relational algebra & calculus，SQL，embedded SQL，optimization，concurrency，recovery，trigger，integrity constraint & NF，概念性简答，零散知识点。

relational algebra & calculus

selection（σ），projection（π），cross-product（×），set-difference（-），union（∪）。

division，ABCD / CD，algebra DRC TRC：

- $\pi_{AB}(R1) - \pi_{AB}((\pi_{AB}(R1) \times R2) - R1)$。
- $\{\langle A, B\rangle \mid \forall \langle C, D\rangle \in R2\ (\langle A, B, C, D\rangle \in R1)\ \}$。
- $$\{P \mid \forall\, t2 \in R2\ \exists\, t1 \in R1 \\ (t1.A = P.A\ \wedge\ t1.B = P.B\ \wedge\ t1.C = t2.C\ \wedge\ t1.D = t2.D)\}$$
- $\{t \mid \forall\, t2 \in R2\ (\langle t, t2\rangle \in R1)\}$。

2. The following relational modes are given: R = (A, B, C), S = (D, E, F). Let the relations r(R) and s(S) be known. Please give the tuple relational calculus expression equivalent to the following expression: (8+10+13=31)
   a) $\prod_A(r)$
   b) $\prod_{A,F}(\sigma_{C=D}(r \times s))$
   c) $r *\bowtie_{C=D} s$
   a) $\prod_A(r) = \{t[A] | t \in r\}$
   b) $\prod_{A,F}(\sigma_{C=D}(r \times s)) = \{t[AF] | t[ABC] \in r \wedge t[DEF] \in s \wedge t.C = t.D\}$
   c) $r *\bowtie_{C=D} s = \{t[ABCDEF] | (t[ABC] \in r \wedge t[DEF] \in s \wedge t.C = t.D) \vee (t[ABC] \in r \wedge \neg(t.C \in s[D]) \wedge t.D = NULL \wedge t.F = NULL \wedge t.F = NULL)\}$

SQL查询

嵌套查询是IN，关联查询是EXIST。

WITH

```
WITH payroll (deptno, totalpay) AS
    (SELECT deptno, sum(salary)+sum(bonus)
    FROM emp
    GROUP BY deptno)
SELECT deptno
FROM payroll
WHERE totalpay=(SELECT max(totalpay)
                FROM payroll);
```

冷门用法

没见过的in

```
WHERE GetMon(N.say) in {7,8}
```

case

```
select sid,
    sum(case when cid=01 then score else null end) as score_01,
    sum(case when cid=02 then score else null end) as score_02,
    sum(case when cid=03 then score else null end) as score_03,
    avg(score)
from sc group by sid
order by avg(score) desc
```

cast： outer join

把 Teacher (name, rank) 和 Course (subject, enrollment, quarter, teacher) 做 outer-join。

```
WITH
    innerjoin(name, rank, subject, enrollment) AS
        (SELECT t.name, t.rank, c.subject, c.enrollment
        FROM teachers AS t, courses AS c
        WHERE t.name=c.teacher AND c.quarter='Fall 19') ,
    teacher-only(name, rank) AS
        (SELECT name, rank
        FROM teachers
        EXCEPT ALL
        SELECT name, rank
        FROM innerjoin) ,
    course-only(subject, enrollment) AS
        (SELECT subject, enrollment
        FROM courses
        EXCEPT ALL
        SELECT subject, enrollment
        FROM innerjoin)
SELECT name, rank, subject, enrollment
FROM innerjoin
UNION ALL
SELECT name, rank,
    CAST (NULL AS Varchar(20)) AS subject,
    CAST (NULL AS Integer) AS enrollment
FROM teacher-only
UNION ALL
SELECT CAST (NULL AS Varchar(20)) AS name,
    CAST (NULL AS Varchar(20)) AS rank,
    subject, enrollment
FROM course-only;
```

recursion

Hoover 直接间接管理的所有人。

```sql
WITH agents(name, salary) AS
    ((SELECT name, salary FROM FedEmp
     WHERE manager='Hoover')
    UNION ALL
    (SELECT f.name, f.salary
     FROM agents AS a, FedEmp AS f
     WHERE f.manager=a.name))
SELECT name
FROM agents
WHERE salary>100000;
```

查找所有零部件。

```sql
WITH wingpart(subpart, qty) AS
    ((SELECT subpart, qty FROM components
     WHERE part='wing')
    UNION ALL
    (SELECT c.subpart, w.qty*c.qty
    FROM wingpart w, components c
    WHERE w.subpart=c.part))
SELECT sum(qty) AS qty
FROM wingpart
WHERE subpart='rivet';
```

embedded SQL

```sql
// 定义宿主变量
EXEC SQL BEGIN DECLARE SECTION;
char SNO[7];
char GIVENSNO[7];
char CNO[6];
char GIVENCNO[6];
float GRADE;
short GRADEI; /*indicator of GRADE*/
EXEC SQL END DECLARE SECTION;

// 连接数据库
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
// SQL INSERT
EXEC SQL INSERT INTO SC(SNO,CNO,GRADE)
    VALUES(:SNO, :CNO, :GRADE);
// SQL 查询
EXEC SQL SELECT GRADE
    INTO :GRADE :GRADEI
    FROM SC
    WHERE SNO=:GIVENSNO AND
        CNO=:GIVENCNO;
// 定义cursor
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT SNO, GRADE
    FROM SC
    WHERE CNO = :GIVENCNO;
EXEC SQL OPEN C1; // open
if (SQLCA.SQLCODE<0) exit(1); // <0，查询发生错误
while (1) {
```

```
    EXEC SQL FETCH C1 INTO :SNO, :GRADE :GRADEI;
    if (SQLCA.SQLCODE==100) break;
    // ...
}
EXEC SQL CLOSE C1; // close
```

query optimization（索引，B+树）

简答

为什么 relational DB 比 hierarchical / network DB 更需要优化?

- 层次和网状数据库的 data model 使用指针表示属性之间的关系，这样的结构固定了数据库的查询路径，优化空间有限。
- 而 relational data model 提供了query optimization 的空间。relational DB 抽象程度高，查询语言 SQL 是非过程性语言，效果等价的不同实现效率相差很大。
- relational data model 用表来表示实体间的联系，是一种软连接，查询涉及大量连接操作，因此必须优化以解决效率问题。

B树是主流：

- 索引一般以文件形式存储在磁盘上，索引查找会带来磁盘IO，因此要选择索引过程中磁盘IO更少的索引数据结构。B树检索一次最多需要 树高-1 次IO（根节点常驻内存），树的度非常大，因此树高非常小（≤3，且受数据量增长影响很小），比红黑树磁盘IO更少。
- B树的变种B+树不仅支持索引查找，还支持O(n)的顺序遍历和范围查询。相比 hash 索引，虽然 hash 索引查找复杂度是O(1)，但仅能精确查询，不能范围查询，不能进行数据排序，大量 hash 值相等时性能显著下降。

B+树比B树好：

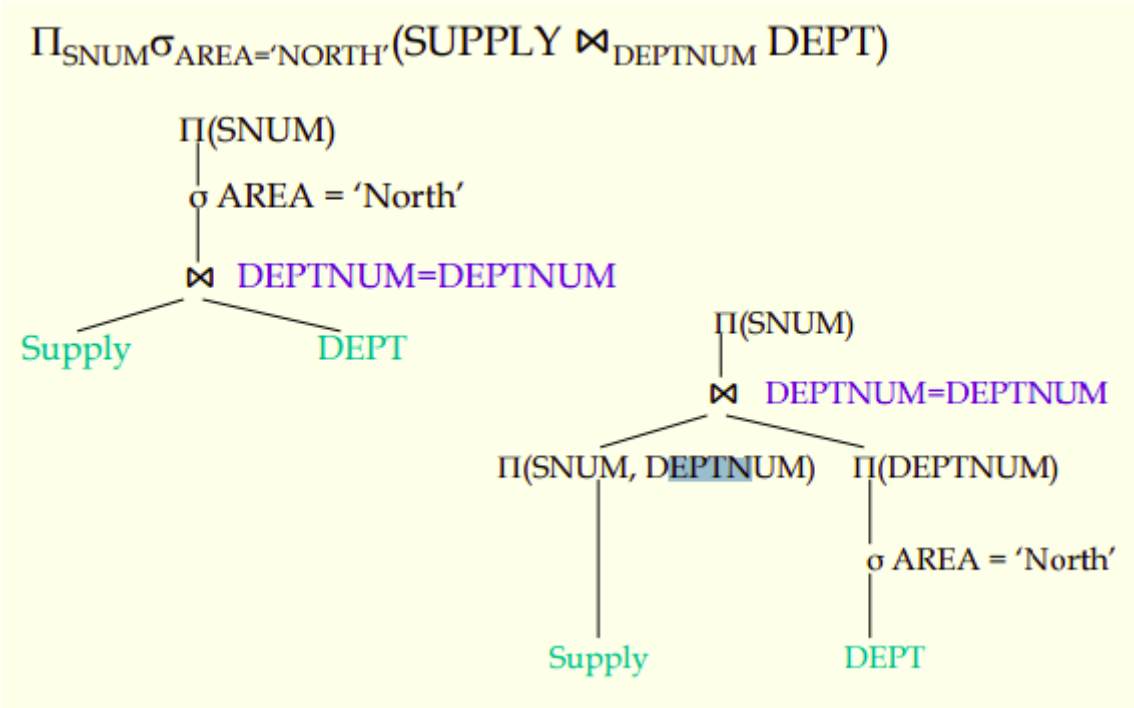
- 查询效率更高。B+树使用双向链表串连所有叶子节点，顺序遍历和区间查询效率更高，B树则需要中序遍历才能完成范围查找。
- 查询效率更稳定。B+树每次都必须查询到叶节点才能找到数据，而B树查询的数据可能在内节点。
- 磁盘读写代价更小。B+树的内部节点不存数据只存索引，因此内节点比B树更小，存内节点的物理块更少，查索引的相应时间更小。通常B+树更矮胖，高度小，查询产生的I/O更少。

不使用or谓词：不好优化。

- 只能按其中各个条件分别选出一个tuple集，再求并，并是开销大的操作。
- 而且，在 OR 连接的多个条件中，只要有一个条件没有合适的存取路径，就不得不采用顺序扫描来处理这种查询，效率大大降低。

algebra optimization

代数优化：变换查询树。主要优化 cross-product、union、set-difference。



operation optimization

join的优化：

- buffer：两个缓冲，大小是物理块大小的整数倍，一个给外循环表，一个给内循环表。外循环中，一次取好多物理块，内循环也是如此；每次比较时，两个缓冲的内容两两比较。
  - 分配缓冲区，希望读物理块个数最少：外循环n-1，内循环1，内循环遍历次数少。
- merge scan：需要两个关系 R S，在磁盘上【按照连接属性的值】，事先做好外排序。双指针法，每个表扫一次。
- B+树索引/哈希索引：
  - 把【连接属性上有索引】的表作为内循环表。根据外循环缓冲中的值，查内循环表的索引，再也不用扫描内循环表了。
  - 如果匹配tuple的数量达到内循环表的20%，用（非簇集的）索引不合算。

concurrency

问题：lost update，dirty read，unrepeatable read。

2PL：如果所有的加锁请求都在释放锁请求之前，即 一起拿到锁 → 搞事情 → 一起还锁，那么transaction：两段事务，two phase transaction；对事务的限制：两段加锁协议，two phase locking protocol，2PL。

well-formed：在操作 object 前先 acquire lock。

serializable：只要 well-formed && two phase，反证法。

(S, U, X) locks：与 AI → DB after commit 策略配合使用，写入DB时再升级为X锁。

简答

为什么并发调度的criterion是serializable?

- 事务并发执行的调度是可串行化的，就是说，这个并发调度与该事务的串行调度等价。
- 对于串行调度，各事务的操作没有交叉，没有互相干扰，因此不会产生并发执行时的冲突问题。
- 所以，与之等价的事务并发调度也不会产生冲突，即并发结果是正确的。

为什么把操作系统中可用的 Requesting all locks at initial time of transaction（一开始拿到所有要用的锁）用于数据库系统，理论上可行，实际却做不到？

- 数据库的操作是动态的增删查改，运行前不能确定具体要访问哪些数据对象，要在程序运行前一次性锁住所有要访问的对象，当然做不到。

已有的 (S,X)、(S,U,X) 锁能解决事务并发中的死锁问题吗？为什么？

- 不能。当一个事物 A 占用数据对象 a 的X锁，事务 B 占用数据对象 b 的X锁，事务 A 和事务 B 又分别申请 b 和 a 的锁，在 (S,X) 和 (S,U,X) 锁中均无法获准，需要等待对方事务释放锁。而进入等待状态则无法释放自己所占用的锁，从而陷入循环等待，即死锁。

U锁的好处？为什么已经加了U锁，不允许其它事务申请加U锁？ 如果允许会出现什么情况？

- S和U锁相容，可以提高并发度；U和U锁相容，造成死锁。
- U锁表示事务对数据对象进行更新的操作，在最后写入阶段再将其升级为X锁。若在U锁阶段允许其他事务申请U锁，则进入写DB阶段、想将U锁升级为X锁时，由于存在其他事物对数据对象的U锁，无法升级为X锁，只能保持U锁；另一个事务也无法升级为X锁，从而导致死锁。

recovery

acid： atomic，consistency preservation（不会操作到一半），isolation（独占），durability（可恢复）。

在做后备副本后，以前的运行记录就失去价值，可以把 CTL / Log 清空。

更新事务在执行时应遵守下列两条规则。

(1) 提交规则(commit rule)

后像必须在事务提交前写入非易失存储器中，即写入数据库或运行记录的后像文件中。

根据 ACID 准则，提交的事务对数据库所产生的影响应是持久的，后像只有写入不易失存储器才能有持久性。但提交规则并不要求后像一定在事务提交前写入数据库。如果后像已经写入后像文件，即使还未写入数据库或未完全写入数据库，事务仍可提交。在后像未写入数据库期间，如果发生故障，可用后像文件中的后像重做；若有其他事务访问这些数据，由于更新的内容已保存在后像文件中，或可能仍在缓冲区中，其他事务可从缓冲区或后像文件访问更新后的内容。

(2) 先记后写规则(log ahead rule)

如果后像在事务提交前写入数据库，则必须把前像首先记入运行记录。

事务在提交前，都有可能失败。事务失败后，须撤销事务对数据库所做的一切更新。为此，必须在改动数据库前，先把前像记入运行记录。也就是先要把老的内容"留底"，才能写入新的内容。

数据冗余：

- 缺点：造成存储浪费，造成数据不一致，带来版本控制问题。
- 优点：对数据库恢复，数据冗余是必须的。

checkpoint：

- DBMS 成批延迟更新数据，给系统故障的redo带来负担。
  - 系统故障：比如DBMS或OS崩掉，并没有丢失数据库的数据，只是现在有些transaction写了一半，不是consistent的。
  - 介质失效：磁盘发生故障，数据库受损，数据全丢失，如磁盘、刺头破损。
- checkpoint： DBMS周期性强制写入所有（已 commit 未写入 DB 的 transaction的）后项。发生系统故障时，只需要对最近 checkpoint 之后 commit 的 transaction 做 redo。
- 用于数据库发生系统失效时，避免大量的无效 redo。

redo 和 undo ， idempotent（幂等）。

redo否的标答：

- 介质失效指磁盘发生故障，数据库受损，如磁盘、刺头破损。 介质失效后，应该在新介质中加载最近后备副本，并基于 Log 内运行记录中的后像，redo 后备副本以后提交的所有更新事物。
- 如果检查点比后备副本要新，则对后备副本以后、检查点以前的事务也应该 redo。

strategies

- AI → DB before commit，直接改数据库。仅 undo。
  - 我们把BI记到Log（log ahead），然后把DB改成AI。commit阶段，只要表示我done了（把 TID 放到 commit list，然后从 active list 里删除）就ok了。
  - 恢复过程：对每个 checkpoint 后的 TID，检查 active/commit list。
    - 若仅 active list，利用 log 中的 BI 来 undo；
    - 若 active+commit list，delete TID from active list；
    - 若仅 commit，则已经完全成功，无需任何操作。
- AI → DB after commit，确认事务可以成功再改DB。仅 redo。
  - 我们把AI记到Log（commit rule），等到 commit 阶段，先把TID放到 commit list 里，然后进行一个AI的改，最后把AI从 active list 里删除。
  - 恢复过程：
    - 若仅 active list，事务还什么都没做呢，delete TID from active list；
    - 若 active+commit list，则还没完全写入DB，根据 log 中的 AI 来 redo；
    - 若仅 commit，则已经完全成功，无需任何操作。
- AI → DB concurrently with commit，利用硬盘的空闲时间，见缝插针改DB。undo+redo。
  - 先把BI AI都记到Log（两个rule），然后就开始改，commit阶段开始了，TID → commit list，然后继续改，直到改完，把自己从 active list 里删除。
  - 恢复过程：
    - 若仅 active list，可能 partially 写入 DB，undo；
    - 若 active+commit list，可能 partially 写入 DB，redo 且 delete TID from active list；
    - 若仅 commit，则已经完全成功，无需任何操作。

trigger

```
CREATE TRIGGER <TriggerName>
  {before|after } {insert|delete|update}
  ON <RelationName>
  [referencing old as <OldName>, new as <NewName>]
  [WHEN <Condition>]
  [for each row|statement]
  <SQLStatement>
```

规则一

```
CREATE TRIGGER referential_integrity_check
BEFORE INSERT ON SC
REFERENCING NEW AS N
FOR EACH ROW
WHEN (NOT (EXISTS (SELECT * FROM STUDENT
                   WHERE SNO=N.SNO)
           AND
           EXISTS (SELECT * FROM COURSE
                   WHERE CNO=N.CNO)))
ROLLBACK;
```

如果 SC 表中插入元组，其外键在 STUDENT 或 COURSE 表中不存在，则卷回插入该元组操作。在此例中，SNO,CNO 既分别是外键，又共同构成 SC 表中的主键。在 SC 表的定义中，它们都以 NOT NULL 说明，因此不能置成 NULL。

规则二

```
CREATE TRIGGER course_delete
BEFORE DELETE ON COURSE
REFERENCING OLD AS O
FOR EACH ROW
WHEN (EXISTS (SELECT * FROM SC
              WHERE CNO=O.CNO))
ROLLBACK;
```

如果 COURSE 表中删除一元组，若该元组的主键是 SC 表中的外键，则卷回删除该元组的操作。因为在 SC 表的定义中，外键 CNO 的定义中采用了 RESTRICT 选项。

规则三

```
CREATE TRIGGER student_delete
AFTER DELETE ON STUDENT
REFERENCING OLD AS O
FOR EACH ROW
WHEN (EXISTS (SELECT * FROM SC
              WHERE SNO=O.SNO))
DELETE FROM SC
WHERE SNO=O.SNO;
```

规则四

```
CREATE TRIGGER SC_fk_update
BEFORE UPDATE OF SNO,CNO ON SC
REFERENCING NEW AS N
FOR EACH ROW
WHEN (NOT (EXISTS (SELECT * FROM STUDENT
                   WHERE SNO=N.SNO)
           AND
           EXISTS (SELECT * FROM COURSE
                   WHERE CNO=N.CNO)))
ROLLBACK;
```

如果修改SC表中的外键，使STUDENT或COURSE表中无相应的主键供其引用，则卷回更新此元组的操作。在同时修改SNO,CNO时，如果其中一个有主键对应，另一个没有主键对应，则有些修改所涉及的元组可以通过，有些元组则不可通过。因此，按元组卷回是必要的。

规则五

```
CREATE TRIGGER course_CNO_update
BEFORE UPDATE OF CNO ON COURSE
REFERENCING OLD AS O
FOR EACH ROW
WHEN (EXISTS (SELECT * FROM SC
              WHERE CNO=O.CNO))
ROLLBACK;
```

在修改COURSE表中的主键CNO时，如果SC表中有元组引用修改前的CNO值作为外键，则卷回此修改操作。

规则六

```
CREATE TRIGGER student_SNO_update
BEFORE UPDATE OF SNO ON STUDENT
REFERENCING OLD AS O
FOR EACH ROW
WHEN (EXISTS (SELECT * FROM SC
              WHERE SNO=O.SNO))
ROLLBACK;
```

在修改STUDENT表中的主键SNO时，如果SC表中有元组引用修改前的CNO值作为外键，则卷回此修改操作。

规则一

```
CREATE TRIGGER student_delete
AFTER DELETE ON STUDENT
REFERENCING OLD TABLE AS OT
FOR EACH STATEMENT
WHEN (EXISTS (SELECT * FROM OT
              WHERE SEX='女'))
DELETE FROM FGRADE;
INSERT INTO FGRADE
    SELECT SNAME,CNO,GRADE
    FROM STUDENT,SC
    WHERE STUDENT.SNO=SC.SNO AND SEX='女';
```

如果删除的元组中有性别为女的元组，则清除FGRADE表的全部内容，重新按定义生成，即刷新FGRADE表。OT是一个表的变量名，它的模式与STUDENT表一致，可与普通表一样处理。在主动数据库系统中，SQL语言的实现应考虑到这类表。

规则三 按照是否更新 SEX 属性分为下面两条规则。

(a) 更新 SEX 属性

```
CREATE TRIGGER student_update_sex
AFTER UPDATE OF SEX ON STUDENT
FOR EACH STATEMENT
DELETE FROM FGRADE;
INSERT INTO FGRADE
    SELECT SNAME,CNO,GRADE
    FROM STUDENT,SC
    WHERE STUDENT.SNO=SC.SNO AND SEX='女';
```

更新 SEX 属性,必然会影响 FGRADE。故在本规则中,略去 WHEN 子句,即无条件刷新。

(b) 不更新 SEX 属性

```
CREATE TRIGGER student_update

    AFTER UPDATE OF SNAME, SNO ON STUDENT
    REFERENCING OLD TABLE AS OT
    FOR EACH STATEMENT
    WHEN (EXISTS (SELECT * FROM OT
                  WHERE SEX='女'))
    DELETE FROM FGRADE;
    INSERT INTO FGRADE
        SELECT SNAME,CNO,GRADE
        FROM STUDENT, SC
        WHERE STUDENT.SNO=SC.SNO AND SEX='女';
```

如果修改女生的 SNAME,SNO 属性,则会影响 FGRADE,FGRADE 须刷新。

规则四

```
CREATE TRIGGER SC_delete
AFTER DELETE ON SC
REFERENCING OLD TABLE AS OT
FOR EACH STATEMENT
WHEN (EXISTS (SELECT * FROM OT, STUDENT
              WHERE OT.SNO=STUDENT.SNO AND SEX='女'))
DELETE FROM FGRADE;
INSERT INTO FGRADE
    SELECT SNAME,CNO,GRADE
    FROM STUDENT,SC
    WHERE STUDENT.SNO=SC.SNO AND SEX='女';
```

在删除的 SC 元组中,若有女生成绩元组,则刷新 FGRADE。

规则六

修改 SC 的任何一个属性，都可能影响 FGRADE。因此在 UPDATE 后不说明修改的属性名。

```
CREATE TRIGGER SC_update
AFTER UPDATE ON SC
REFERENCING OLD TABLE AS OT, NEW TABLE AS NT
FOR EACH STATEMENT
WHEN (EXISTS (SELECT * FROM OT, STUDENT
                WHERE OT.SNO=STUDENT.SNO AND SEX='女')
      OR
      EXISTS (SELECT * FROM NT, STUDENT
                WHERE NT.SNO=STUDENT.SNO AND SEX='女'))
DELETE FROM FGRADE;
INSERT INTO FGRADE
    SELECT SNAME,CNO,GRADE
    FROM STUDENT,SC
    WHERE STUDENT.SNO=SC.SNO AND SEX='女';
```

当库存量低于库存下限，而在购订单表中没有该零件的订单时，则将该零件的订单插入到在购订单表中，订单的订购数量按库存表中的规定。下面是控制库存量的规则：

```
CREATE TRIGGER 库存控制
AFTER UPDATE OF 库存量 ON 库存
REFERENCING NEW AS N
FOR EACH ROW
WHEN (N.库存量<N.库存下限
      AND
      NOT EXISTS (SELECT * FROM 在购订单
                    WHERE 零件号=N.零件号))
INSERT INTO 在购订单
    VALUES (N.零件号,N.订购量,SYSDATE);
```

在上面的规则中，订购日期指发出订单的日期，即系统的日期。系统日期的表示在不同的 SQL 版本中不完全统一，如 SYSDATE,CURRENT_DATE 等，这里暂用 SYSDATE。

8、试编写一个触发器，监视上面数据库中 enroll 表上的 INSERT 操作，对每条 INSERT 语句，判断其插入元组的 grade 值是否小于 3.0，将这样的元组自动插入到 failedcourse 表中(设 filedcourse 表与 enroll 表的模式完全相同)。(10 分)

```
CREATE TRIGGER insert_grade_check
AFTER INSERT ON enroll
REFERENCING NEW TABLE AS NE
FOR EACH STATEMENT
WHEN (EXISTS(SELECT * FROM NE
                WHERE grade<3.0))
INSERT
    INTO failedcourse
    SELECT * FROM NE
    WHERE NE.grade<3.0
```

d.指出每个关系属于第

8. 假设规定每个水手最多收两名徒弟，编写一个触发器，监视第 3 题 Sailors 表上的 Insert 操作，对添加的每条记录判断其师傅水手是否满足该约束（如果有师傅水手），若不满足约束，执行回卷操作。(8 分)

```
create trigger xxx before insert on Sailors
referencing NEW as N for each row
when not (
    exists (
        select * from sailors
        where sailors.sid = N.master)
    and
        (select count(*) from sailors
        where sailors.master = N.master) < 2
) rollback;
```

integrity constraint，NF

NF

- 1NF：atomic。
- 2NF：1NF，且no partially function dependency exists between attributes，不存在部分函数依赖（只依赖一个主键）。
- 3NF【常用】：2NF，且no transfer function dependency exists between attributes，不存在属性对主键的传递依赖。
- 4NF：3NF，无多值依赖。5NF：4NF，无连接依赖。
- insert（难以记录信息） / delete abnormity（删没了就丢失信息），redundancy / hard to update。
- criterion：一事一地，one fact in one place。
- 范式不是越高越好，应取决于应用。高范式的主要目的是防止数据冗余和更新 / 插入 / 删除异常，而范式高会存在处理速度缓慢和处理逻辑复杂的问题，从而降低数据库性能，因此需要权衡考虑。

integrity constraint

- inherent constraint：固有的约束，不允许表中套表。
- implicit constraints：在schema里定义的。
  - domain constraint（学生绩点不是字符串）。
  - primary key constraint（主键不能重复不能为NULL）。
  - foreign key constraint（外键不能荡空，引用完整性）。具体实现：报错 / 级联删除 级联更新（修改主键时）。
- explicit constraint：断言，check语句。
  - 在程序里定义：不合法的事情直接报错，不会送给数据库。
  - 用ASSERT断言语句。

    ```
    ASSERT balanceCons ON account: balance>=0;
    ```

  - 在CREATE TABLE时使用CHECK语句。

    ```
    -- Interlake不能借
    CREATE TABLE Reserves
    (sname CHAR(10), bid INTEGER, day DATE,
    PRIMARY KEY (bid, day),
    CONSTRAINT noInterlakeRes
    CHECK ('Interlake' <>
    (SELECT B.bname FROM Boats B WHERE B.bid=bid)
    ));
    ```

- 多张表上的constraints：创建断言。

```
CREATE ASSERTION smallClub
CHECK
((SELECT COUNT(S.sid) FROM Sailors S) +(SELECT COUNT(B.bid) FROM
Boats B) < 100);
```

概念性简答

data、DB、DBMS、DBS、data model、schema：

- database： a very large, integrated collection of data，数据的集合。
- DBMS： a software package designed to store and manage databases，管理数据，os和user之间的系统软件。好处：data independence, efficient access, reduced application development time, integrity & security, concurrency。
- database system：包括DBMS、application、data本身、DBMS的开发工具、DBA、用户，是一整套数据库的集合。
- data：描述现实事物的符号，信息的存在形式。
- data model： a collection of concepts and definitions for describing data。
- schema： a description of a particular collection of data, using a given data model，具体数据模型的实例。

data independence：

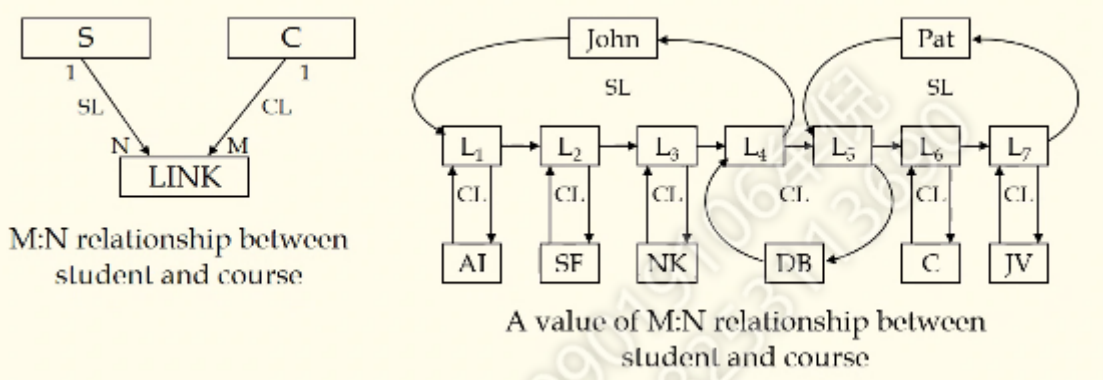
- 约定好层间的接口，然后互不影响。物理 / 逻辑独立性。
- 文件 / DB：文件系统中的文件面向应用，一个文件对应一个application，文件之间不存在联系，冗余大，共享性差，独立性差；数据库系统中的文件面向整个应用系统，文件之间相互联系，冗余小，共享性好，独立性好。
- ANSI-SPARC： external schema/view，conceptual schema，physical schema。
- 数据独立性越高越好：应用程序对数据库 / 数据逻辑结构对物理存储方式 的依赖越严重，整个系统就越难维护。

relational algebra，calculus，SQL：

- relational algebra：需要保证运算顺序，即对查询过程给出确定性描述，procedural。
- relational calculus：不需要知道运算过程和运算符顺序，non-procedural。
- SQL： structured query language → standard query language。relational algebra / calculus 为 SQL 提供理论基础。区别：技术规范 / 数学理论。
  - query language不是programming language，不是图灵完备（turing complete）的。
  - data definition language (DDL)，query language (QL)，data manipulation language (DML)，data control language (DCL)。

hierarchical data model： tree structure，无法建模M:N的关系（3个学生同时参加两门课），multi parents，三方的关系。补救方法：virtual record，或者叫做pointer。

network data model： set（1:N的关系，owner对应N个member）。用循环链表实现。M:N，multi parents，三方关系：LINK record type（relational dm是soft link）。



relational data model：

- 优点：关系模型中将现实世界中的实体以及实体之间的联系统一用关系（表）来表示。与层次 & 网状数据模型中用物理指针表示实体间的联系相比，关系模型用表实现了“软连接”。由于被查询的对像是关系，查询结果还是关系，因此可构成一个代数系统，即关系代数，这使得我们可以用数学的方法来研究数据库中的问题。关系模型概念非常简单，又解决了查询效率的问题，因此自70年代出现后，很快就取代层次和网状数据模型而成为主流。
- 缺点：以记录为基础，不能很好面向用户和应用；不能以自然的方式表达实体间联系；语义贫乏；数据类型太少，不能满足应用需要。

结构化/非结构化数据：

- 结构化：由二维表结构来逻辑表达和实现的数据，严格地遵循数据格式与长度规范，主要通过 relational DB 进行存储和管理。
- 非结构化：数据结构不规则或不完整，没有预定义的数据模型，不能用二维表来表达。包括文本、图片、HTML、各类报表、音视频信息等。
- 半结构化：虽不符合 relational DB 的 data model，但包含用来分隔语义元素 / 对记录和字段进行分层的相关标记。数据格式不固定，如json，同一键值下可能存储数值/文本/字典/列表。

零散知识点

DBMS支持null：

- 因为：某些数据不确定、不知道、不存在。
- DBMS需要在查询处理中支持True False Null的三值逻辑，需要针对空值的实体完整性约束、引用完整性约束等。

deadlock：

- wait-die： TA想要TB的资源，如果TA比TB old 就等待；否则abort，并且用原来的时间戳retry。
- wound-wait： TA想要TB的资源，如果TA比TB young 就等待；否则kill掉TB，抢掉TB的资源，TB用原来的时间戳retry。

通用计算方法 conceptual evaluation：

- 计算所有 relation_list 的 cross-product。
- discard 不满足 qualification 的 tuples。
- 删掉不在 target_list 里的 attributes。
- 如果 DISTINCT，eliminated duplicate rows。

统计数据库的安全：

- 限制统计查询的tuple的数量：【$b \le |SET(T)| \le (n-b)$】
- 通用追踪器，general tracker：【$2b \le |SET(T)| \le (n-2b), b < n/4$】，查到任意集合：$SET(p) = SET(p \text{ or } T) \cup SET(p \text{ or not } T) - SET(T) - SET(\text{not } T)$。

水手的师傅是水手这张表的外键，自己引用自己。

ER图：参与度描述的是，实体可以参与多少个这种联系。