossless Decomposition into BCNF

### Algorithm TO\_BCNF

- D := {R<sub>1</sub>, R<sub>2</sub>, ... R<sub>n</sub>}
- While ( there exists a R<sub>i</sub> ∈ D and R<sub>i</sub> is not in BCNF ) Do
  - 1 . find a X → Y in R<sub>i</sub> that violates BCNF;
  2. replace R<sub>i</sub> in D by ( R<sub>i</sub> – Y ) and (X ∪ Y);

39

## Lossless Decomposition into BCNF

### Example:

Find a BCNF decomposition of the relation scheme below:

**SHIPPING** (Ship , Capacity , Date , Cargo , Value)

F consists of:

Ship → Capacity

{Ship , Date} → Cargo

{Cargo , Capacity} → Value

We know this relation is not in BCNF

### Algorithm TO\_BCNF

D := {R<sub>1</sub>, R<sub>2</sub>, ... R<sub>n</sub>}

While ( there exists a R<sub>i</sub> ∈ D and R<sub>i</sub> is not in BCNF ) Do

- 1 . find a X → Y in R<sub>i</sub> that violates BCNF;
2. replace R<sub>i</sub> in D by ( R<sub>i</sub> – Y ) and (X ∪ Y);

40

## Lossless Decomposition into BCNF (V1)

From Ship → Capacity, we decompose SHIPPING into R<sub>1A</sub> and R<sub>2A</sub>

R<sub>1A</sub>(Ship , Date , Cargo , Value) with Key: {Ship,Date}

A nontrivial FD in F<sup>+</sup> violates BCNF: {Ship , Cargo} → Value

and

R<sub>2A</sub>(Ship , Capacity) with Key: {Ship}

Only one nontrivial FD in F<sup>+</sup>: Ship → Capacity

SHIPPING (Ship , Capacity , Date , Cargo , Value)

F consists of: Ship → Capacity, {Ship , Date}→ Cargo, {Cargo , Capacity}→ Value

41

## Lossless Decomposition into BCNF (V1)

R<sub>1</sub> is not in BCNF so we must decompose it further into R<sub>11A</sub> and R<sub>12A</sub>

R<sub>11A</sub>(Ship , Date , Cargo) with Key: {Ship,Date}

Only one nontrivial FD in F<sup>+</sup> with single attribute on the right side: {Ship , Date} → Cargo and

R<sub>12A</sub>(Ship , Cargo , Value) with Key: {Ship,Cargo}

Only one nontrivial FD in F<sup>+</sup> with single attribute on the right side: {Ship,Cargo} → Value

This is in BCNF, and the decomposition is lossless but not dependency preserving (the FD {Capacity, Cargo} → Value) has been lost.

SHIPPING (Ship , Capacity , Date , Cargo , Value)

F consists of: Ship → Capacity, {Ship , Date}→ Cargo, {Cargo , Capacity}→ Value

42

## Lossless Decomposition into BCNF (V2)

Or we could have chosen {Cargo , Capacity} → Value, which would give us:

R<sub>1B</sub>(Ship , Capacity , Date , Cargo) with Key: {Ship,Date}

A nontrivial FD in F<sup>+</sup> violates BCNF: Ship → Capacity

and

R<sub>2B</sub>(Cargo , Capacity , Value) with Key: {Cargo,Capacity}

Only one nontrivial FD in F<sup>+</sup> with single attribute on the right side: {Cargo , Capacity} → Value

Once again, R<sub>1B</sub> is not in BCNF so we must decompose it further...

SHIPPING (Ship , Capacity , Date , Cargo , Value)

F consists of: Ship → Capacity, {Ship , Date}→ Cargo, {Cargo , Capacity}→ Value

43

## Lossless Decomposition into BCNF (V2)

R<sub>1</sub> is not in BCNF so we must decompose it further into R<sub>11B</sub> and R<sub>12B</sub>

R<sub>11B</sub>(Ship , Date , Cargo) with Key: {Ship,Date}

Only one nontrivial FD in F<sup>+</sup> with single attribute on the right side: {Ship , Date} → Cargo and

R<sub>12B</sub>(Ship , Capacity) with Key: {Ship}

Only one nontrivial FD in F<sup>+</sup>: Ship → Capacity

This is in BCNF, and the decomposition is both lossless and dependency preserving.

SHIPPING (Ship , Capacity , Date , Cargo , Value)

F consists of: Ship → Capacity, {Ship , Date}→ Cargo, {Cargo , Capacity}→ Value

44

## Lossless Decomposition into BCNF

With this algorithm from the previous slide...

We get a decomposition  $D$  of  $R$  that does the following:

- May **not** preserves dependencies
- Has the lossless join property
- Is such that each resulting relation schema in the decomposition is in BCNF

45

## Lossless decomposition into BCNF

Review: Algorithm TO\_BCNF

$D := \{R_1, R_2, \dots, R_n\}$

**While**  $\exists$  a  $R_i \in D$  and  $R_i$  is not in BCNF **Do**

{ find a  $X \rightarrow Y$  in  $R_i$  that violates BCNF; replace  $R_i$  in  $D$  by  $(R_i - Y)$  and  $(X \cup Y)$ ; }

Since a  $X \rightarrow Y$  violating BCNF is not always in  $F$ , the main difficulty is to verify if  $R_i$  is in BCNF;

46

## Computing a Minimal Cover (Step 1)

**Step 1: Reduce Right:** For each FD  $X \rightarrow Y \in F$  where  $Y = \{A_1, A_2, \dots, A_k\}$ , we use all  $X \rightarrow \{A_i\}$  (for  $1 \leq i \leq k$ ) to replace  $X \rightarrow Y$ .

Practice:

$$\begin{aligned} R &= (A, B, C, D, E, G) \\ F &= \{A \rightarrow BCD, B \rightarrow CDE, AC \rightarrow E\} \end{aligned}$$

At the end of step 1 we have :  $F' = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, B \rightarrow C, B \rightarrow D, B \rightarrow E, AC \rightarrow E\}$

58

## Computing a Minimal Cover (Step 2)

**Step 2: Reduce Left:** For each  $X \rightarrow \{A\} \in F$  where  $X = \{A_i : 1 \leq i \leq k\}$ , do the following. For  $i = 1$  to  $k$ , replace  $X$  with  $X - \{A_i\}$  if  $A_i \notin (X - \{A_i\})^+$ .

From Step 1, we had:  $F' = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, B \rightarrow C, B \rightarrow D, B \rightarrow E, AC \rightarrow E\}$

$AC \rightarrow E$   
 $C^+ = \{C\}$ ; thus  $C \rightarrow E$  is not inferred by  $F'$ .  
Hence,  $AC \rightarrow E$  cannot be replaced by  $C \rightarrow E$ .  
 $A^+ = \{A, B, C, D, E\}$ ; thus,  $A \rightarrow E$  is inferred by  $F'$ .  
Hence,  $AC \rightarrow E$  can be replaced by  $A \rightarrow E$ .  
We now have  $F'' = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, B \rightarrow C, B \rightarrow D, B \rightarrow E\}$

59

## Computing a Minimal Cover (Step 3)

**Step 3: Reduce\_redundancy:** For each FD  $X \rightarrow \{A\} \in F$ , remove it from  $F$  if:  $A \in X^+$  with respect to  $F - \{X \rightarrow \{A\}\}$ .

From Step 2, we had:  $F''' = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, B \rightarrow C, B \rightarrow D, B \rightarrow E\}$

$A + |_{F''' - \{A \rightarrow B\}} = \{A, C, D, E\}$ ; thus  $A \rightarrow B$  is not inferred by  $F''' - \{A \rightarrow B\}$ .

That is,  $A \rightarrow B$  is not redundant.

$A + |_{F''' - \{A \rightarrow C\}} = \{A, B, C, D, E\}$ ; thus,  $A \rightarrow C$  is redundant.

Thus, we can remove  $A \rightarrow C$  from  $F'''$  to obtain  $F''''$ .

We find that we can remove  $A \rightarrow D$  and  $A \rightarrow E$  but not the others.

Thus,  $F_{min} = \{A \rightarrow B, B \rightarrow C, B \rightarrow D, B \rightarrow E\}$ .

60

## 3NF Decomposition Algorithm

Algorithm 3NF decomposition

1. Find a minimal cover  $G$  for  $F$ .
2. For each left-hand-side  $X$  of a functional dependency that appears in  $G$ , create a relation schema in  $D$  with attributes  $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_d\}\}$ , where  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_d$  are the only dependencies in  $G$  with  $X$  as left-hand-side ( $X$  is the key to this relation).
3. If none of the relation schemas in  $D$  contains a key of  $R$ , then create one more relation schema in  $D$  that contains attributes that form a key of  $R$ .
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation  $R$  is considered redundant if  $R$  is a projection of another relation  $S$  in the schema; alternately,  $R$  is subsumed by  $S$ .

62

## 3NF Decomposition Algorithm

With this algorithm from the previous slide...

We get a decomposition  $D$  of  $R$  that does the following:

- Preserves dependencies
- Has the nonadditive (lossless) join property
- Is such that each resulting relation schema in the decomposition is in 3NF

63

## 3NF Decomposition Algorithm

### Example ONE:

$$R = (A, B, C, D, E, G)$$

$$F_{\min} = \{A \rightarrow B, B \rightarrow C, B \rightarrow D, B \rightarrow E\}.$$

Candidate key:  $(A, G)$

$$R_1 = (A, B), R_2 = (B, C, D, E)$$

$$R_3 = (A, G)$$

64

## 3NF Decomposition Algorithm

### Example TWO:

Following from the *SHIPPING* relation. The functional dependencies already form a canonical cover.

- From  $Ship \rightarrow Capacity$ , derive  $R_1(Ship, Capacity)$ ,
- From  $\{Ship, Date\} \rightarrow Cargo$ , derive  $R_2(Ship, Date, Cargo)$ ,
- From  $\{Capacity, Cargo\} \rightarrow Value$ , derive  $R_3(Capacity, Cargo, Value)$ .
- There are no attributes not yet included and the original key  $\{Ship, Date\}$  is included in  $R_2$ .

*SHIPPING* (*Ship*, *Capacity*, *Date*, *Cargo*, *Value*)

$F$  consists of:  $Ship \rightarrow Capacity$ ,  $\{Ship, Date\} \rightarrow Cargo$ ,  $\{Cargo, Capacity\} \rightarrow Value$

65

## 3NF Decomposition Algorithm

**Example THREE:** Apply the algorithm to the LOTS example given earlier.

One possible minimal cover is

$$\begin{aligned} & \{ Property\_Id \rightarrow Lot\_No, \\ & \quad Property\_Id \rightarrow Area, \{City, Lot\_No\} \rightarrow Property\_Id, \\ & \quad Area \rightarrow Price, Area \rightarrow City, City \rightarrow Tax\_Rate \}. \end{aligned}$$

This gives the decomposition:

$$\begin{aligned} & R_1(Property\_Id, Lot\_No, Area) \\ & R_2(City, Lot\_No, Property\_Id) \\ & R_3(Area, Price, City) \\ & R_4(City, Tax\_Rate) \end{aligned}$$

66

# 函数依赖

## 1. 定义

- 函数依赖描述了关系中不同属性之间的相互关系。
- 如果关系中任意两个元组在属性集  $\alpha$  上的值相同，那么在属性集  $\beta$  上的值也相同，则称  $\alpha$  函数决定  $\beta$ ，记作  $\alpha \rightarrow \beta$ 。
- 示例：
  - $ID \rightarrow Name$  表示在所有元组中，若 ID 相同，则 Name 也相同。

## 2. 作用与意义

- 函数依赖用于：
  - 描述属性之间的语义关系，例如主键和其他属性之间的关系。
  - 作为数据库设计的约束，确保数据的一致性和完整性。
  - 消除冗余，通过合理的分解将关系划分成符合规范化形式的子关系，减少数据重复。

## 3. 函数依赖的种类

- 完全函数依赖：若  $\alpha \rightarrow \beta$  且  $\beta$  不能仅由  $\alpha$  的子集唯一确定，则称  $\beta$  完全函数依赖于  $\alpha$ 。
- 部分函数依赖：如果  $\beta$  仅能由  $\alpha$  的某个子集唯一确定，则称  $\beta$  对  $\alpha$  部分依赖。
- 传递函数依赖：如果  $\alpha \rightarrow \beta$  且  $\beta \rightarrow \gamma$ ，则可得出  $\alpha \rightarrow \gamma$ 。这种情况称为传递依赖。

## 4. 冗余与更新异常

函数依赖直接关系到冗余数据和更新异常的问题：

- **插入异常**: 插入新数据时, 需要提供冗余信息, 可能因为无相关数据而被迫使用 NULL。
- **删除异常**: 删除某些元组会导致相关重要信息的丢失。
- **修改异常**: 修改某个属性的值时, 可能需要修改多行数据, 增加了出错的风险。

## 5. 设计函数依赖

- 函数依赖不能直接从一个关系的具体实例中推断出来, 而是基于**对属性语义的理解**。
- 设计良好的函数依赖有助于判断关系是否满足规范化条件, 并决定如何对关系进行分解, 以减少冗余和异常。

## 6. 阿姆斯特朗公理 (Armstrong's Axioms)

- 阿姆斯特朗公理是一组推理规则, 用于从已知的函数依赖中推导其他依赖:
  1. **自反性 (Reflexivity)**: 如果  $\beta \subseteq \alpha$ , 则  $\alpha \rightarrow \beta$  成立。
  2. **增广性 (Augmentation)**: 如果  $\alpha \rightarrow \beta$ , 则对任何属性集  $\gamma$ , 有  $\gamma\alpha \rightarrow \gamma\beta$ 。
  3. **传递性 (Transitivity)**: 如果  $\alpha \rightarrow \beta$  且  $\beta \rightarrow \gamma$ , 则  $\alpha \rightarrow \gamma$ 。
- 其他推理规则:
  1. **加法性 (Additivity)**: 若  $\alpha \rightarrow \beta$  且  $\alpha \rightarrow \gamma$ , 则  $\alpha \rightarrow \beta\gamma$ 。
  2. **伪传递性 (Pseudo-Transitivity)**: 若  $\alpha \rightarrow \beta$  且  $\gamma\beta \rightarrow \delta$ , 则  $\alpha\gamma \rightarrow \delta$ 。

## 7. 计算属性闭包 (Closure of Attributes)

- **属性闭包 ( $\alpha^+$ )**: 指的是给定属性集  $\alpha$ , 通过函数依赖推导出所有由  $\alpha$  函数确定的属性集。
- 计算属性闭包的方法:

- 初始令  $\text{result} = \alpha$ 。
- 循环遍历每个函数依赖  $\beta \rightarrow \gamma$ , 如果  $\beta \subseteq \text{result}$ , 则将  $\gamma$  加入  $\text{result}$ , 直到  $\text{result}$  不再变化。

## 8. 计算候选键 (Candidate Key)

- 候选键是能够唯一标识关系中元组的最小属性集。
- 计算候选键的方法:
  - 设定一个包含所有属性的初始集合  $X$ 。
  - 通过不断移除  $X$  中的属性, 保留那些能够保持  $X^+$  覆盖整个关系的属性, 最终得到的最小  $X$  就是候选键。

## 9. 闭包与函数依赖推导

- 函数依赖闭包 ( $F^+$ ): 指的是从给定的函数依赖集合  $F$  推导出的所有可能的函数依赖。
- 通过计算闭包  $F^+$ , 可以确定哪些新的函数依赖是由原始集合  $F$  推导出来的。

## 10. 实例练习

给定关系  $R(A, B, C, G, H, I)$  和函数依赖  $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$ , 可以使用阿姆斯特朗公理推导其他的函数依赖, 例如:

- $A \rightarrow H$  通过传递性从  $A \rightarrow B$  和  $B \rightarrow H$  推导得出。
- $AG \rightarrow I$  通过增广和传递性推导得出。

## 11. 应用示例与依赖图

- 依赖图: 用于可视化地表示属性之间的函数依赖关系, 横线代表函数依赖的左侧, 箭头指向依赖的右侧属性。

# 内存管理

## 一、存储管理与内存层次

数据库系统中的数据存储在多个层次中，这些层次决定了数据的访问速度、成本和持久性。

### 1. 主存 (Primary Storage)

- 特点：
  - 存储在主内存中（例如 RAM），访问速度非常快，时间量级在纳秒 (ns) 范围。
  - 主内存价格昂贵，容量有限，通常不够大，无法存储整个数据库。
  - 容量较小，且是易失性存储，意味着断电或者系统崩溃时，主存中的数据将会丢失。
- 作用：主存主要用于处理查询和计算，操作需要的数据首先要从磁盘读入主存。数据库管理系统（DBMS）在主存中维护一个缓冲池（Buffer Pool），将需要经常访问的数据放入其中以减少磁盘的读取次数。

### 2. 次级存储 (Secondary Storage)

- 磁盘存储：
  - 数据通常存储在磁盘上，典型的如机械硬盘（HDD）或者固态硬盘（SSD）。
  - 访问速度比主存慢得多，但比主存价格便宜，可以用来存储大量数据。
- 特点：
  - **数据持久性：**次级存储的数据不会因电源故障而丢失。
  - 访问速度较慢，因为磁盘的读取速度受物理硬件的影响，包括磁盘旋转和机械臂的移动等因素。

- 磁盘存储架构:
  - 磁盘通常由多个盘片（Platter）组成，每个盘片包含多个磁道（Track），每个磁道又分为若干扇区（Sector）。
  - \*\*块（Block）\*\*是存储和数据传输的基本单位，磁盘上的数据会按块读写。每次读写操作，磁盘都会以块为单位从磁盘移动数据到主存。即使只需要一个记录，也必须读取整个包含该记录的块。

### 3. 存储访问机制

- 块与页的概念:
  - 在数据库中，磁盘存储单位称为块（Block）或者页（Page）。它是存储分配和数据传输的单位。
  - 一个页可以包含多个数据记录，因此数据库操作通常以页为单位进行。
- 访问机制:
  - 数据要被操作时必须先从磁盘读取到主存中，数据的读写速度受到磁盘物理限制，如寻道时间（找到合适的磁道）、旋转延迟（找到合适的扇区）和传输时间（读取数据）等。
  - 磁盘访问时间的影响主要来自寻道时间和旋转延迟，这两者会导致磁盘的随机访问性能不如顺序访问。

## 二、缓冲池管理

缓冲池是数据库系统中用于减少磁盘 I/O 操作的重要机制，以下是详细的缓冲池管理机制及相关概念：

### 1. 缓冲池（Buffer Pool）

缓冲池是主存中用于缓存磁盘块的内存空间，由\*\*页槽（frames）\*\*组成，每个页槽可以缓存一个磁盘块。

- **作用:** 缓冲池用于存储从磁盘读取的数据块，当数据库管理系统需要频繁访问某些数据时，可以通过缓冲池直接访问这些数据，避免反复从磁盘读取，提高系统效率。

## 2. 缓冲区管理的操作

- **请求块 (request\_block):**
  - 当上层需要访问某个数据块时，首先会检查该块是否已经在缓冲池中。如果该块在缓冲池中，则可以直接使用，避免额外的 I/O 操作。
  - 如果该块不在缓冲池中，则需要从磁盘读取数据到缓冲池中。
- **释放块 (release\_block):**
  - 当一个数据块使用结束后，系统会调用**释放块**操作来标记该块不再使用，以便将来可以被替换。
  - 如果块数据被修改过，还需要使用**写块 (write\_block)** 操作将修改后的数据写回磁盘。
- **缓冲池的组成:**
  - 每个缓冲池页槽 (frame) 中包含一些管理信息，例如：
    - **脏位 (Dirty Bit):** 指示该页槽是否被修改过。如果被修改过，那么在将其从缓冲池中移除时，需要将数据写回磁盘。
    - **固定计数 (Pin Count):** 记录当前页槽被多少个事务使用。如果计数大于零，则该页槽不能被替换。

## 三、页替换策略

当缓冲池中没有空闲帧时，系统需要决定应该替换哪一个现有页。数据库管理系统通常使用不同的替换策略来优化性能：

### 1. 最近最少使用 (LRU):

- 替换最久未使用的页。该策略通过维护一个链表，记录页最近的使用情况，越早使用的页会被移到链表的尾部。
- **优点：**在某些应用中（如经常访问的数据页比较稳定），LRU 策略表现较好。
- **缺点：**在“顺序访问”场景中，LRU 策略可能表现较差（称为“**顺序泛滥 (Sequential Flooding)**”），每次访问新的数据页都会导致缓存命中率降低。

## 2. 最常使用 (MRU):

- 替换最近使用的页，通常用于“栈”模式的工作负载。
- **优点：**适用于重复扫描的场景，比如当同一组数据被反复访问时，MRU 能有效提高命中率。
- **缺点：**在非重复的访问模式下性能可能较差。

## 3. 先进先出 (FIFO):

- 按照页进入缓冲池的顺序进行替换，最先进入的页最先被替换。
- **优点：**实现简单，不需要记录复杂的访问历史。
- **缺点：**可能会替换掉仍然需要的页，导致较低的缓存命中率。

## 4. 随机选择 (Random):

- 随机选择缓冲池中的页进行替换。
- **优点：**实现简单，可以避免某些替换策略中的“最坏情况”。
- **缺点：**性能无法保证，缓存命中率完全依赖于随机性。

### 例题：页替换策略的应用

假设数据页有 P1, P2, P3, P4，查询顺序为：

- Q1: 读取 P1; Q2: 读取 P2; Q3: 读取 P3; Q4: 读取 P1; Q5: 读取 P2; Q6: 读取 P4。
- 对于不同策略：
  - **LRU 策略**: 当读取 P4 时, 需要替换掉最久未使用的页 (例如 P3)。
  - **MRU 策略**: 最近使用的页 (例如 P2) 可能会被替换。

从例子中可以看出, 不同策略在不同访问模式下的表现差异较大, 没有一种策略能够在所有情况下表现最佳。

## 四、磁盘空间管理

### 1. 磁盘访问时间

磁盘的访问时间由多个因素组成, 包括:

- **寻道时间**: 磁头移动到目标磁道的时间。
- **旋转延迟**: 磁盘旋转使得目标扇区到达磁头位置所需的时间。
- **传输时间**: 将数据块读写到内存中所需的时间。

磁盘的随机访问时间通常受寻道时间和旋转延迟的影响, 因此顺序访问通常比随机访问更快。

### 2. 提高磁盘访问效率的方法

- **聚簇 (Clustering)**: 将经常一起访问的记录放在同一个块中, 以减少多次读取操作。
- **位图 (Bitmap)**: 用来记录磁盘中的空闲块, 方便快速查找可用空间。

## 五、记录管理

记录在数据库系统中以块的形式存储, 可以是固定长度或者可变长度。

### 1. 固定长度记录

- **特点:** 每个字段都有固定的长度，便于空间管理和定位。
- **优势:** 由于每个记录大小固定，数据库系统可以通过计算偏移量快速定位记录。
- **劣势:** 可能浪费空间，如果某些字段经常为空，或者字段长度远大于实际需要的数据。

## 2. 可变长度记录

- **特点:** 某些字段可能具有可变长度，例如姓名、地址等文本字段。
- **存储方式:**
  - **前缀长度 (Prefix Length):** 在字段前记录长度，方便读取。
  - **字段终止符 (Delimiter):** 使用特殊字符（如 '/') 标识字段结束。
  - **偏移数组 (Offset Array):** 使用偏移数组记录字段在记录中的位置。

## 3. 插槽页管理

- **插槽页 (Slotted Page):**
  - 用于管理存储在页中的多个记录。插槽页包含一个插槽目录 (Slot Directory)，该目录用于记录每个记录的开始位置和长度。
  - **插入和删除操作:** 插入新记录时，会在插槽目录中创建新的条目；删除记录时，标记相应插槽为空，其他记录不受影响。

# 六、索引

索引用于提高查询效率，帮助数据库系统更快地定位需要的数据。索引的类型和使用场景如下：

## 1. B+ 树索引

- **特点:** B+ 树是一种平衡树，叶子节点包含指向数据记录的指针。

- **应用场景：**适合进行范围查询和等值查询，尤其在数据有序存储时，B+ 树能够通过叶节点的链表实现快速的顺序扫描。

## 2. 哈希索引

- **特点：**哈希索引使用哈希函数将键值映射到特定的桶中，桶中存储对应的索引数据。
- **应用场景：**适合进行等值查询（如查找特定学生的记录），因为哈希函数能将键快速映射到特定位置。但它不适合范围查询，因为哈希映射无法保留数据的有序性。

## 七、缓冲区替换策略的性能比较

\*\*缓存命中率（Cache Hit Rate）\*\*是评估缓存性能的一个关键指标，表示有多少请求可以通过缓存直接提供而不需要从磁盘读取。在特定访问模式下，不同策略的表现会有所不同：

- **顺序泛滥（Sequential Flooding）：**在顺序访问大量页的情况下，LRU 策略表现不佳，每次新页的读取都会导致旧页的移除，导致缓存命中率下降。
- **重复访问：**在重复访问同一组数据的情况下，MRU 策略可以显著提高命中率。

# 事务管理

## 一、事务管理

事务是数据库系统中一个执行单元，它包含了一组对数据库的操作，这些操作要么全部执行成功，要么全部执行失败，确保数据的完整性和一致性。

### 1. 事务的 ACID 属性

ACID 是事务管理中的核心原则：

- **Atomicity (原子性):**

- 事务是不可分割的单位，所有操作要么全部执行，要么全部不执行。  
系统需要在发生失败时回滚未完成的操作，确保数据库不被部分更新。
  - 例如，在从账户 A 转移\$50 到账户 B 的事务中，如果在“写入账户 B”的操作前失败，系统必须撤销账户 A 的变化，以确保原子性。

- **Consistency (一致性):**

- 每个事务的执行必须使数据库从一个一致状态变为另一个一致状态。  
在执行事务的过程中，数据库可能处于临时的不一致状态，但事务结束后，必须使得所有约束都得到满足。
  - 例如，银行转账中，无论转账过程如何，最终账户 A 和 B 的余额总和应该保持不变。

- **Isolation (隔离性):**

- 即使多个事务同时执行，它们的执行效果应该与某一个串行执行的事务序列的结果相同。事务的中间状态对其他事务不可见。
  - 例如，如果一个事务修改了账户 A，但尚未完成对账户 B 的更新，则其他事务在此期间不能看到账户 A 的变化，否则会看到不一致的数据。

- **Durability (持久性):**

- 一旦事务提交，修改的数据应该持久保存，即使系统崩溃也不应该丢失。这是通过将事务的日志持久化到磁盘上来实现的。

## 二、事务状态

事务在执行过程中会经历以下几个状态：

1. **Active** (活跃): 事务在执行中。
2. **Partially Committed** (部分提交): 事务的所有操作都已执行，但还未完成持久化。
3. **Failed** (失败): 事务执行过程中出现了错误。
4. **Aborted** (中止): 事务由于失败而被回滚到初始状态，可以选择重新启动或终止。
5. **Committed** (提交): 事务完成了所有操作，并且所有的修改已被持久化。

## 三、并发控制与调度

并发控制的目标是确保多个事务并发执行时不破坏数据库的一致性，同时提高系统的并行性。

### 1. 调度 (Schedule) 与可串行化

- **调度 (Schedule)**: 调度是并发事务的操作执行顺序。
- **串行调度 (Serial Schedule)**: 所有事务的操作按顺序依次执行，不存在交错。
- **可串行化 (Serializable)**: 如果一个调度与某个串行调度结果等效，则称其为可串行化的。通过检测冲突等价性可以判断调度的可串行化。

### 2. 冲突可串行化 (Conflict Serializability)

- **冲突操作**:

- 例如，如果两个事务中的操作访问相同的数据项目且至少有一个是写操作，则称它们是冲突的。
  - 对于非冲突的操作，可以通过交换操作的顺序来保持调度等价。
- 冲突可串行化的检测：
    - 通过构建\*\*优先图（Precedence Graph）\*\*来检测可串行化，如果图中无环，则调度是可串行化的。

## 四、锁机制与并发控制协议

为了保证隔离性，数据库使用锁机制来控制并发事务的操作顺序。

### 1. 锁的种类

- 共享锁（**Shared Lock, S 锁**）：允许多个事务同时读数据，但不允许写。
- 排他锁（**Exclusive Lock, X 锁**）：允许事务读取和写入数据，其他事务不能获得该数据的锁。

### 2. 锁协议

- 两阶段锁协议（**Two-Phase Locking, 2PL**）：
  - 增长阶段：事务可以申请新锁，但不能释放任何锁。
  - 收缩阶段：事务释放锁，但不能申请新锁。
  - 2PL 可以保证调度是可串行化的，但可能会导致死锁。

### 3. 死锁与死锁预防

- 死锁（**Deadlock**）：多个事务相互等待对方持有的锁，导致无法继续执行。
- 死锁预防：
  - 等待超时：如果一个事务等待超过一定时间，则认为其进入了死锁状态，系统强制中止该事务。

- 等待图检测：系统通过构建等待图（Wait-for Graph）来检测死锁，如果存在环则发生死锁。

## 五、事务恢复与数据库日志

在事务执行失败时，数据库必须能够回滚事务的操作以恢复到一致状态，这通过维护\*\*系统日志（System Log）\*\*实现。

### 1. 系统日志

- 日志记录：
  - **[start, T]**: 记录事务 T 的开始。
  - **[write, T, X, old\_value, new\_value]**: 记录事务 T 对数据项 X 的修改，包括修改前后的值。
  - **[commit, T]**: 记录事务 T 的提交，表明 T 的所有修改都可以永久保存。
  - **[abort, T]**: 记录事务 T 的中止。

### 2. 写前日志（Write-Ahead Logging, WAL）

- 写前日志策略：在将数据页写入磁盘前，必须先将对应的日志记录持久化。这确保了即使系统崩溃，日志也能提供足够的信息来恢复数据库。
- UNDO 和 REDO 操作：
  - **UNDO**: 用于回滚失败事务的操作，将数据恢复到旧的值。
  - **REDO**: 用于重新执行已提交事务的操作，确保所有的修改持久化。

### 3. 检查点（Checkpoint）

为了减少恢复时间，系统会周期性地进行检查点操作。检查点将当前所有活跃事务的状态和缓冲区的内容写入磁盘，从而减少在崩溃后需要重新执行的日志数量。

## 六、并发控制协议与恢复策略

## 1. 并发控制协议

- 基于锁的协议:
  - 简单锁协议: 每次使用数据前申请锁, 使用后释放锁。
  - 两阶段锁协议 (2PL): 通过分阶段申请和释放锁来保证可串行化。

## 2. 恢复策略

- 基于日志的恢复:
  - 在事务失败或系统崩溃时, 通过日志来恢复数据库。
  - 通过 **Undo** 恢复未提交事务, 通过 **Redo** 重做已提交的事务, 确保数据库的持久性。

## 七、例子总结

以下是关于并发控制与恢复的一个经典例子——银行转账问题:

- 假设事务 T1 要从账户 A 中取出\$50 并将其存入账户 B, 而事务 T2 要从账户 A 中取出 10% 的余额并存入账户 B。
- 如果两个事务 T1 和 T2 并发执行, 可能会导致不一致的状态。因此需要对这些操作加锁, 并确保按正确的顺序执行, 以保持数据库的 ACID 属性。
- 通过优先图的构建, 系统可以检测冲突并确保调度是可串行化的。同时, 如果系统崩溃, 基于日志的恢复策略会确保事务的原子性和持久性。

## Serializability Testing: Precedence Graph

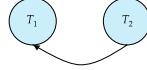
Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$   
**Precedence graph** — a directed graph  $G = (V, E)$  where the vertices ( $V$ ) are the transactions.

We draw an arc from  $T_i$  to  $T_j$ ,  $T_i \rightarrow T_j$ , if the two transactions are conflict, and  $T_i$  accessed the data item earlier.

- $T_i$  executes write(Q) before  $T_j$  executes read(Q)
- $T_i$  executes read(Q) before  $T_j$  executes write(Q)
- $T_i$  executes write(Q) before  $T_j$  executes write(Q)

We may label the arc by the item that was accessed.

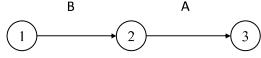
Example (of a precedence graph):



5

## Example

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



If there is **no** cycle in the precedence graph, this schedule is conflict-serializable

Note: Here we label the arc by the item that was accessed.

7

## Example (3)

Example 1:

Schedule	$T_1$	$T_2$	$T_3$
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)			read(C)
$B \leftarrow f_2(B)$		$B \leftarrow f_2(B)$	
$C \leftarrow f_3(C)$			$C \leftarrow f_3(C)$
write(C)			write(C)
$w_1(A)$			
write(A)	write(A)		
read(B)		read(B)	
read(A)			read(A)
$A \leftarrow f_4(A)$		$A \leftarrow f_4(A)$	
read(C)		$A \leftarrow f_4(A)$	
write(A)			write(A)
$C \leftarrow f_5(C)$	$C \leftarrow f_5(C)$		
write(C)			write(C)
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)

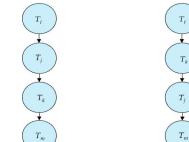
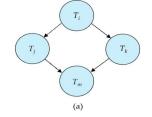
9

## Conflict Serializability Testing

A schedule is conflict serializable if and only if its precedence graph is **acyclic** (cycle free).

If the precedence graph is acyclic, the **serializability order** can be obtained by a **topological sorting** of the graph.

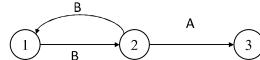
- This is a **linear order** consistent with the partial order of the graph.
- For (a), there are two linear orders (b) and (c).



6

## Example (2)

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



If there is a cycle in the precedence graph, this schedule is NOT conflict-serializable

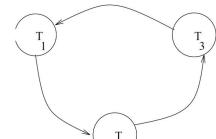
Note: Here we label the arc by the item that was accessed.

8

## Example (3) - Answer

Example 1:

Schedule	$T_1$	$T_2$	$T_3$
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		read(B)
read(C)			read(C)
$B \leftarrow f_2(B)$		$B \leftarrow f_2(B)$	
$C \leftarrow f_3(C)$			$C \leftarrow f_3(C)$
write(C)			write(C)
$w_1(A)$			
write(A)	write(A)		
read(B)		read(B)	
read(A)			read(A)
$A \leftarrow f_4(A)$		$A \leftarrow f_4(A)$	
read(C)		$A \leftarrow f_4(A)$	
write(A)			write(A)
$C \leftarrow f_5(C)$	$C \leftarrow f_5(C)$		
write(C)			write(C)
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)



10

## Example (4)

Example 2:

Schedule	$T_1$	$T_2$	$T_3$
read(A)	read(A)		
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)	read(C)		
write(A)	write(A)		
$A \leftarrow f_2(C)$	$A \leftarrow f_2(C)$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow f_3(B)$	$B \leftarrow f_3(B)$		
write(B)	write(B)		
$C \leftarrow f_4(C)$		$C \leftarrow f_4(C)$	
read(B)		read(B)	
write(C)		write(C)	
$A \leftarrow f_5(A)$	$A \leftarrow f_5(A)$		
write(A)	write(A)		
$B \leftarrow f_6(B)$		$B \leftarrow f_6(B)$	
write(B)		write(B)	

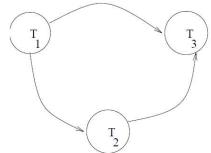
11

## Example (4) - Answer

Example 2:

Schedule	$T_1$	$T_2$	$T_3$
read(A)	read(A)		
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)	read(C)		
write(A)	write(A)		
$A \leftarrow f_2(C)$	$A \leftarrow f_2(C)$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow f_3(B)$	$B \leftarrow f_3(B)$		
write(B)	write(B)		
$C \leftarrow f_4(C)$		$C \leftarrow f_4(C)$	
read(B)		read(B)	
write(C)		write(C)	
$A \leftarrow f_5(A)$	$A \leftarrow f_5(A)$		
write(A)	write(A)		
$B \leftarrow f_6(B)$		$B \leftarrow f_6(B)$	
write(B)		write(B)	

12



# NoSQL

## 一、NoSQL 数据库简介

- **NoSQL** 是“not only SQL”的缩写，指的是与传统 SQL 不同的数据库管理系统。它们最早于 2009 年提出，主要用于处理**大数据** (Big Data)，包括非结构化和半结构化的数据。
- 大数据的特点：
  - **Volume** (数据量)：数据量非常大，例如 Facebook 每天产生数百 TB 的用户日志和图片。
  - **Velocity** (速度)：数据生成和插入速度非常高，例如物联网设备、社交媒体数据等。
  - **Variety** (多样性)：数据类型复杂，涵盖文本、图片、音频等多种格式。

## 二、RDBMS 和 NoSQL 的比较

- **RDBMS** 采用表和列来存储数据，使用结构化查询语言 (SQL) 进行数据操作，提供 **ACID** 特性（原子性、一致性、隔离性、持久性），支持事务和数据的一致性。
- **NoSQL** 则不使用关系模型，大多数采用分布式集群架构，具有动态数据模式，可以进行水平扩展，特别适合处理大规模非结构化数据。

在**大数据**背景下，传统的关系型数据库往往面临扩展性瓶颈，因此一些应用场景选择牺牲 ACID 特性来换取更高的扩展性和灵活性，例如使用 NoSQL 来存储和处理这些数据。

## 三、CAP 定理

CAP 定理指出，在分布式系统中，最多只能同时满足以下三个特性中的两个：

1. **一致性 (Consistency)**：所有节点在同一时间的数据状态一致。

2. **可用性 (Availability):** 每个请求都能接收到成功的响应，无论系统内部状态如何。
3. **分区容错性 (Partition Tolerance):** 系统即使在部分节点发生故障或通信失败的情况下仍能继续运作。

对于 NoSQL 数据库，通常会在一致性和可用性之间做出权衡，选择合适的特性组合来适应不同的应用需求。

#### 四、NoSQL 数据库的类型

NoSQL 数据库根据数据模型可以分为以下几种类型：

##### 1. 键值存储 (Key-Value Stores)

- **数据模型:** 类似哈希表，以键值对形式存储数据，键用于唯一标识数据，值为未结构化的数据。
- **特点:**
  - 数据简单，不要求固定的模式。
  - 可以快速进行数据插入、删除和查找，具有很好的扩展性。
- **缺点:** 对于复杂的数据查询和连接操作支持不足。
- **代表:** Redis、Amazon DynamoDB。

##### 2. 文档数据库 (Document Stores)

- **数据模型:** 使用类似 JSON 或 BSON 格式来存储数据。每个文档是自描述的，可以包含嵌套的数组和对象。
- **特点:**
  - 支持复杂的层次结构，数据可以包含嵌套的集合。
  - 可以根据文档中的字段创建索引，提升查询效率。
- **使用场景:** 适用于需要灵活数据模式的应用，如内容管理系统。

- **代表:** MongoDB。
- **示例:** 文档可以类似于:

```
{  
  "_id": 1,  
  "name": "John Doe",  
  "age": 29,  
  "addresses": [  
    { "city": "New York", "zip": "10001" },  
    { "city": "San Francisco", "zip": "94105" }  
  ]  
}
```

### 3. 列族存储 (Column-Family Stores)

- **数据模型:** 起源于 Google 的 BigTable，将数据按列族进行存储，每个行键对应多个列，且每个列族内的列可以动态增加。
- **特点:**
  - 适合 OLAP (在线分析处理)，可以进行高效的数据聚合操作。
  - 数据按列存储，更适合进行大规模的列访问。
- **缺点:** 不适用于高并发的 OLTP (在线事务处理)。
- **代表:** Apache Cassandra、HBase。

### 4. 图数据库 (Graph Databases)

- **数据模型:** 使用图结构来表示数据，包括节点 (实体) 和边 (关系)，每个节点和边都可以有属性。

- **特点:**
  - 适合表示高度互联的数据，特别是社交网络、推荐系统等场景。
  - 查询语言通常是图专用的语言，例如 Neo4j 的 Cypher。
- **使用场景:** 社交网络、推荐系统、物联网等需要复杂关系的数据。
- **代表:** Neo4j。
- **示例:** 在社交网络中，节点表示用户，边表示“朋友”关系，查询某用户的朋友可以通过图遍历实现。

## 五、MongoDB 和 Neo4j 的对比

- **MongoDB** 主要用于存储具有层次结构的半结构化数据，适合需要灵活数据结构和高扩展性的场景。
- **Neo4j** 主要用于处理复杂关系的数据，适合需要频繁进行关系查询和遍历的场景，例如社交网络中的好友推荐。

## 六、选择合适的数据库

在选择数据库时，需要根据应用的具体需求来决定：

- **结构化数据:** RDBMS 更适合。
- **非结构化或半结构化数据:** NoSQL 更具灵活性，尤其是在高并发、大数据量场景下。
- **高度互联的数据:** 图数据库如 Neo4j 是更好的选择。

例如：

- 如果要存储社交网络的好友关系并进行推荐，那么图数据库是最佳选择，因为它能够高效地处理复杂的关系。
- 如果需要快速扩展且不要求复杂事务管理的系统，可以选择键值存储或文档存储，例如用 MongoDB 来管理用户个人信息和偏好。